HSD
Hochschule Düsseldorf
University of Applied Sciences

Fachbereich Medien
Faculty of Media

Entwicklung individueller Ansätze für die Anwendung von Secure Coding Richtlinien für Java basierend auf statischer Code-Analyse

Masterarbeit

im Studiengang Medieninformatik
zur Erlangung des akademischen Grades
Master of Science

Fachbereich Medien
Hochschule Düsseldorf

Marc Treiber
Matrikel-Nr.: [redacted]
Datum: April 2020

Erst- und Zweitprüfer

Prof. Dr.-Ing. Holger Schmidt
Prof. Dr.-Ing. Markus Dahm

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Masterarbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Die verwendeten Quellen sind vollständig zitiert. Diese Arbeit wurde weder in gleicher noch ähnlicher Form einem anderen Prüfungsamt vorgelegt oder veröffentlicht. Ich erkläre mich ausdrücklich damit einverstanden, dass diese Arbeit mittels eines Dienstes zur Erkennung von Plagiaten überprüft wird.

Ort, Datum *Marc Treiber*

Kontaktinformationen
Marc Treiber
—
marc.treiber@hs-duesseldorf.de

Zusammenfassung

Entwicklung individueller Ansätze für die Anwendung von Secure Coding Richtlinien für Java basierend auf statischer Code-Analyse

Marc Treiber

Die Sicherheit von Softwareprojekten ist ein zentraler Faktor für ihren Erfolg, der oft nicht ausreichend gewährleistet wird. Aus diesem Grund werden in der vorliegenden Arbeit neue Ansätze entwickelt, um dem Entwickler bei der Durchsetzung der Softwaresicherheit zu helfen. Dafür wird untersucht, welche neuen Ansätze für die Unterstützung bei der Vermeidung ausgewählter Fehler hilfreich sein können, sowie welche konzeptionellen Vor- und Nachteile sie aufweisen.

Zunächst wird ein Überblick über die nötigen Grundlagen für die Entwicklung der neuen Ansätze geschaffen. Elementar ist dabei das Secure Coding Prinzip, nach dem verschiedenste sicherheitsrelevante Implementierungsfehler bereits früh in dem Lebenszyklus eines Softwareprojekts, während seiner Implementierung zu vermeiden sind. Ein Mittel für diesen Zweck sind die sogenannten Secure Coding Richtlinien, die typische Fehler beschreiben und konkrete Vorgehensweisen für ihre Vermeidung und Behebung definieren, aber oft nicht eingehalten werden. Werkzeuge zur statischen Code-Analyse können hierbei Abhilfe schaffen, da sie verschiedenste Fehler automatisiert erkennen und den Entwickler über sie informieren.

Auf der Basis dieser Grundlagen werden Verbesserungsmöglichkeiten untersucht, um den Entwickler anhand von statischer Code-Analyse besser bei der Einhaltung relevanter Secure Coding Richtlinien zu unterstützen. Der Fokus dieser Arbeit liegt dabei auf der Vermeidung von Fehlern bei der Validierung von Eingaben, da diese Fehlerform besonders weit verbreitet und schwerwiegend ist.

Um eine entsprechende Hilfestellung gewährleisten zu können, werden drei Ansätze zur Unterstützung bei der Anwendung von Richtlinien für die Programmiersprache Java näher betrachtet. Diese Ansätze helfen dem Entwickler komplementär zu der Funktionalität herkömmliche Werkzeuge zur statischen Code-Analyse für das Auffinden von Fehlern auch bei der Vermeidung und Behebung dieser Fehler. Die neuen Ansätze basieren auf der Verwendung von textuellen Annotationen für die Umsetzung des Validierungsprozesses einer Eingabe, dem Einsatz einer API zur Validierung einer Eingabe und der Nutzung eines IDE-Plugins, um den nötigen Quellcode für den Validierungsprozess automatisiert zu generieren. Die nähere Betrachtung und Evaluierung dieser Ansätze zeigt, dass der annotationsbasierte Ansatz sich nicht für diesen Zweck eignet, während sowohl der API-basierte Ansatz, als auch der Ansatz für das Generieren des Quellcodes eine effektive und effiziente Ergänzung herkömmlicher Werkzeuge zur statischen Code-Analyse darstellen und dementsprechend in der weiteren Forschung aufgegriffen werden sollten.

Abstract

Development of individual approaches to utilize secure coding guidelines for Java based on static code analysis

Marc Treiber

Software security is a defining factor for the success of any software, that is often not sufficiently enforced. On this account, several new approaches to assist the developer with the implementation of software security are developed in the thesis at hand. In doing so, new approaches that might aid in the prevention of selected mistakes are explored and their conceptual advantages and disadvantages are evaluated.

Initially, an outline of the needed fundamentals for the development of new approaches is established. In this regard, the Secure Coding Principle is elementary in its ability to prevent security-relevant errors already during the implementation phase of the Software Development Life Cycle. A tool for this purpose are Secure Coding Guidelines, which depict specific mistakes and define best practices for their prevention and remediation, but are often not adhered to. Static Code Analysis Tools can help with the development of secure software, as they can automatically detect various types of such mistakes and report them to the developer.

Based on these fundamentals, options for improvement are examined to better aid the developer in the appliance of appropriate Secure Coding Guidelines with the help of static code analysis. More specifically, this thesis is focused on preventing mistakes in the area of input validation, as this type of mistake is widely spread and may result in severe vulnerabilities.

In order to provide adequate assistance, three approaches to aid in the appliance of Secure Coding Guidelines for the Java Programming Language are explored. These approaches complement the common functionality of Static Code Analysis Tools for the detection of errors by also helping with their prevention and remediation. The new approaches are based around the ideas of using textual annotations to define the validation process for an input, using APIs to validate inputs and employing a plugin for an IDE to automatically generate the required source code for the process of input validation. On closer examination of these approaches, it becomes apparent that the usage of annotations is not suited for this purpose, whereas both the API based approach, as well as the idea of generating the needed source code can provide effective and efficient extensions of traditional Source Code Analysis Tools and should be closer examined in further research.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Kontext 1
1.2 Motivation 1
1.3 Thema der Arbeit 2
1.4 Wahl des Szenarios 4
1.5 Wissenschaftlicher Beitrag 4
1.6 Ziele der Arbeit 5
1.7 Weiterer Aufbau der Arbeit 5

2 Grundlagen des Secure Coding **7**
2.1 Terminologie 7
2.1.1 Asset 7
2.1.2 Schwachstelle 7
2.1.3 Verwundbarkeit 8
2.1.4 Bedrohung 8
2.1.5 Risiko 8
2.1.6 Angriff 8
2.1.7 Safety und Security 8
2.2 Thematische Einordnung des Secure Coding 8
2.3 Definition des Secure Coding Begriffs 11
2.4 Relevanz des Secure Coding 13
2.5 Methoden zur Gewährleistung sicheren Codes 15
2.6 Sicherheit in Java 17

3 Secure Coding Richtlinien **24**
3.1 Definition des Richtlinien-Begriffs 24
3.2 Quellen für Secure Coding Richtlinien 27
3.3 Relevanz von Secure Coding Richtlinien 29
3.4 Beispiele für Secure Coding Richtlinien 30
3.5 Gründe für die Missachtung von Secure Coding Richtlinien 31

4 Statische Code-Analyse **32**
4.1 Grundlagen statischer Code-Analyse 32
4.2 Typische Anwendungsgebiete 34
4.3 Vor- und Nachteile statischer Code-Analyse 35
4.4 Allgemeine Funktionsweise von Analysewerkzeugen 40
4.5 Beispiele für traditionelle Analysewerkzeuge 48

5 Eingrenzung des Schwerpunktes für die Entwicklung neuer Ansätze 50
5.1 Vorgehensweise . . . 50
5.2 Existierende nicht-traditionelle Ansätze . . . 52
5.2.1 ASIDE . . . 52
5.2.2 CQUAL . . . 54
5.2.3 SpotBugs Annotationen . . . 56
5.2.4 Composable Thread Coloring . . . 58
5.2.5 SpongeBugs . . . 59
5.2.6 FixBugs . . . 60
5.2.7 CogniCrypt . . . 61
5.2.8 Beobachtungen . . . 64
5.3 Betrachtete Richtlinien . . . 64
5.3.1 Bildung relevanter Kategorien . . . 64
5.3.2 Kriterien zur Auswahl der Richtlinien . . . 67
5.3.3 Auswahl der Richtlinien . . . 68
5.4 Definition des Schwerpunktes . . . 70

6 Entwicklung individueller Ansätze zur Anwendung von Richtlinien 71
6.1 Einleitung . . . 71
6.2 Eingabevalidierung . . . 73
6.2.1 Grundlagen der Eingabevalidierung . . . 73
6.2.2 Eingabevalidierung am Beispiel von Richtlinien . . . 74
6.2.3 Folgerungen für die Entwicklung neuer Ansätze . . . 76
6.3 Existierende Konzepte zur Umsetzung der Eingabevalidierung . . . 77
6.3.1 Einschränkungsbasierte Konzepte . . . 77
6.3.2 API-basierte Konzepte . . . 79
6.3.3 Beobachtungen . . . 80
6.4 Auswahl grundlegender Konzepte . . . 80
6.5 Ansatz 1: Annotationsbasierte Validierung . . . 81
6.5.1 Konzept . . . 81
6.5.2 Erste Beobachtungen . . . 82
6.6 Ansatz 2: APIs zur Validierung . . . 85
6.6.1 Konzept . . . 85
6.6.2 Funktionsweise an praktischen Beispielen . . . 86
6.6.3 Erste Beobachtungen . . . 93
6.7 Ansatz 3: Generieren von Quellcode . . . 94
6.7.1 Konzept . . . 94
6.7.2 Funktionsweise an praktischen Beispielen . . . 96
6.7.3 Erste Beobachtungen . . . 103

7 Evaluierung 106
7.1 Definition des Kriterienkatalogs . . . 106
7.2 Evaluierung der entwickelten Ansätze . . . 107
7.2.1 Effektivität . . . 107
7.2.2 Effizienz . . . 109
7.2.3 Sonstige Kriterien . . . 111
7.2.4 Einschätzung der Ansätze . . . 112

8 Fazit **114**
8.1 Zusammenfassung 114
8.2 Ausblick 116

A Inhalt der Begleit-CD **119**

B Hinweise zur Ausführung der entwickelten Proof of Concepts **120**
B.1 Ansatz 2: SCAT-basierte APIs 120
B.2 Ansatz 3: Generierung von Quellcode 121
B.3 Aufsetzen der MySQL-Datenbank 122

Abbildungsverzeichnis

1.1 Exemplarische Analyseergebnisse des Werkzeugs SonarQube 3

2.1 Darstellung der CIA-Triade 9
2.2 Software Security Best Practices nach McGraw 10
2.3 Taxonomie sicherheitsbezogener Attribute 12
2.4 Anzahl erfasster CVE-Einträge der letzten 20 Jahre 14
2.5 Verschiedene Komponenten der Java-Plattform 19
2.6 Zusammenspiel von Code, Class Loader und Security Policy in Java . 20
2.7 Beispielhafter Eintrag einer Security Policy in Java 21
2.8 Prozentueller Anteil an speicherbasierten Verwundbarkeiten in Microsoft-Produkten von 2006 bis 2018 22

3.1 Unterscheidung verschiedener Formen von Secure Coding Richtlinien 27

4.1 Laufzeit von Werkzeugen zur statischen Code-Analyse 35
4.2 Beispiel für ein False-Positive Ergebnis in SonarLint 37
4.3 Funktionsweise von Werkzeugen zur statischen Code-Analyse 41
4.4 Beispiel für einen Parsebaum 43
4.5 Beispiel für einen Abstract Syntax Tree 44
4.6 Beispiel für einen Aufrufgraphen 45
4.7 Beispiel für einen Kontrollflussgraphen 45
4.8 Zusammenhang lokaler und globaler Analysealgorithmen 46
4.9 Aufbau und Ergebnisdarstellung des Analysewerkzeugs SpotBugs . . 49
4.10 Fehlerdarstellung in dem Analysewerkzeug SonarLint 49

5.1 Kontextmenü in ASIDE 53
5.2 Grafische Annotationen in ASIDE 54
5.3 Beispiel für eine Warnung durch Annotationen in SpotBugs 57
5.4 Beispiel für eine vollautomatische Korrektur durch SpongeBugs . . . 59
5.5 Beispiel für eine Korrektur für Exception-Handling in FixBugs 60
5.6 Beispiel für eine Korrektur für Streams in FixBugs 61
5.7 Beispiel für einen automatisch erkannten Fehler in CogniCrypt 62
5.8 Angabe der Aufgabe im Assistenten von CogniCrypt 63
5.9 Angabe von Zusatzinformationen im Assistenten von CogniCrypt . . 63
5.10 Zusammenhänge wichtiger Verwundbarkeiten und Schwachstellen . . 66

6.1 Beschreibung einer Verwundbarkeit für SQL-Injektionen in SpotBugs 71
6.2 Beschreibung einer Verwundbarkeit für SQL-Injektionen in SonarQube 72
6.3 Komponenten für einen Ansatz zur annotationsbasierten Validierung . 82
6.4 Komponenten für einen API-basierten Ansatz 85

6.5 Checker Framework Warnung für eine fehlende Validierung 87
6.6 Komponenten für einen Ansatz zum Generieren von Quellcode 95
6.7 Kontextmenü zum Aufruf eines Assistenten zur Quellcode-Generierung 97
6.8 Assistent zum Generieren von Quellcode für die Validierung eines Strings 98
6.9 Checker Framework Warnung für mögliche SQL-Injektionen 100
6.10 Generieren einer Vorlage zur Vermeidung von SQL-Injektionen 101
6.11 Generieren einer Lösung zur Vermeidung von SQL-Injektionen 102
6.12 Generieren einer Lösung zur Vermeidung von XML-Injektionen 103

B.1 Einbinden neuer Annotationen für das Checker Framework in IntelliJ 121

Tabellenverzeichnis

5.1 Priorisierung von Kategorien für Schwachstellen und Verwundbarkeiten 67
5.2 Priorisierte CERT-Richtlinien zur Eingabevalidierung 69

6.1 Umfang des Proof of Concepts des API-basierten Ansatzes 86
6.2 Umfang des Proof of Concepts des Ansatzes für das Generieren des Quellcodes 96

7.1 Kriterienkatalog zur Evaluierung der entwickelten Ansätze 107

Listings

3.1 Beispiel für Quellcode der für eine SQL-Injektion anfällig ist 30
4.1 Java-Quellcode zur Veranschaulichung einer lexikalischen Analyse . . 42
4.2 Beispielhafte Regeln für eine lexikalische Analyse 42
4.3 Beispielhafte Produktionsregeln für einen Sprachparser 43
4.4 Quellcode ohne garantierte Bereinigung einer Benutzereingabe 44
4.5 Quellcodebeispiel für das Prinzip von Taint Propagation Regeln . . . 47
5.1 Definition von Qualifikatoren in CQUAL 55
5.2 Annotieren von Signaturen in CQUAL 55
5.3 Beispiel für die Verwendung von CQUAL in Quellcode 55
5.4 Beispiel für die Verwendung von Annotationen in SpotBugs 56
5.5 Quellcodebeispiel für Composable Thread Coloring 58
6.1 Definition einer validierbaren JavaBean 78
6.2 Durchführung einer programmatischen Validierung einer JavaBean . . 78
6.3 Unterbindung von SQL-Injektionen anhand von ESAPI 79
6.4 Quellcode für das Konzept eines annotationsbasierten Ansatzes 83
6.5 Für SQL-Injektionen anfälliger Quellcode 83
6.6 Quellcode für die Vermeidung von SQL-Injektionen 84
6.7 Beispiel für die Verwendung der Annotationen des Checker Frameworks 87
6.8 Verwendung des API-basierten Ansatzes zur Validierung einer Eingabe 88
6.9 Markierung des Validierungsprozesses durch Annotationen 89
6.10 Quellcode in dem das Checker Framework eine SQL-Injektion erkennt 89
6.11 Verwendung des API-basierten Ansatzes zur Vermeidung von SQL-Injektionen 90
6.12 Verwendung des API-basierten Ansatzes zur Vermeidung von XML-Injektionen 91
6.13 Inhalt der Datei xmlSchema.xsd 92
6.14 Quellcode ohne Eingabevalidierung eines Strings 97
6.15 Stub-Datei zur Erkennung einer Verwundbarkeit für SQL-Injektionen 99
6.16 Quellcode mit einer Verwundbarkeit für SQL-Injektionen 99
A.1 Inhalt der Begleit-CD 119
B.1 Erstellen einer neuen MySQL Datenbank 122
B.2 Importieren des beigelegten Datenbank-Backups 122
B.3 Inhalt des beigelegten Datenbank-Backups 122

Abkürzungsverzeichnis

API Application Programming Interface
ASIDE Application Security plug-in for the Integrated Development Environment
AST Abstract Syntax Tree
AUTOSAR Automotive Open System Architecture
BSA Business Software Alliance
BSI Bundesamt für Sicherheit in der Informationstechnik
CERT Computer Emergency Response Team
CLASP Comprehensive, Lightweight Application Security Process
CSRF Cross-Site-Request-Forgery
CWE Common Weakness Enumeration
DoS Denial-of-Service
ESAPI Enterprise Security API
HQL Hibernate Query Language
IDE Integrated Development Environment
JCA Java Cryptography Architecture
JDBC Java Database Connectivity
JPA Java Persistence API
JRE Java Runtime Environment
MISRA Motor Industry Software Reliability Association
NVD National Vulnerability Database
ORM Object-Relational Mapping
OWASP Open Web Application Security Project
PKI Public-Key-Infrastruktur
PSI Program Structure Interface
SAMM Software Assurance Maturity Model
SANS SysAdmin, Networking and Security
SCALe Source Code Analysis Laboratory
SCAT Static Code Analysis Tool
SE Standard Edition
SEI Software Engineering Institute
SSO Security Sensitive Operation
TLS Transport Layer Security
UML Unified Modeling Language
VM Virtual Machine

Kapitel 1

Einleitung

Dieses Kapitel beschreibt die grundlegende Thematik dieser Arbeit sowie ihre Relevanz. Weiterhin werden ein konkretes Szenario als Rahmenbedingung und die zu erreichenden Ziele definiert.

1.1 Kontext

Die Sicherheit von Softwareprojekten ist ein maßgeblicher Faktor für ihren Erfolg: Versäumnisse bei der Umsetzung sicherer Software können schnell zu juristischen Folgen, finanziellen Verlusten oder auch Personenschäden führen (de Bruhin & Janssen, 2017, Seite 1). In einer Studie des Ponemon Institute (2018, Seite 3) wurden die Verletzungen der Datensicherheit innerhalb eines Jahres in 477 Firmen untersucht. Hierbei wurde festgestellt, dass die Kosten einer solchen Verletzung von 2017 bis 2018 um 6,4% auf durchschnittlich 3.86 Millionen US-Dollar angestiegen sind - die höchsten Schäden entstehen dabei unter anderem in den USA, Kanada und Deutschland. Die Menge der Angriffsversuche nimmt zudem zu, das Unternehmen Symantec verzeichnete im Rahmen einer Studie des Jahres 2019 einen Anstieg webbasierter Angriffe um 56% zum Vorjahr (Symantec, 2019, Seite 6). Sicherheitsbezogene Probleme in Software sind außerdem weit verbreitet, ein Bericht des Unternehmens Veracode (2019, Seite 8) hält etwa fest, dass 83% der 85000 untersuchten Anwendungen mindestens ein solches Problem aufweisen. Im Hinblick auf die Zukunft stellte das Bundesamt für Sicherheit in der Informationstechnik (BSI) in einer Umfrage unter 1039 teilnehmenden Institutionen fest, dass 88% der Teilnehmer eine weitere Verschärfung der Bedrohungslage im Rahmen der Digitalisierung erwarten (BSI, 2019, Seite 7). Die Vermeidung von Schwachstellen und Verwundbarkeiten im Allgemeinen wie auch speziell in dem Quellcode eines Programms ist daher ein enorm wichtiger Aspekt in der Softwareentwicklung.

1.2 Motivation

Viele der Schwachstellen und Verwundbarkeiten in Softwareprojekten entstehen durch Entwicklungsfehler im Zuge der Programmierung. Best Practices, wie etwa die Secure Coding Richtlinien des Kodierungsstandards des Computer Emergency Response

Team (CERT) für Java[1], existieren für verschiedenste Programmiersprachen und beschreiben oft einfache Maßnahmen, um Schwachstellen und Verwundbarkeiten zu vermeiden. Sie werden jedoch nicht immer angewandt, da sie vielen Entwicklern aufgrund fehlender Kenntnisse hinsichtlich der Programmierung sicherer Software nicht geläufig sind und weil ihre konsistente Anwendung zudem anspruchsvoll ist (Lipford, Thomas, Chu & Murphy-Hill, 2014, Seite 17).

Aus diesem Grund existieren verschiedene teil- oder vollautomatisierte Ansätze, die Softwareentwicklern dabei helfen, ihren Programmcode durch die Einhaltung solcher Best Practices sicherer zu gestalten. Traditionelle Werkzeuge zur statischen Code-Analyse wie SonarQube[2] oder SpotBugs[3] - der Nachfolger von FindBugs[4] - mit dem dazugehörigen Plugin für die Erkennung von Sicherheitsproblemen[5] versuchen beispielsweise unterschiedlichste Arten von Verwundbarkeiten automatisch aufzuspüren. Das *Application Security plug-in for the Integrated Development Environment (ASIDE)* nach Lipford et al. (2014) verfolgt hingegen einen weniger traditionellen, interaktiven Ansatz, bei dem der Entwickler durch die Verwendung grafischer Annotationen auf mögliche Verwundbarkeiten hingewiesen und bei deren Behebung unterstützt wird.

Viele dieser Werkzeuge liefern jedoch nur grobe Aussagen zu den verschiedenen Problemtypen, ohne etwa den relevanten Kontext des jeweiligen Programmcodes miteinzubeziehen und beschreiben das Problem lediglich auf eine eher abstrakte Art und Weise, ohne aber eine konkretere Hilfestellung bei seiner Behebung zu leisten. Abbildung 1.1 veranschaulicht dies an einem Beispiel: Ob und zu welchem Zweck der hier berechnete Zufallswert `r` verwendet wird, ist für die Bewertung durch das Werkzeug SonarQube nicht relevant. Da es sich hierbei aber um einen vorhersehbaren Wert handelt, wird seine Verwendung grundsätzlich als mögliches Sicherheitsrisiko erkannt und der Entwickler muss den konkreten Fall auf der Basis der bereitgestellten Problembeschreibung selbst einschätzen und die korrekten Anpassungen vornehmen.

Aus diesen Gründen ist es hilfreich festzustellen, inwiefern neue Ansätze zur Einhaltung ausgewählter Secure Coding Richtlinien, welche individuell auf deren Eigenschaften zugeschnitten sind und sich von dem Vorgehen herkömmlicher Werkzeuge zur statischen Code-Analyse wie SonarQube unterscheiden, realisiert werden können, um den Entwickler bei der Programmierung sicherer Software zu unterstützen. In diesem Zusammenhang ist auch relevant, inwiefern sich diese Ansätze für die Verwendung in der Praxis eignen und ob sie den Entwickler effizienter unterstützen können als herkömmliche Werkzeuge zur statischen Code-Analyse.

1.3 Thema der Arbeit

Im Rahmen der vorliegenden Masterarbeit zu dem Thema „Entwicklung individueller Ansätze für die Anwendung von Secure Coding Richtlinien für Java basierend auf statischer Code-Analyse“ werden neue Ansätze entwickelt und evaluiert, um den Pro-

[1] https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java, aufgerufen am 15.12.2019
[2] https://www.sonarqube.org/, aufgerufen am 15.12.2019
[3] https://spotbugs.github.io/, aufgerufen am 02.10.2019
[4] http://findbugs.sourceforge.net/, aufgerufen am 02.10.2019
[5] http://find-sec-bugs.github.io/, aufgerufen am 02.10.2019

Abbildung 1.1: Exemplarische Analyseergebnisse des Werkzeugs SonarQube

grammierer bei der Implementierung sicherer Software zu unterstützen, die für eine möglichst effektive Hilfe individuell auf die jeweiligen Eigenschaften konkreter Secure Coding Richtlinien zugeschnitten werden. Zudem sollen diese Ansätze eine statische Code-Analyse als Grundlage verwenden, um Fehlerquellen aufzufinden. Dynamische Testtechniken werden in dieser Arbeit damit nicht betrachtet.

Die zentralen Ziele sind somit festzustellen, welche individualisierten Ansätze unter der Verwendung statischer Code-Analyse denkbar sind und inwiefern sie sich für die Behandlung bestimmter Probleme besser eignen als herkömmliche Werkzeuge.

Zu diesem Zweck sind drei wesentliche Arbeitsschritte notwendig: In dem ersten Schritt werden die Grundlagen der sicheren Programmierung und ihrer konkreten Ausprägungen in der Form von Secure Coding Richtlinien zusammengefasst, um einen allgemeinen Überblick über die Relevanz dieser Thematik zu schaffen.

In dem darauffolgenden Schritt werden die verschiedenen Aspekte statischer Code-Analyse näher untersucht, um festzuhalten, anhand welcher grundlegenden Vorgehensweisen etwa Fehler in dem Programmcode einer Software automatisch erkannt oder anderweitige Zusatzinformationen aus dem Code gewonnen werden können. Zudem wird betrachtet, wozu statische Code-Analysen allgemeinhin verwendet werden, welche Vor- und Nachteile sie aufweisen, wie sie genutzt werden können, um die Sicherheit einer Software zu verbessern und inwiefern sie Verbesserungspotenzial aufweisen.

In dem letzten Schritt werden vergleichbare bereits existierende Ansätze untersucht, um Mittel und Konzepte zu identifizieren, welche die neuen Ansätze verwenden können, um den Entwickler bei dem Verfassen sicheren Quellcodes zu unterstützen. Basierend auf diesen Ergebnissen werden die neuen Ansätze entwickelt und anschließend evaluiert, um ihre Praxistauglichkeit einzuschätzen.

1.4 Wahl des Szenarios

Als Szenario wird die Analyse und Behandlung von sicherheitsrelevanten Fehlern für die Entwicklung in der Programmiersprache Java gewählt. Java bietet sich an dieser Stelle an, da es sich hierbei um eine High-Level Programmiersprache handelt, mit der Software für verschiedenste Anwendungsgebiete wie Smartphone-Apps oder Desktop- und Webanwendungen entwickelt wird. Java hat außerdem im Rahmen des Internet of Things auf verschiedensten Geräten, wie beispielsweise Haushaltsgeräten oder in Fahrzeugelektronik Einzug erhalten (Long, Mohindra, Seacord, Sutherland & Svoboda, 2013, Seite xiii). Weiterhin ist Java eine ausgereifte und sehr weit verbreitete Sprache, die auch Konstrukte wie Annotationen nativ unterstützt und etwa nach dem TIOBE Index[6] als aktuell populärste Programmiersprache gilt. Auch auf der Liste der am meisten verwendeten Sprachen auf GitHub steht Java an dritter Stelle[7]. In einer Studie des Unternehmens Veracode wurde weiterhin festgestellt, dass knapp 86% aller Anwendungen, die in Java geschrieben wurden mindestens ein sicherheitsrelevantes Problem aufweisen (Veracode, 2019, Seite 24). Aus diesen Gründen und weil Java eine in ihrer Benutzung recht einfache Sprache ist, eignet sie sich gut als Wahl, um neue Ansätze zu entwickeln, die ggf. auch für andere Sprachen adaptiert werden können.

Die Verwendung von Java bietet sich zudem an, da durch den Kodierungsstandard des CERT[8] eine umfangreiche Liste an gut dokumentierten Secure Coding Richtlinien bereitgestellt wird. Durch die konsistente Anwendung der hier beschriebenen Richtlinien hätten auch verschiedene bekannte Schwachstellen vermieden werden können. Der Internet-Browser Firefox für das Betriebssystem Android verwendete etwa zeitweise das schwache Seeding von `Math.random`[9], ein Punkt der prinzipiell auch durch die Richtlinie *MSC02-J. Generate strong random numbers* behandelt wird. Ein weiteres Beispiel ist die Software SAP NetWeaver, welche es dem Anwender ermöglichte Dateien hochzuladen, ohne dabei das Dateiformat zu validieren[10]. Mit dieser Problematik befasst sich beispielsweise die Richtlinie *IDS56-J. Prevent arbitrary file upload.*

1.5 Wissenschaftlicher Beitrag

Eine Verbesserung der wissenschaftlichen Situation auf dem Gebiet der Entwicklung sicherer Software soll im Rahmen dieser Arbeit zunächst erreicht werden, indem ein Überblick über mögliche alternative Konzepte zu denen herkömmlicher Werkzeuge für das Auffinden von Schwachstellen und Verwundbarkeiten geschaffen wird. Diese Konzepte sollen als individuelle Ansätze näher betrachtet und auf einige ausgewählte Sicherheitsprobleme bezogen exemplarisch umgesetzt werden, um zu veranschaulichen, wie sie den Entwickler bei dem Schreiben sicheren Quellcodes unterstützen können, sowie welche Vor- und Nachteile sie aufweisen. So ist festzustellen, inwiefern effizientere Ansätze als die herkömmlicher Werkzeuge realisierbar sind, um die Durchsetzung verschiedener Secure Coding Richtlinien zu gewährleisten

[6]https://www.tiobe.com/tiobe-index/, aufgerufen am 15.12.2019
[7]https://octoverse.github.com/, aufgerufen am 15.12.2019
[8]https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java, aufgerufen am 15.12.2019
[9]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1516, aufgerufen am 19.07.2019
[10]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-0327, aufgerufen am 15.12.2019

Die Ergebnisse der Arbeit können damit verwendet werden, um neue Werkzeuge unter der Verwendung der praxistauglichen Ansätze umzusetzen, die den Entwickler effektiv und effizient bei dem Verfassen sicheren Quellcodes unterstützen oder um bestehende Werkzeuge entsprechend zu ergänzen. Ansätze, die sich im Rahmen dieser Arbeit als nicht sinnvoll herausstellen, können zudem in der weiteren Forschung ignoriert werden.

1.6 Ziele der Arbeit

Das grundlegende Ziel dieser Masterarbeit ist festzustellen, inwiefern die verschiedenen Mittel und Methoden der statischen Code-Analyse genutzt werden können, um den Softwareentwickler anhand von Ansätzen, die sich von denen herkömmlicher Werkzeuge zur statischen Code-Analyse unterscheiden, möglichst effizient bei der Anwendung von Secure Coding Richtlinien zu unterstützen, um so die Softwaresicherheit zu fördern. Zu diesem Zweck wird eine Reihe an Teilzielen definiert:

1. Die Grundlagen und die Relevanz sicherer Programmierung im Allgemeinen, sowie etwaige Besonderheiten in Java müssen herausgearbeitet werden.
2. Die grundlegenden Möglichkeiten zur Hilfe bei der Umsetzung sicheren Programmcodes, wie etwa existierende Best Practices und Secure Coding Richtlinien sind zu betrachten.
3. Die verschiedenen Aspekte, Methoden und Vorgehensweisen von Werkzeugen zur statischen Code-Analyse müssen aufgearbeitet werden.
4. Eine Reihe von individuellen Ansätzen für die Einhaltung einiger ausgewählter Richtlinien soll basierend auf den Möglichkeiten der statischen Code-Analyse konzipiert werden.
5. Eine Teilmenge der entwickelten Ansätze soll im Stile eines Proof of Concepts praktisch umgesetzt werden.
6. Die entwickelten Ansätze sollen im Hinblick auf ihre Praxistauglichkeit evaluiert werden.

1.7 Weiterer Aufbau der Arbeit

In dem Hauptteil der Arbeit werden zunächst die nötigen Grundlagen für diese Thematik zusammengefasst und die verwandten Themen betrachtet, bevor die eigentliche Entwicklung der neuen Ansätze erfolgt. Sämtliche Literaturrecherchen im Rahmen dieser Arbeit werden durchgeführt, indem zunächst wissenschaftliche Quellen für die relevanten Themen anhand von Schlagwörtern wie beispielsweise „Informationssicherheit“, „Secure Coding“, „Static Code Analysis“ und ihrer Synonyme gesucht werden. Hierfür werden im Internet verfügbare Werkzeuge wie Google[11], IEEE Xplore[12] oder Researchgate[13] verwendet. Ausgehend von den so gefundenen Quellen wird

[11] https://www.google.de/, aufgerufen am 16.04.2020
[12] https://ieeexplore.ieee.org/, aufgerufen am 16.04.2020
[13] https://www.researchgate.net/, aufgerufen am 16.04.2020

ein Schneeballsystem genutzt, um anhand der jeweiligen Autoren, der verwendeten Begriffe und der referenzierten Quellen weitere Quellen, Schlagwörter und verwandte Themengebiete zu finden und um die gefundenen Quellen anhand von Querverweisen auf ihre Glaubwürdigkeit zu prüfen.

In Kapitel 2 werden die für das Thema der Arbeit notwendigen Grundlagen der sicheren Programmierung etabliert. Die zentralen Punkte dieses Kapitels sind die Definition des Secure Coding Begriffs sowie seine Relevanz, die konkreten Möglichkeiten, um die Sicherheit des Programmcodes zu fördern und die Sicherheitsarchitektur der Programmiersprache Java.

Darauffolgend befasst sich Kapitel 3 genauer mit dem Konzept der Secure Coding Richtlinien als Hilfsmittel zur Gewährleistung der Sicherheit des Quellcodes. Hier wird unter anderem beschrieben, welche Formen von Richtlinien existieren, welche Quellen Richtlinien bereitstellen und welche dieser Quellen hier relevant sind.

Kapitel 4 beschreibt die Grundlagen der statischen Code-Analyse. An dieser Stelle werden zunächst ihre wesentlichen Merkmale zusammengefasst und es wird beschrieben, wodurch sich die statische Code-Analyse von dynamischen Tests unterscheidet. Weiterhin werden hier die grundlegenden Vorgehensweisen für statische Code-Analysen erläutert und es wird beschrieben, zu welchen typischen Verwendungszwecken diese genutzt werden. So wird auch veranschaulicht, wie traditionelle Werkzeuge zur statischen Code-Analyse bei der Einhaltung von Secure Coding Richtlinien helfen.

Im Anschluss daran wird in Kapitel 5 untersucht, welches Verbesserungspotenzial die zuvor genannten traditionellen Analysewerkzeuge bieten sowie welche nicht-traditionellen Ansätze zur Verwendung statischer Code-Analyse existieren und welche Richtlinien als Schwerpunkt für die Entwicklung der neuen Ansätze gewählt werden.

Daraufhin wird in Kapitel 6 die übergeordnete Thematik der als Schwerpunkt gewählten Richtlinien näher beschrieben. Zudem werden die neuen Ansätze zunächst auf einer theoretischen Ebene betrachtet, um so festzustellen, inwiefern diese sich für die Unterstützung des Entwicklers bei der Anwendung der ausgewählten Richtlinien eignen. Diejenigen Ansätze, die sich als vielversprechend herausstellen, werden zusätzlich anhand eines implementierten Proof of Concepts veranschaulicht. So kann konkreter dargestellt werden, inwiefern die entwickelten Ansätze praktisch umsetzbar sind und wie sie bei der Anwendung der ausgewählten Richtlinien helfen können.

Eine Evaluierung der entwickelten Ansätze wird in Kapitel 7 durchgeführt. So wird an dieser Stelle herausgearbeitet, wie effektiv und effizient die Ansätze den Entwickler bei der Anwendung verschiedener Secure Coding Richtlinien zur Entwicklung sicherer Software unterstützen, ob und inwiefern sie sich zu diesem Zweck besser eignen als herkömmliche Analysewerkzeuge und welche Nachteile sie aufweisen.

Abschließend werden in Kapitel 8 die Ergebnisse der Arbeit zusammengefasst und es wird ein Ausblick darauf gegeben, wie sie in der weiteren Forschung verwendet werden können, beispielsweise welche Unterthemen näher betrachtet werden sollten oder welche Ansätze weiterverfolgt werden sollten.

Kapitel 2

Grundlagen des Secure Coding

In diesem Kapitel werden zunächst die Grundlagen für ein besseres Verständnis der Thematik der sicheren Programmierung etabliert. Zu diesem Zweck wird diese Thematik innerhalb des Forschungsgebietes der Informationssicherheit eingegrenzt und der Begriff des *Secure Coding* wird definiert. Weiterhin wird verdeutlicht, warum die sichere Programmierung eine wichtige Thematik für die Informationssicherheit ist und wie ihre Durchsetzung gewährleistet werden kann. Abschließend werden die zentralen sicherheitsspezifischen Eigenschaften der für den weiteren Verlauf dieser Arbeit besonders relevanten Programmiersprache Java zusammengefasst.

2.1 Terminologie

In diesem Abschnitt werden einige für die Thematik der Informationssicherheit besonders wichtige Begriffe erläutert. Dies ist notwendig, da nicht alle der hier verwendeten Begrifflichkeiten einheitlich in der Literatur verwendet werden (Eckert, 2014, Seite 3).

2.1.1 Asset

Im Kontext der Informationssicherheit beschreibt der Begriff der *Assets* die Güter und Werte einer Organisation, welche adäquat zu schützen sind (ISO/IEC 27000, 2018, Seite 11). Assets können somit beispielsweise in materieller Form wie finanziellen Ressourcen oder Hardware sowie immaterieller Form wie dem Ruf eines Unternehmens oder als Informationen existieren (Peltier, 2013, Seite xi).

2.1.2 Schwachstelle

Als *Schwachstelle* wird eine Schwäche eines Systems bezeichnet, welche beispielsweise während des Entwurfs oder der Implementierung in das System eingeführt wird und in einer Verwundbarkeit resultieren kann. Ein konkretes Beispiel für eine Schwachstelle ist unübersichtlicher Programmcode, der im Falle eines Angriffs zwar selbst nicht ausgenutzt werden kann, möglicherweise jedoch zu der Einführung einer Verwundbarkeit durch einen Implementierungsfehler als Folge des schlecht verständlichen Codes führt (ISO/IEC 27000, 2018, Seite 11).

2.1.3 Verwundbarkeit

Eine *Verwundbarkeit* beschreibt eine Schwachstelle, welche durch eine Bedrohung ausgenutzt werden kann, um die Sicherheit des Systems zu gefährden, etwa indem ein Angreifer diese ausnutzt, um unautorisierten Zugriff auf geschützte Informationen zu erlangen (ISO/IEC 27000, 2018, Seite 11).

2.1.4 Bedrohung

Der Begriff der *Bedrohung* beschreibt eine Gefahr, welche unter der Ausnutzung einer Verwundbarkeit das System oder die Organisation schädigen kann, etwa indem die Vertraulichkeit, Integrität oder Verfügbarkeit der Assets gefährdet wird (ISO/IEC 27000, 2018, Seite 10; Eckert, 2014, Seite 17).

2.1.5 Risiko

Als *Risiko* definiert die ISO/IEC 27000 (2018, Seite 8) die Kombination aus der Eintrittswahrscheinlichkeit eines Ereignisses und seinen Konsequenzen, wie beispielsweise der Schwere des potenziell folgenden Schadens. Im Bereich der Informationssicherheit kann der Risikobegriff damit als Kombination der Wahrscheinlichkeit, mit welcher Bedrohungen die Verwundbarkeiten eines Systems ausnutzen, und dem daraus resultierenden Schaden für eine Organisation angesehen werden.

2.1.6 Angriff

Ein *Angriff* bezeichnet das unautorisierte Zugreifen oder den unautorisierten Zugriffsversuch auf Assets, um diese beispielsweise zu zerstören, veröffentlichen, verändern, zu sperren oder zu stehlen (ISO/IEC 27000, 2018, Seite 1).

2.1.7 Safety und Security

Die englischen Begriffe der *Safety* und der *Security* fallen in der deutschen Sprache unter den Ausdruck der *Sicherheit*. Die Unterscheidung beider Begriffe ist in der IT-Sicherheit jedoch besonders relevant: Der Begriff der Safety bezeichnet hier Systeme, welche unter allen normalen Betriebsbedingungen funktionieren und die Assets somit vor *versehentlicher* Gefährdung schützen. Der Begriff der Security bezeichnet im Gegensatz dazu die Funktionssicherheit eines IT-Systems im Bezug auf den Schutz vor Angriffen, etwa indem unautorisierte Zugriffe auf Informationen unterbunden werden. Die Sicherheit im Sinne der Security befasst sich daher mit dem Schutz der Assets vor *absichtlicher* Gefährdung (Eckert, 2014, Seite 6; D. Firesmith, 2004).

2.2 Thematische Einordnung des Secure Coding

Um die Bedeutsamkeit der Thematik der sicheren Programmierung erfassen zu können, muss diese zunächst in dem ihr übergeordneten Feld der *Informationssicherheit* eingeordnet werden. Der Grundgedanke der Informationssicherheit im Allgemeinen ist der Schutz der verschiedenen Assets einer Organisation oder eines Unternehmens wie beispielsweise ihrer Informationen, finanzieller oder anderer physischer Ressourcen,

der Reputation, ihrer Rechtsposition oder ihrer Angestellten (Peltier, 2013, Seite xi). Die Standards der ISO/IEC 27000 (2018, Seite 4) definieren drei zentrale Schutzziele, welche es für die Gewährleistung der Informationssicherheit zu erhalten gilt: *Confidentiality* (Vertraulichkeit), *Integrity* (Integrität) und *Availability* (Verfügbarkeit). Diese Schutzziele bilden die in Abbildung 2.1 dargestellte CIA-Triade (Andress, 2011, Seite 5). Die Komponenten der CIA-Triade werden im Folgenden genauer erläutert.

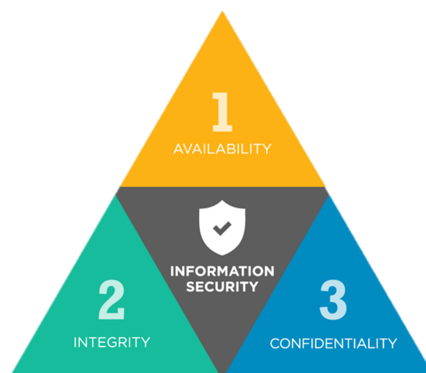

Abbildung 2.1: Die Schutzziele der Informationssicherheit bilden die CIA-Triade[1]

Das Schutzziel der Vertraulichkeit beschreibt die Eigenschaft eines Systems, dass der Zugriff auf Informationen sowohl an ihrem Speicherort als auch auf Transportwegen ausschließlich autorisierten Individuen, Instanzen und Prozessen ermöglicht werden darf (ISO/IEC 27000, 2018, Seite 2; Schönefeld, 2011, Seite 15). Das zweite Schutzziel, Integrität, befasst sich mit der Wahrung der Korrektheit und Vollständigkeit von Informationen und Systemen, sodass diese nur durch autorisierte Parteien verändert werden dürfen, auch um undefinierte Systemzustände zu vermeiden (ISO/IEC 27000, 2018, Seite 5; Pfleeger, Pfleeger & Margulies, 2015, Seite 6) Das finale Schutzziel der Verfügbarkeit besagt, dass der Zugriff auf Informationen und Systeme den autorisierten Parteien zeitnah und zuverlässig gewährleistet werden muss (ISO/IEC 27000, 2018, Seite 2). Beispielsweise sollen Systeme Fehlerzuständen gegenüber robust sein, um auch im Falle eines solchen Fehlerzustandes weiterhin ihre Dienste anbieten zu können (Schönefeld, 2011, Seite 16).

Die *IT-Sicherheit* bildet ein wichtiges Teilgebiet der Informationssicherheit, welches sich speziell mit dem Schutz elektronisch verarbeiteter Informationen im Rahmen von soziotechnischen Systemen auseinandersetzt. Das bedeutet konkreter, dass sich die IT-Sicherheit zunächst mit der Maschine im Kontext ihrer Umgebung, welche sich insbesondere durch die soziale Teilkomponente in Form des Menschen auszeichnet, befasst (Zave & Jackson, 1997). Das Forschungsfeld der IT-Sicherheit nimmt aufgrund der zunehmenden Verbreitung und des schnellen Wandels von verschiedensten Technologien einen immer höheren Stellenwert ein. Dies äußert sich etwa dadurch, dass Informationstechnologie heute in nahezu allen Bereichen des Lebens wie etwa Gesundheit, Mobilität, Bildung, Unterhaltung und Logistik von zentraler Bedeutung ist und pervasive Technologien wie das Internet of Things zunehmend neue Anwendungsgebiete erschließen (Eckert, 2014, Seite v). An den verschiedenen Anwendungsgebieten kann die Bedeutsamkeit der IT-Sicherheit weiter verdeutlicht werden, so

[1] Abbildung entnommen von https://www.ibm.com/blogs/cloud-computing/2018/01/16/drive-compliance-cloud/, aufgerufen am 13.10.2019

wird Software beispielsweise für die Überwachung und Steuerung von kritischen Infrastrukturen wie Kraftwerken eingesetzt, womit die Technologien und Verfahren der IT-Sicherheit nicht nur für die jeweiligen Assets einer Organisation, sondern auch für die Gewährleistung der öffentlichen Sicherheit relevant sind (Eckert, 2014, Seite 1).

Die *Computersicherheit* bildet ein Teilgebiet der IT-Sicherheit, welches sich mit dem Schutz der technischen Teilkomponente der zuvor beschriebenen soziotechnischen Systeme in der Form von Computern auseinandersetzt (Pfleeger et al., 2015, Seite 60). Der IT-Sicherheitsexperte Dr. Gary McGraw benennt Verwundbarkeiten in Software als ein zentrales und kritisches Problem der Computersicherheit, das bereits weit verbreitet ist und im Zuge der zunehmenden Komplexität und Erweiterbarkeit von Software auch weiterhin an Relevanz gewinnt (McGraw, 2006, Seite 25).

In diesem Zusammenhang stellt McGraw (2006, Seite 40) zudem eine maßgebliche Unterscheidung zwischen den Begriffen *Anwendungssicherheit* und *Softwaresicherheit* als Teilgebiete der Computersicherheit auf: Die Anwendungssicherheit beschäftigt sich mit reaktiven Methoden für den Schutz der Software wie Virenscannern, Firewalls, Patches und Angriffserkennungssystemen, welche sich mit dem Finden und Beheben von Verwundbarkeiten befassen, nachdem die Software bereits ausgeliefert wurde und die jeweiligen Verwundbarkeiten möglicherweise sogar schon für Angriffe ausgenutzt wurden (Chess & West, 2007, Seite xxi). Die Softwaresicherheit verfolgt hingegen einen proaktiven Ansatz, bei dem die Software gemäß des Prinzips *„Building Security In“* schon von Beginn ihres Lebenszyklus an so entwickelt werden soll, dass sie Angriffen standhalten kann und auch in Angriffsszenarien weiterhin möglichst fehlerfrei funktioniert (McGraw, 2006, Seite 40). In dem in Abbildung 2.2 dargestellten Modell zur sicheren Software-Produktentwicklung am Beispiel des herkömmlichen Wasserfallmodells ordnet McGraw insbesondere die Phasen der Anforderungsanalyse, des Entwurfs, der Implementierung und des Tests dem Begriff der Softwaresicherheit zu, während die Phasen der Auslieferung und der Produktion unter den Begriff der Anwendungssicherheit fallen (McGraw, 2006, Seite 40; Schönefeld, 2011, Seite 16). An diesem Modell wird weiterhin ersichtlich, dass verschiedene Methoden zu allen Zeitpunkten im Entwicklungsprozess anzuwenden sind, um die Sicherheit eines Programms in allen Belangen gewährleisten zu können (Chess & West, 2007, Seite 20).

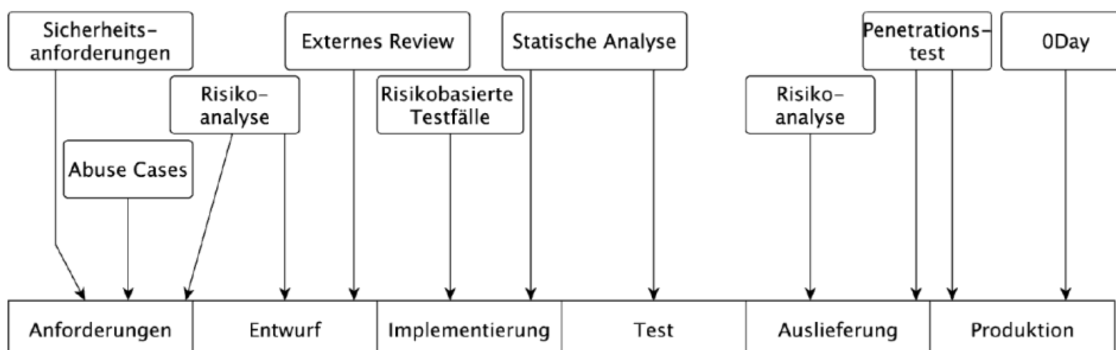

Abbildung 2.2: Verschiedene Software Security Best Practices in dem Modell zur sicheren Software-Produktentwicklung nach McGraw. Abbildung entnommen aus Schönefeld (2011, Seite 17).

Auf dem Gebiet der Softwaresicherheit teilen sich die sicherheitsbezogenen Probleme vornehmlich auf zwei zentrale Ebenen auf: Zum einen können Verwundbar-

keiten in der Architektur bzw. dem Entwurf der Software eingeführt werden, etwa in Form fehlender oder unsicherer Logging-Systeme, und zum anderen während der Implementierung der Software, beispielsweise durch inkorrekte Eingabevalidierungen (McGraw, 2006, Seite 34-38). Das Software Engineering Institute (SEI) schätzt, dass etwa 90% aller berichteten Sicherheitsvorfälle durch die Ausnutzung von Fehlern auf diesen Ebenen resultieren (DHS Cyber Security, 2013). Die Verwundbarkeiten teilen sich dabei nahezu gleichmäßig auf die Ebene der Architektur bzw. des Entwurfs sowie die Ebene der Implementierung auf. Heffley und Meunier (2004) stellten etwa fest, dass 64% der fast 2500 in der National Vulnerability Database (NVD) gelisteten Verwundbarkeiten im Jahre 2004 auf Implementierungsfehlern basieren, von denen wiederum 51% grundlegende Fehler sind.

In diesem Zusammenhang ist auch der Unterschied zwischen Softwaresicherheit und Sicherheitssoftware hervorzuheben: Die Anwendung von Sicherheitsfunktionalitäten wie etwa kryptographischen Algorithmen resultiert nicht automatisch in einer sicheren Software und auch die fehlerfreie Implementierung dieser Funktionalitäten ist nicht ausreichend. Stattdessen muss die Softwaresicherheit an allen Stellen im Quellcode durchgesetzt werden (Howard & Lipner, 2003, Seite 58). Bei der Verwundbarkeit CVE-2002-1363[2] handelt es sich beispielsweise um einen Fehler im Programmcode zur Bilddarstellung der Bibliothek libPNG, der zu Denial-of-Service (DoS)-Angriffen genutzt werden konnte und sich unter anderem auf die Browser Firefox, Opera und Safari auswirken konnte (Chess & West, 2007, Seite 8). Daher ist das Schreiben sicheren, verlässlichen und robusten Quellcodes in jedem Teil der Software, sodass Verwundbarkeiten bereits früh im Lebenszyklus der Software vermieden oder begrenzt werden, und die Software dementsprechend Angriffen proaktiv standhalten kann, eine elementare Komponente, um die Softwaresicherheit zu gewährleisten (McGraw, 2006, Seite 47). Diese Vorgehensweise wird häufig als *Secure Coding* oder *Secure Programming* bezeichnet (Bishop, 2012, Seite 1). Dementsprechend ist das Konzept des Secure Coding der Definition von McGraw nach auf dem Gebiet der Softwaresicherheit konkret auf der Implementierungsebene anzusiedeln.

2.3 Definition des Secure Coding Begriffs

Da es sich bei der Idee des Secure Coding um ein zentrales Thema dieser Arbeit handelt, muss der Begriff an dieser Stelle zunächst eindeutig definiert werden. Das Grundprinzip des Secure Coding basiert auf der Anwendung des zuvor beschriebenen Konzeptes *„Building Security In“* auf der Implementierungsebene. So werden Verwundbarkeiten im Quellcode der Software von vornherein ausgeschlossen, um Angriffe auf die Vertraulichkeit, Integrität und Verfügbarkeit von Beginn an zu vermeiden, anstatt Verwundbarkeiten erst im Nachhinein zu beheben (Schönefeld, 2011, Seite 51, 69). Die folgende Analogie von McGraw verdeutlicht das grundlegende Prinzip:

> „In the fight for better software, treating the disease itself is better than taking an aspirin to stop the symptoms.“ (McGraw, 2006, Seite 47)

Die Qualität eines Systems basiert zu einem großen Teil auf verschiedenen nichtfunktionalen Anforderungen in der Form von Qualitätsattributen, wie beispielsweise

[2]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-1363, aufgerufen am 16.12.2019

seiner Verlässlichkeit, Robustheit, Widerstandsfähigkeit, Verfügbarkeit, Wartbarkeit oder seiner Sicherheit im Sinne der Safety und der Security. Da die Methoden des Secure Coding das Ziel verfolgen, die Sicherheit der zu entwickelnden Software auf der Implementierungsebene durchzusetzen, ist es somit offensichtlich, dass die Gewährleistung einer hohen Qualität der beiden Sicherheitsattribute hierfür wichtig ist (Long, Mohindra, Seacord, Sutherland & Svoboda, 2011, Seite xxiii). Die einzelnen Qualitätsattribute eines Systems stehen jedoch nicht für sich alleine, sondern existieren in einer Beziehung zueinander und üben daher auch Einfluss aufeinander aus. Ein Beispiel für diese Wechselwirkungen kann an dem Qualitätsattribut der Lesbarkeit veranschaulicht werden: Eine schlechte Lesbarkeit des Quellcodes und eine damit eventuell verbundene Missverständlichkeit kann dazu führen, dass Verwundbarkeiten während der ersten Programmierung oder während der darauffolgenden Wartung in die Software eingeführt werden, weil der Entwickler den Quellcode falsch interpretiert. Durch diesen Zusammenhang kann beispielsweise ein schlecht lesbarer Quellcode indirekt die beiden Qualitätsattribute der Sicherheit gefährden (Long et al., 2011, Seite xxiii). Avizienis, Laprie, Randell und Landwehr (2004) führen in einer Taxonomie die Primärattribute Verfügbarkeit (Availability), Zuverlässigkeit (Reliability), Sicherheit im Sinne der Safety, Vertraulichkeit (Confidentiality), Integrität (Integrity) und Wartbarkeit (Maintainability) auf, die sich auf die Attribute der Security und der Systemstabilität (Dependability) auswirken können, siehe Abbildung 2.3.

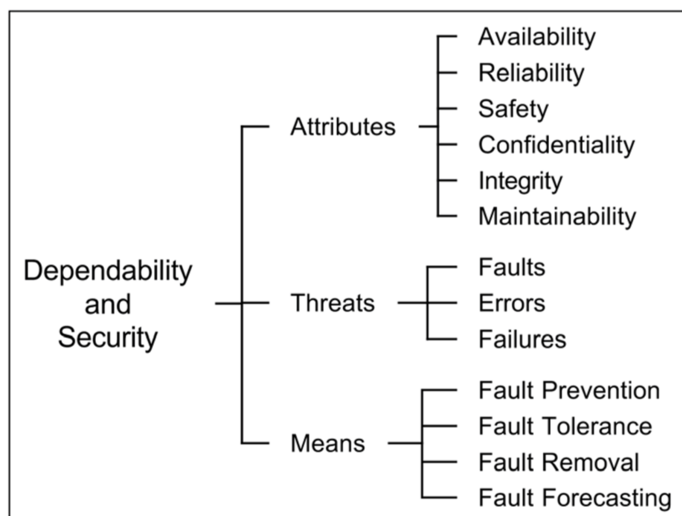

Abbildung 2.3: Taxonomie sicherheitsbezogener Attribute. Abbildung entnommen aus Avizienis, Laprie, Randell und Landwehr (2004, Seite 4).

D. G. Firesmith (2005) führt zudem die Attribute der Sicherheit im Sinne der Security, Überlebensfähigkeit (Survivability), Robustheit (Robustness) und Defensivfähigkeit (Defensibility) als Unterpunkte der Systemstabilität (Dependability) auf. Im Rahmen der Kategorie „Security-Significant Requirements“ seiner Taxonomie beschreibt Firesmith zusätzlich, dass die Relevanz der einzelnen Attribute, welche sich auf die Sicherheit des Systems auswirken können, gegeneinander abgewogen werden muss, um das durch sie entstehende Sicherheitsrisiko antizipieren zu können. Die Norm ISO/IEC 25000[3] stellt ebenfalls eine Zuordnung einzelner Sicherheitsattribute

[3] https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en, aufgerufen am 27.10.2019

auf, bei der auffällt, dass etwa die Attribute Zuverlässigkeit und Wartbarkeit nicht in einem Zusammenhang mit der Sicherheit der Software aufgelistet werden, obwohl auch diese sich wie zuvor beschrieben auf die Sicherheit auswirken können.

Somit ist festzuhalten, dass die exakten Zusammenhänge zwischen den einzelnen Qualitätsattributen zwar nicht einheitlich definiert und verwendet werden, sie sich jedoch eindeutig gegenseitig beeinflussen und daher auch auf die Sicherheit einer Software auswirken können, wenn dies zunächst nicht offensichtlich ist. Als Konsequenz dieser Wechselwirkungen ergibt sich, dass das Secure Coding sich nicht ausschließlich mit offensichtlichen Verwundbarkeiten befassen kann, sondern auch mit Schwachstellen auseinandersetzen muss, aus denen sich potenzielle Verwundbarkeiten über Umwege ergeben können. Typische Beispiele mit denen sich die Thematik des Secure Coding befasst sind daher etwa die Vermeidung von Buffer Overflows oder die Wahl adäquater kryptographischer Verfahren, aber auch Vorgehensweisen für eine bessere Lesbarkeit, Verständlichkeit oder Wartbarkeit des Quellcodes.

2.4 Relevanz des Secure Coding

Im Allgemeinen ist das Secure Coding bereits durch seinen Bezug zur Informationssicherheit ein wichtiges Konzept, um ihre drei zentralen Schutzziele, Vertraulichkeit, Integrität und Verfügbarkeit, zu gewährleisten. Um konkreter zu verdeutlichen, warum Secure Coding ein wichtiger Bestandteil der IT-Sicherheit ist, muss zunächst veranschaulicht werden, wie weit verbreitet die Verwendung von Software ist: Unter anderem Computer, Tablets, Smartphones, Waschmaschinen, GPS-Navigationssysteme, Banken, Krankenhäuser und Stromnetze verwenden Software zu unterschiedlichsten Zwecken und sind häufig auch an das Internet angebunden, womit sich diese Systeme zusätzlich angreifbar machen (Pfleeger et al., 2015). Neuere Forschungsgebiete wie das Internet of Things oder selbstfahrende Kraftfahrzeuge zeigen, dass sich auch weiterhin zusätzliche Anwendungsgebiete erschließen, in denen die Softwaresicherheit ein wichtiger Bestandteil ist, während die Anzahl erfasster Verwundbarkeiten auch ganz allgemein stetig zunimmt, siehe Abbildung 2.4.

Ein konkretes Beispiel für die Konsequenzen, die aus unzureichend sicherer Programmierung entstehen können, liefert der als Heartbleed-Bug[4] bekannte Fehler in der Software OpenSSL. Aufgrund eines Programmierfehlers, welcher Angreifern ohne großen Aufwand einen Buffer Overread ermöglichte, konnten Teile des Arbeitsspeichers kompromittierter Systeme ausgelesen und so unter anderem Zugriff auf die Zugangsdaten von Nutzern oder auch private Schlüssel von Serverzertifikaten erlangt werden, wodurch wiederum die Entschlüsselung zuvor aufgezeichneten Datenverkehrs ermöglicht wurde.

Der für den Heartbleed-Bug verantwortliche Implementierungsfehler war die fehlende Validierung einer Eingabevariable, welche die Länge der durch die Software aus dem Speicher auszulesenden und zurückzugebenden Daten festlegt. Dieser vergleichsweise triviale Fehler hatte weitreichende Folgen. Da es sich bei OpenSSL um die populärste Open Source Kryptographie Bibliothek und Transport Layer Security (TLS) Implementierung handelt, war auch eine Vielzahl großer Internetseiten und Online-Dienstleister wie beispielsweise Facebook, Google und Twitter betroffen (Michels, 2014). Da durch den Heartbleed-Bug auch Zugangsdaten kompromittiert

[4] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, aufgerufen am 10.11.2019

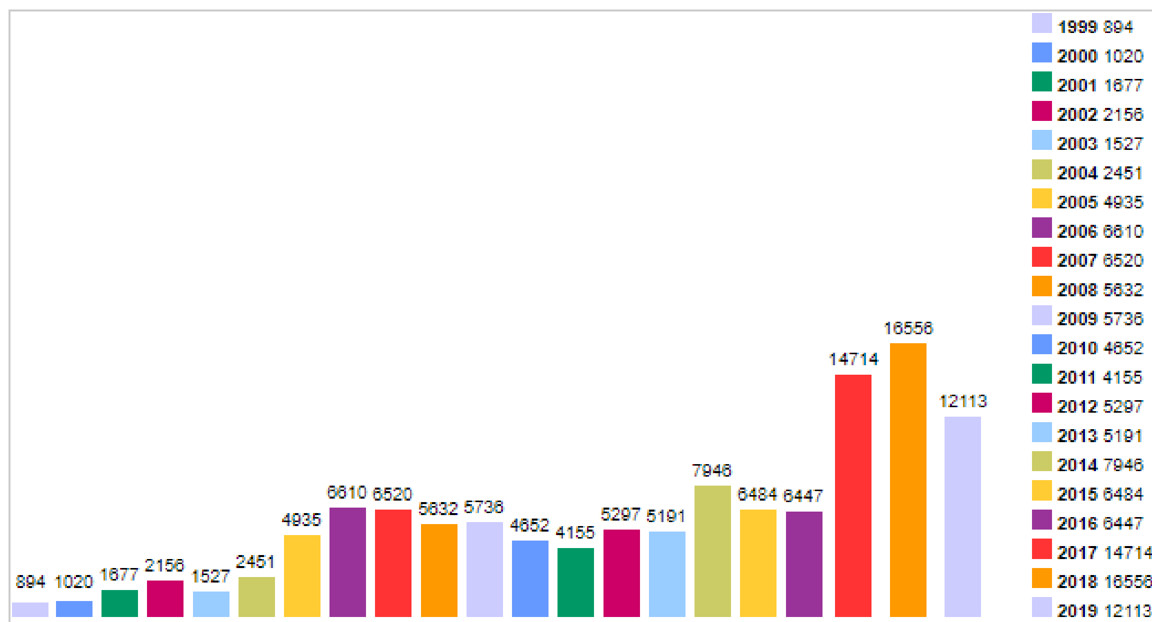

Abbildung 2.4: Die Anzahl der erfassten CVE-Einträge im Verlauf der letzten 20 Jahre[5]

werden konnten, wird an diesem Beispiel weiterhin deutlich, dass einfache Programmierfehler auch als Einstiegspunkt für weitreichendere Folgeangriffe verwendet werden können (Pfleeger et al., 2015, Seite 163). Der Sicherheitsexperte Bruce Schneier bewertete den Heartbleed-Bug dementsprechend wie folgt:

> „‚Catastrophic‘ is the right word. On the scale of 1 to 10, this is an 11.“[6]

Zusätzlich wird an dem Heartbleed-Bug die Bedeutsamkeit der proaktiven Natur des Secure Coding sichtbar, da die Verwundbarkeit nach ihrem Bekanntwerden zwar schnell behoben wurde, zuvor aber etwa 27 Monate Bestand hatte und die fehlerhafte Software auf den Systemen oft erst viel später aktualisiert wird. Die Suchmaschine Shodan erfasst etwa im Oktober 2019, über fünf Jahre nach Bekanntwerden der Verwundbarkeit, noch knapp 87.000 mit dem Heartbleed-Bug infizierte Geräte[7].

Das Beheben von Verwundbarkeiten im Nachhinein gilt zudem ganz allgemein als deutlich aufwändiger und teurer. Neben den monetären Kosten, etwa für die Koordinierung, das Finden und das Entfernen der Verwundbarkeit sowie die darauffolgenden Tests, kommt es auch zu Verlusten von Reputation und Zeit, die stattdessen in das Schreiben neuen Programmcodes investiert werden könnte (Howard & LeBlanc, 2003, Seite 10; Tassey, 2002, Seite 1-12).

Der Heartbleed-Bug ist somit ein deutliches Beispiel für die Relevanz des Secure Coding und seines proaktiven Ansatzes zur Vermeidung von Verwundbarkeiten und veranschaulicht dabei konkret, wie die sichere Programmierung einen zentralen Schritt zur Einhaltung der Schutzziele der Informationssicherheit darstellt.

[5] Abbildung entnommen von https://www.cvedetails.com/browse-by-date.php, aufgerufen am 14.10.2019
[6] https://www.schneier.com/blog/archives/2014/04/heartbleed.html, aufgerufen am 05.10.2019
[7] https://www.shodan.io/report/3uHHEtmX, aufgerufen am 15.10.2019

2.5 Methoden zur Gewährleistung sicheren Codes

Um die Sicherheit von Software im Allgemeinen und von Quellcode im Speziellen zu gewährleisten, existiert eine Vielzahl von Methoden auf unterschiedlichen Abstraktionsebenen und mit verschiedenen Zielen. Auf einer hohen Abstraktionsebene existieren verschiedene Methoden wie das von Microsoft veröffentlichte Konzept des „Security Development Lifecycle“[8] und verschiedene Frameworks, etwa das der Business Software Alliance (BSA) oder das Open Web Application Security Project (OWASP) Software Assurance Maturity Model (SAMM), welche sich über den gesamten Lebenszyklus der Softwareentwicklung erstrecken und dabei helfen, Strategien für die Gewährleistung der Softwaresicherheit durchzusetzen (BSA, 2019; OWASP, 2017). McGraw beschreibt in diesem Zusammenhang eine Reihe an *Software Security Touchpoints*, wie beispielsweise *Code Reviews* und *Architectural Risk Analysis*, als grundlegende Methoden, die in den unterschiedlichen Phasen des Entwicklungszyklus anzuwenden sind und eine von drei elementaren Säulen für die Gewährleistung der Softwaresicherheit darstellen (McGraw, 2006, Seite 101). Ein konkreter Faktor vor der eigentlichen Entwicklung der Software ist auch die Aufstellung von *Anforderungen* an die Sicherheit, um festzulegen welche Assets gegen welche Bedrohungen und in welchem Umfang zu schützen sind (Fabian, Gürses, Heisel, Santen & Schmidt, 2009, Seite 7). Bei dem OWASP Application Security Verification Standard handelt es sich dabei um eine Methode zur Unterstützung bei der Aufstellung adäquater Anforderungen an die Sicherheit (OWASP, 2019). Zusätzlich ist auch die Definition von *Security Policies* relevant, in denen technische und organisatorische Regeln, Verantwortlichkeiten und Maßnahmen zur Gewährleistung der Schutzziele festgehalten werden (Fabian et al., 2009, Seite 8). Ideen wie die frühen *Entwurfsprinzipien* nach Saltzer und Schroeder (1975) oder die aktuelleren *Entwurfsmuster* des SEI nach Dougherty, Sayre, Seacord, Svoboda und Togashi (2009) sind zudem relevant, um die Softwaresicherheit auf der Ebene der Architektur und des Entwurfs zu gewährleisten.

Auf einer etwas weniger abstrakten Ebene kann sich auch die Wahl der Programmiersprache auf die Sicherheit der damit zu entwickelnden Software auswirken. Java gilt beispielsweise im direkten Vergleich mit C oder C++ als eine relativ sichere Programmiersprache. Abschnitt 2.6 beschreibt die verschiedenen Komponenten und Funktionalitäten, die Java zu einer eher sicheren Sprache machen. Auch kann sich die Verwendung einer Untermenge einer Programmiersprache anbieten, welche die Entwicklung sicherer Software vereinfacht. Die Sprache Joe-E[9] stellt beispielsweise eine Untermenge von Java dar, die das Principle of Least Privilege durchsetzt, indem sichergestellt wird, dass jedes Objekt nur die für es absolut notwendigen Berechtigungen erhält und zudem Funktionalitäten entfernt, welche die Verkapselung übergehen können, um so das Verfassen sicheren Programmcodes zu fördern (Mettler & Wagner, 2009, Seite 1).

Die Kenntnisse des Entwicklers über mögliche Verwundbarkeiten und Methoden, um diese in der gewählten Programmiersprache zu vermeiden, wirkt sich ebenfalls direkt auf die Sicherheit des verfassten Codes aus und können beispielsweise durch spezialisierte Kurse wie die *SysAdmin, Networking and Security (SANS) Cyber Security Courses*[10] gefördert werden.

[8] https://www.microsoft.com/en-us/securityengineering/sdl/practices, aufgerufen am 16.12.2019
[9] https://code.google.com/archive/p/joe-e/, aufgerufen am 29.10.2019
[10] https://www.sans.org/courses/, aufgerufen am 16.12.2019

Je nach Wahl der Programmiersprache ist auch die korrekte Verwendung der durch sie eventuell bereitgestellten Sicherheitsfunktionalitäten wichtig, um die Sicherheit des Programmcodes zu gewährleisten. Die Implementierung der *Security Policy* in Java ist beispielsweise eine solche Sicherheitsfunktionalität, deren adäquate Nutzung wichtig ist, um die Softwaresicherheit zu optimieren. Abschnitt 2.6 beschreibt diese Funktionalitäten in Java und ihren jeweiligen Sicherheitsbezug detaillierter.

Konkret in Bezug auf die implementierungsbezogenen Phasen der Softwareentwicklung und somit auf einer sehr niedrigen Abstraktionsstufe gibt es verschiedene Methoden und Prinzipien, die zu beachten sind, um die Sicherheit des verfassten Programmcodes im Sinne des *„Building Security In“*-Prinzips zu gewährleisten. Die Verwendung von Frameworks und Application Programming Interfaces (APIs), sowohl für allgemeine als auch für sicherheitsspezifische Zwecke, statt eigener Implementierungen kann beispielsweise durch die damit verbundene Entlastung des Programmierers die Entstehung weiterer Verwundbarkeiten vermeiden. Namhafte sicherheitsbezogene Veröffentlichungen sind etwa das Spring Security Framework[11] oder die durch die Bouncy Castle Crypto APIs[12] bereitgestellten Kryptographiealgorithmen.

Eine weit verbreitete und für diese Arbeit zentrale Methode für die Vermeidung von Schwachstellen und Verwundbarkeiten während der Implementierung stellen die unterschiedlich gearteten *Best Practices* zur sicheren Programmierung dar. Diese Best Practices beschreiben verschiedene Fehlertypen sowie Vorgehensweisen, die während der Implementierung anzuwenden sind, um die jeweiligen Fehler zu vermeiden, wie etwa die korrekte Validierung externer Eingaben oder das Überschreiben sensitiver Informationen, damit diese nicht im Nachhinein durch unautorisierte Instanzen ausgelesen werden können. Teils werden solche Best Practices technologieagnostisch definiert, wie beispielsweise die Anleitungen des ISO/IEC TR 24772 (2013), die Implementierungsbezogenen Muster des SEI nach Dougherty et al. (2009), dementgegengesetzte Antimuster, welche zu vermeiden sind, oder verschiedene Secure Coding Praktiken, wie sie in dem OWASP Secure Coding Practices Quick Reference Guide oder in den durch das OWASP bereitgestellten Cheat Sheets[13] vorzufinden sind (OWASP, 2010).

Auf der anderen Seite existieren technologiespezifische Best Practices, die dementsprechend konkreter definiert sind: Bekanntere Beispiele sind die Motor Industry Software Reliability Association (MISRA) C und MISRA C++ *Richtlinien*, die für Java definierten Secure Coding *Regeln* und *Empfehlungen* des CERT-Standards[14] oder die durch Oracle selbst herausgegebenen Richtlinien[15], welche sich ebenfalls mit der Vermeidung von Schwachstellen und Verwundbarkeiten in der Programmiersprache Java befassen. Abschnitt 3.1 befasst sich detaillierter mit den verschiedenen Arten von Secure Coding Richtlinien. Auch allgemeine Regeln für den Programmierstil, die keinen direkten Bezug zu etwaigen Verwundbarkeiten aufweisen wie *Code Conventions* oder *Style Guides* können dabei helfen, sicheren Programmcode zu verfassen, da sie bei der Gewährleistung besser verständlichen und wartbaren Quellcodes helfen und sich

[11] https://spring.io/projects/spring-security, aufgerufen am 29.10.2019
[12] https://www.bouncycastle.org/, aufgerufen am 29.10.2019
[13] https://cheatsheetseries.owasp.org/, aufgerufen am 29.10.2019
[14] https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java, aufgerufen am 12.11.2019
[15] https://www.oracle.com/technetwork/java/seccodeguide-139067.html, aufgerufen am 29.10.2019

diese Qualitätsattribute wie in Abschnitt 2.3 beschrieben auch auf die Sicherheit der Software auswirken können.

Um Verwundbarkeiten in bereits geschriebenem Quellcode aufzufinden können Vorgehensweisen für *Code-Analysen* wie *Code Reviews* oder *Code Walkthroughs* angewendet werden. Teil- oder vollautomatisierte Werkzeuge zur statischen Analyse des Codes können hierbei sehr hilfreich sein, um typische Schwachstellen und Verwundbarkeiten aufzudecken, da ihre Verwendung die Code-Analyse deutlich effizienter gestaltet. Kapitel 4 befasst sich genauer mit Werkzeugen zur statischen Code-Analyse. Dynamische Testmethoden wie *Penetrationstests*, *Fuzzing* oder *Unit Tests* können ebenfalls bei der Identifizierung von Verwundbarkeiten helfen und auch durch automatisierte Werkzeuge unterstützt werden.

2.6 Sicherheit in Java

Da diese Arbeit sich im weiteren Verlauf mit der sicheren Programmierung in Java befasst, wird an dieser Stelle zunächst die grundlegende Funktionalität der Sicherheitsaspekte der Plattform und Programmiersprache Java erläutert, um zu veranschaulichen, inwiefern Java die sichere Entwicklung und Ausführung von Software ermöglicht und fördert. Diese Aspekte werden hier zudem näher im Detail beleuchtet, um darzustellen, inwiefern Sicherheitsmechanismen vorhanden sind, die bestimmte Verwundbarkeiten bereits von vornherein ausschließen und wie Verwundbarkeiten dennoch in Java-Programmen entstehen können. Die konkreten Sicherheitsfunktionalitäten und Konzepte von Java, wie beispielsweise Zugriffsmodifizierer, der Access Controller oder die Idee privilegierten Programmcodes werden an dieser Stelle auch genauer erklärt, da diese durch den Entwickler korrekt verwendet werden müssen, um die Entstehung von Verwundbarkeiten auf der Implementierungsebene auszuschließen. Anzumerken ist hier, dass einige dieser Sicherheitsfunktionalitäten von Java wie der Security Manager für die Entwicklung von Desktop-Programmen zwar optional sind, jedoch verwendet werden sollten, um mögliche negative Auswirkung nicht vertrauenswürdiger Akteure oder Programme einzugrenzen (Schönefeld, 2011, Seite 28).

Die in diesem Abschnitt beschriebenen Sicherheitsaspekte von Java beziehen sich auf die aktuelle Version Java Standard Edition (SE) 13 und basieren auf der offiziellen Dokumentation[16,17]. Die wesentlichen Prinzipien und Komponenten des hier veranschaulichten Sicherheitsmodells wurden bereits in der 1998 veröffentlichten Version Java 2 etabliert und werden in ihrer grundlegenden Form bis heute so verwendet. Dies wird belegt durch den Vergleich der von Oaks (2001) beschriebenen Sicherheitsarchitektur und der Dokumentation der aktuellen Java-Version. Aus diesem Grund sind die hier benannten Sicherheitsaspekte für viele Versionen vor Java SE 13 gültig und es ist anzunehmen, dass sie auch für zukünftige Versionen gültig sein werden.

Im Vergleich mit Sprachen wie C oder C++ gilt Java allgemeinhin als eine sichere Programmiersprache (Schönefeld, 2011, Seite 11). Diese Sicherheit von Java setzt sich aus zwei wesentlichen Bestandteilen zusammen: Zunächst stellt Java eine Plattform bereit, auf der sichere Software entwickelt und abgesichert ausgeführt

[16]https://docs.oracle.com/en/java/javase/13/security/java-security-overview1.html, aufgerufen am 22.10.2019

[17]https://docs.oracle.com/en/java/javase/13/security/java-se-platform-security-architecture.html, aufgerufen am 23.10.2019

werden kann, welche in der Praxis durch die Sicherheitsarchitektur von Java umgesetzt wird. Weiterhin stellt Java verschiedene sicherheitsbezogene Algorithmen, Werkzeuge und Dienste bereit, welche die Entwicklung sicherer Software vereinfachen. Hierzu zählen beispielsweise die sicherheitsbezogenen APIs, die durch Java bereitgestellt werden. Diese APIs implementieren Kryptographiealgorithmen, Public-Key-Infrastruktur (PKI), Algorithmen zur sicheren Kommunikation und Authentifizierung sowie Zugriffskontrollen und ermöglichen dem Entwickler daher die einfache Integration komplexer Sicherheitsalgorithmen in die eigene Software, ohne dass diese von dem Entwickler selbst implementiert werden müssen. Da im weiteren Verlauf vor allem relevant ist, wie der Programmierer sicheren Programmcode auf der Basis von Java verfassen kann, befasst sich der weitere Text dieses Abschnitts genauer mit der Sicherheit der Plattform Java und nicht mit den bereitgestellten Sicherheitsalgorithmen, Werkzeugen oder Diensten.

Grundlegend für die Funktionsweise des Sicherheitsmodells von Java ist die Unterscheidung zwischen *vertrauenswürdigem (trusted)* und *nicht vertrauenswürdigem (untrusted) Code*. Als vertrauenswürdig kann jeglicher Code definiert werden, der durch bekannte und vertrauenswürdige Instanzen bereitgestellt wird und daher umfangreichere Berechtigungen erhält als nicht vertrauenswürdiger Code (Long et al., 2013, Seite 252). Als nicht vertrauenswürdig wird hingegen Code bezeichnet, dessen Herkunft unbekannt oder selbst nicht vertrauenswürdig ist und dessen Ausführung potenziell schädlich sein könnte, weswegen sein Zugriff auf die verschiedenen Systemressourcen eingeschränkt werden sollte (Long et al., 2013, Seite 253). Beide Arten von Code können eng miteinander zusammenarbeiten, beispielsweise kann als nicht vertrauenswürdig eingestufter Code eine API nutzen, welche von vertrauenswürdigem Code bereitgestellt wird (Long et al., 2013, Seite 44). Die Plattform Java selbst trifft in den Versionen seit Java 2 keine klar abgegrenzte Unterscheidung zwischen vertrauenswürdigem und nicht vertrauenswürdigem Programmcode mehr, da ihre moderne Sicherheitsarchitektur eine konfigurierbare und feingranulare Zugriffskontrolle statt einer binären Unterscheidung umsetzt.

Konkreter ermöglicht die Programmiersprache selbst das Schreiben sicheren Programmcodes durch mehrere Mechanismen, wie ihre *Typsicherheit* und die *fehlende Pointer-Arithmetik* oder verschiedene Funktionalitäten, welche die Sprache möglichst einfach in ihrer Anwendung machen, beispielsweise die *automatische Speicherverwaltung* und den eingebauten *Garbage Collector*, sowie die *automatische Überprüfung von Array- und Stringgrenzen* zur Laufzeit. Indem Java diese Funktionalitäten bereitstellt, muss der Entwickler sie nicht selbst umsetzen oder beachten und ist somit während der Implementierung weniger belastet als etwa bei der Verwendung von C oder C++. Diese Mechanismen und auch die damit verbundene geringere Belastung des Entwicklers reduzieren die Wahrscheinlichkeit, dass Verwundbarkeiten, insbesondere solche die die Speicherintegrität gefährden können, während der Implementierung entstehen.

Die Plattform Java ermöglicht es dem Entwickler außerdem durch die Zugriffsmodifizierer `private`, `protected`, `package-private` und `public` den Zugriff auf die Details seiner Implementierung zu beschränken. Diese Zugriffsmodifizierer sind für die Sicherheit relevant, da durch ihre korrekte Anwendung nicht vertrauenswürdiger und potenziell gefährlicher Code möglichst wenig Zugriff auf die Implementierung des vertrauenswürdigen Codes erhält. Entsprechend sollte der Zugriff auf Klassen und ihre Implementierungsdetails durch die Verwendung von Zugriffsmodifizierern immer so weit wie möglich eingeschränkt werden, da nicht vertrauenswürdiger Code sonst

beispielsweise durch öffentliche Methoden Zugriff auf sicherheitssensitive, in privaten Variablen gespeicherte Informationen erlangen kann (Long et al., 2013, Seite 85). Somit stellen auch die Zugriffsmodifizierer eine Funktionalität von Java dar, welche für das Schreiben sicheren Programmcodes relevant ist.

Zentral für die Sicherheit von Programmcode ist der Schutz der Vertraulichkeit, Integrität und Verfügbarkeit der verschiedenen kritischen und sicherheitsrelevanten Systemressourcen wie etwa dem Speicher oder den lokalen Dateien. Um diese Ressourcen vor nicht autorisierten Zugriffen zu schützen, basiert das moderne Sicherheitsmodell von Java auf der Idee einer konfigurierbaren Sandbox, die sicherstellt, dass der Zugriff auf diese Ressourcen nur den dazu berechtigten Programmteilen gewährt wird (Oaks, 2001, Seite 14; Schönefeld, 2011, Seite 28).

Die praktische Realisierung des logischen Sandbox-Modells bildet die Sicherheitsarchitektur mit den zentralen Komponenten des Bytecode Verifiers, des Class Loaders, des Security Managers und des Access Controllers (Schönefeld, 2011, Seite 30). Diese Komponenten werden durch die Java Virtual Machine (VM) bereitgestellt, die es auch ermöglicht, den plattformunabhängigen Java-Code auf unterschiedlichster Hardware auszuführen. Diese Komponenten und weitere, in dem jeweiligen Zusammenhang relevante Elemente der Sicherheit in Java werden im Folgenden genauer beschrieben. Abbildung 2.5 veranschaulicht dazu die wesentlichen Zusammenhänge zwischen den einzelnen Komponenten der Sicherheitsarchitektur.

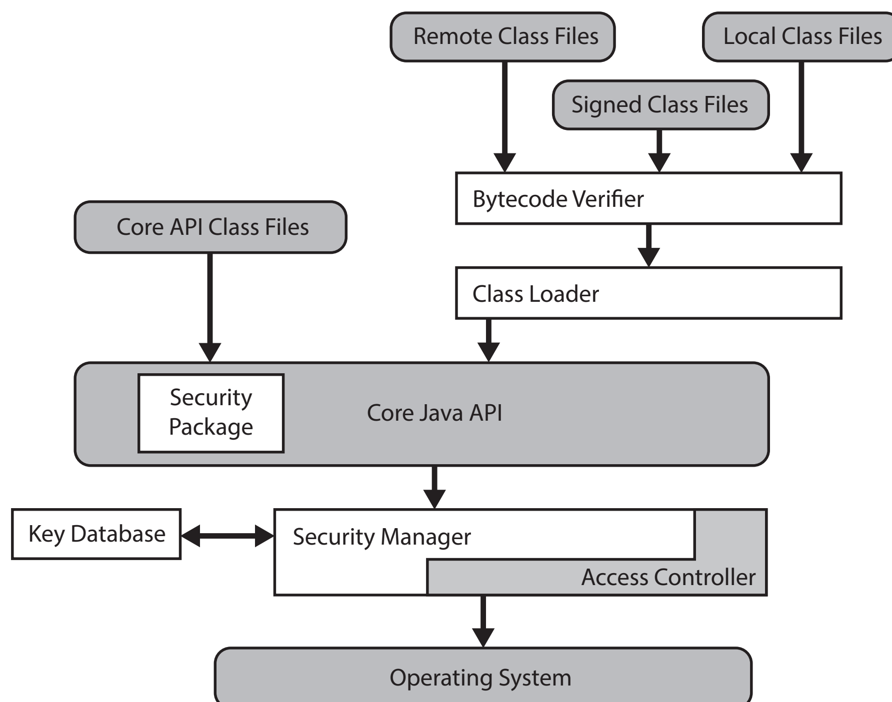

Abbildung 2.5: Verschiedene Komponenten der Java-Plattform. Die rechteckigen Komponenten bilden die Java-Sicherheitsarchitektur. Abbildung basierend auf Oaks (2001, Seite 17).

Vor seiner Ausführung wird der durch den *Compiler* generierte, plattformunabhängige Java-Bytecode durch die *Bytecode Verifier*-Komponente überprüft, um sicherzustellen dass nur legitimer Bytecode, der den Regeln der Sprachspezifikation

folgt und somit keine wesentlichen Verletzungen der Speicherverwaltung wie illegale Typumwandlungen enthält durch das Java Runtime Environment (JRE) ausgeführt wird. Damit ist auch der Bytecode-Verifier eine der sicherheitsrelevanten Komponenten, welche für die Gewährleistung der Speicherintegrität verantwortlich sind.

Der ebenfalls in der Java VM integrierte *Class Loader* ermöglicht die Installation von Softwarekomponenten zur Laufzeit. Der Class Loader ist eine zentrale Komponente für die Sicherheitsarchitektur von Java, da dieser basierend auf der zur Laufzeit installierten *Security Policy* den einzelnen kompilierten Klassendateien ihre jeweiligen Berechtigungen für den Zugriff auf die verschiedenen Systemressourcen zuordnet. Indem der Class Loader so verschiedene *Schutzdomänen*, also Zuordnungen von Code und seinen Berechtigungen bildet, setzt er ein grundlegendes und zentrales Systemsicherheitsprinzip praktisch für die Plattform Java um (Saltzer & Schroeder, 1975). Abbildung 2.6 veranschaulicht diesen Zusammenhang.

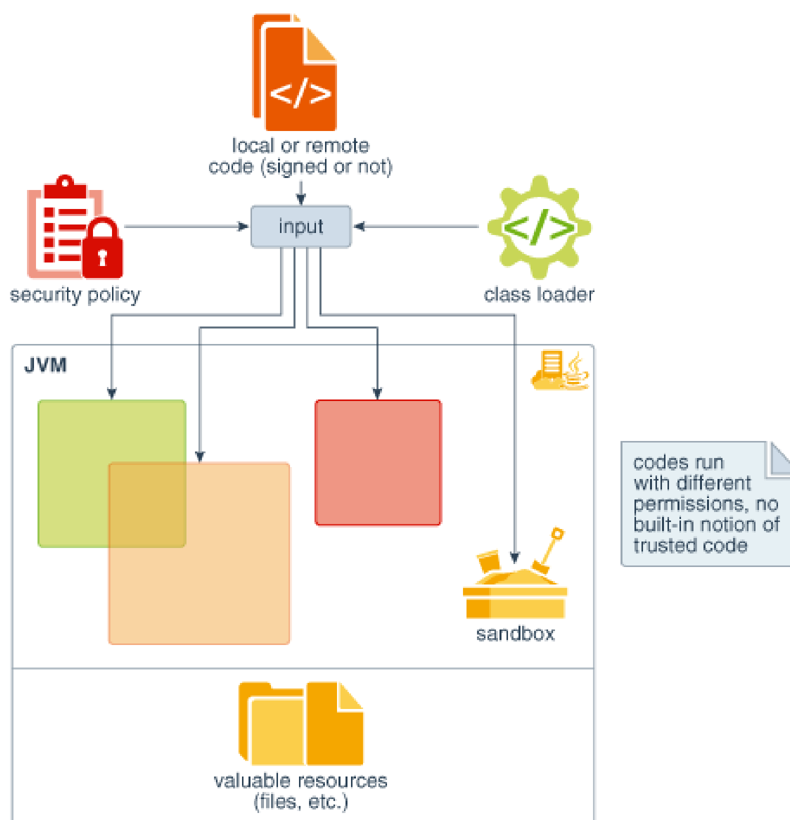

Abbildung 2.6: Veranschaulichung des Zusammenspiels zwischen Code, Class Loader und Security Policy in Java[18].

Die *Security Policy* kann durch Administratoren verwendet werden, um zu definieren, auf welche Systemressourcen der Programmcode aus verschiedenen Quellen zugreifen darf. Während der Laufzeit kann zu jedem Zeitpunkt nur ein solches Security Policy Objekt installiert sein. Standardmäßig werden diese Berechtigungen in Textdateien festgelegt, eigene Implementierungen wie etwa durch das Kontaktieren von Datenbanken oder anderen Diensten können jedoch auch umgesetzt werden. Ein

[18] Abbildung entnommen von https://docs.oracle.com/en/java/javase/13/security/java-se-platform-security-architecture.html, aufgerufen am 27.10.2019

exemplarischer Eintrag einer Security Policy ist in Abbildung 2.7 dargestellt. Dieser Eintrag in einer Security Policy Datei gewährt dem Code aus der lokalen Quelle `/home/sysadmin/` die Berechtigung, lesend auf die Datei `/tmp/abc` zuzugreifen.

```
grant codeBase "file:/home/sysadmin/" {
    permission java.io.FilePermission "/tmp/abc", "read";
};
```

Abbildung 2.7: Ein beispielhafter Eintrag einer Security Policy in Java[19]

Abbildung 2.5 veranschaulicht zudem die Klassen des Packages `java.security` sowie deren Erweiterungen, mit denen Java APIs bereitstellt, welche die Verwendung von Sicherheitsfunktionalitäten wie beispielsweise kryptographischen Algorithmen in dem Programm des Entwicklers ermöglichen, ohne dass diese von dem Programmierer selbst umgesetzt werden müssen.

Der *Security Manager* ist die primäre Instanz zur Gewährung oder Unterbindung von Zugriffen auf die Systemressourcen zur Laufzeit und arbeitet zu diesem Zweck eng mit dem *Access Controller* zusammen. Prinzipiell repräsentiert der Security Manager das Konzept einer zentralen Zugriffskontrolle, während der Access Controller den Algorithmus zur Kontrolle des Zugriffs auf Systemressourcen implementiert und spezielle Funktionalitäten wie die Methode `doPrivileged()` bereitstellt, die durch den Entwickler genutzt werden kann, um Teile des Programmcodes als *privilegiert* zu markieren. Der Access Controller erleichtert dem Programmierer das Schreiben sicheren Programmcodes insbesondere dadurch, dass ihm die Implementierung eines eigenen Algorithmus zur Prüfung von Zugriffskontrollen erspart wird und stattdessen eine zu diesem Zweck bewährte Methodik verwendet werden kann. Beide Komponenten sind somit für den Schutz sensitiver Systemressourcen wie beispielsweise lokaler Dateien verantwortlich.

Sobald während der Laufzeit ein Zugriff auf eine Systemressource angefordert wird, evaluiert der Access Controller basierend auf der installierten Security Policy, ob dieser Zugriff dem jeweiligen Code gewährt wird. Somit realisieren diese Komponenten im Zusammenspiel eine feingranulare Zugriffskontrolle, bei der genau festgelegt werden kann, welche Berechtigungen die verschiedenen Arten ausführbaren Programmcodes erhalten. Um festzustellen ob der Zugriff gewährt wird, gleicht der Access Controller bei einer Anfrage jedes Codeelement des Aufrufstapels mit der Security Policy ab und gewährt den Zugriff nur, wenn jedes der Codeelemente über die für diesen Zugriff notwendigen Berechtigungen verfügt. Eine Ausnahme in diesem Zusammenhang bildet das Konzept des *privilegierten Codes*. Durch den Aufruf der Methode `doPrivileged` kann vertrauenswürdiger Code anderem Programmcode temporär den Zugriff auf mehr Ressourcen gewähren als ihm standardmäßig zur Verfügung stehen würden. In diesem Fall muss nicht jedes Codeelement des Aufrufstapels über die nötigen Berechtigungen verfügen, da der vertrauenswürdige Code seine Berechtigungen temporär weitergibt. Der Security Manager ist damit eine zentrale Komponente für die Sicherheit von Java, die allerdings entweder durch den Anwender selbst über einen Kommandozeilenbefehl oder durch den Code für die Ausführung der

[19] Abbildung entnommen von https://docs.oracle.com/en/java/javase/13/security/permissions-jdk1.html, aufgerufen am 27.10.2019

Software installiert werden muss. Wird bei der Ausführung eines Java-Programmes kein Security Manager installiert, so werden jegliche sensitive Aktionen ohne Einschränkungen ausgeführt (Long et al., 2011, Seite 69, 71).

Bei der in Abbildung 2.5 dargestellten *Key Database* handelt es sich um eine Menge von Schlüsseln zum Erstellen oder Verifizieren digitaler Signaturen (Oaks, 2001, Seite 15, 16). Üblicherweise wird dies in Java durch eine .keystore-Datei umgesetzt.

Die Java VM ist für die Sicherheit der Programmiersprache Java im Vergleich zu etwa C und C++ zudem besonders relevant, da sie den Zugriff auf arbiträre Speicheradressen zur Laufzeit vermeiden kann, etwa indem sie sicherstellt, dass String- und Array-Grenzen eingehalten werden (Oaks, 2001, Seite 43). Diese Vermeidung des Zugriffs auf beliebige Speicheradressen zur Laufzeit ist ein wichtiger Bestandteil für die Durchsetzung der Sicherheit in High-Level Programmiersprachen wie Java, C# oder Perl, da speicherbasierte Verwundbarkeiten wie Buffer Overflows so typischerweise bereits von vornherein unterbunden werden (Howard & LeBlanc, 2003, Seite 128). Ähnlich verhält es sich mit Stack Overflow Fehlern. Wird ein solcher zur Laufzeit festgestellt, beendet die Java VM das Programm frühzeitig, wodurch die Gefahr durch diese Form von Verwundbarkeit auf die Beeinträchtigung der Verfügbarkeit begrenzt wird. Dies ist ein großer Vorteil, da diese Art von Verwundbarkeiten in anderen Sprachen, beispielsweise C und C++, ein enorm weit verbreitetes Sicherheitsrisiko darstellt. Knapp 70% aller Verwundbarkeiten in Microsoft-Produkten sind etwa Probleme der Speichersicherheit, siehe Abbildung 2.8.

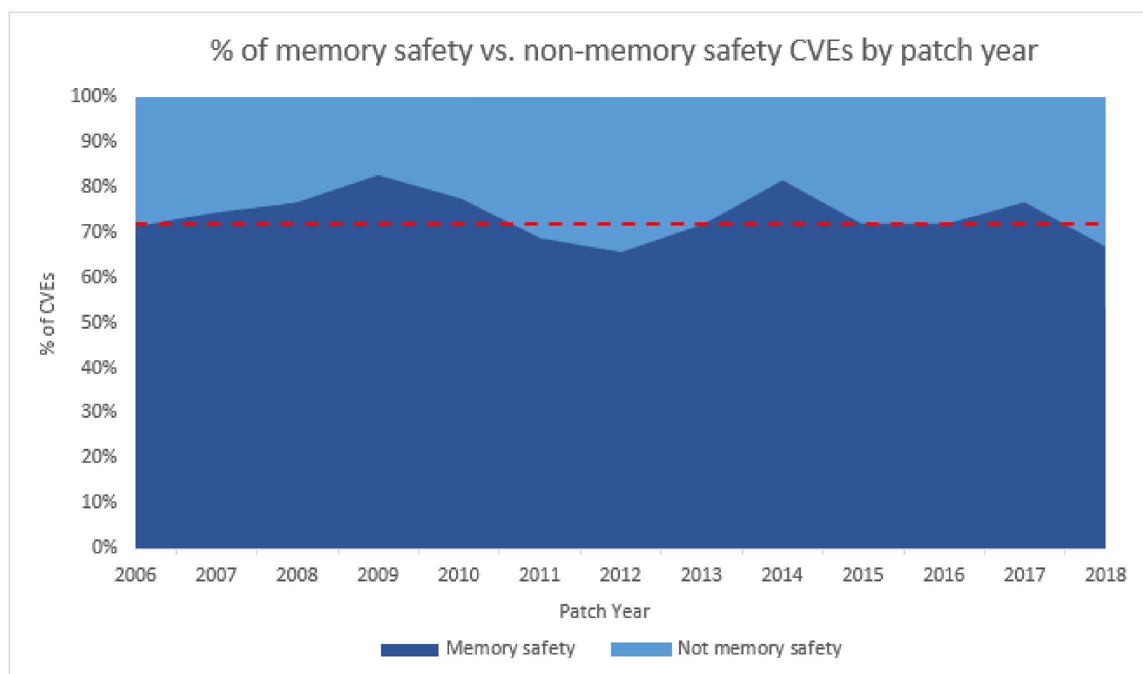

Abbildung 2.8: Prozentueller Anteil an speicherbasierten Verwundbarkeiten in Microsoft-Produkten von 2006 bis 2018[20]

Die hier beschriebene Sicherheitsarchitektur von Java unterstützt den Entwickler somit von Grund auf bei der Entwicklung sicherer Software. Dennoch gibt es diverse Arten von Schwachstellen und Verwundbarkeiten, die durch Fehler während der

[20]Abbildung basierend auf https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/, aufgerufen am 15.10.2019

Implementierung in eine Software eingeführt werden können. Ein Beispiel dafür wie die Zugriffsmodifizierer, eine der grundlegendsten Sicherheitsmaßnahmen von Java, durch einen Programmierfehler umgangen werden können bietet die Möglichkeit zur Serialisierung: Durch Serialisierung kann der Zustand einer Klasse, beispielsweise also auch repräsentiert durch den Wert einer als `private` deklarierten Variable, zur späteren Wiederherstellung über einen Bytestream gespeichert werden. Falls dieser Bytestream selbst nicht angemessen gesichert ist, kann die normalerweise unzugängliche Variable an dieser Stelle ausgelesen und modifiziert werden, womit die Vertraulichkeit und die Integrität des Systems gefährdet werden (Long, 2005, Seite 3). Die zuvor genannten Secure Coding Richtlinien befassen sich im Detail mit der Vermeidung solcher Verwundbarkeiten und Schwachstellen.

Weiterhin ist auch zu beachten, dass Verwundbarkeiten nicht nur durch die eigene fehlerhafte Programmierung, sondern auch durch die Verwendung fremden Programmcodes in eine Software eingeführt werden können (Brandt, Guo, Lewenstein, Dontcheva & Klemmer, 2009). Meng, Nagy, Yao, Zhuang und Argoty (2017) haben bei ihrer Untersuchung der Herausforderungen sicherer Programmierung in Java festgestellt, dass etwa die häufig verwendeten Spring Security APIs oft zu kompliziert in ihrer Verwendung und zu schlecht dokumentiert sind, was dazu führen kann, dass sie falsch verwendet werden. Ebenso können APIs die selbst Verwundbarkeiten beinhalten dazu führen, dass diese sich bei ihrer Verwendung in die eigene Software übertragen, indem etwa Frameworks Sicherheitsmechanismen bereitstellen, welche zur Absicherung der eigenen Software verwendet werden sollen, selbst aber keinen adäquaten Schutz gewährleisten (Long et al., 2013, Seite 13). Zusätzlich ist anzumerken, dass die Implementierung von Java selbst auch Verwundbarkeiten beinhalten kann, welche die Software des Entwicklers angreifbar machen können. Ein solches Problem wird beispielsweise durch CVE-2010-4476[21] beschrieben. Hierbei konnte ein Fehler in der Abbruchbedingung einer Schleife der Methode `parseDouble()` in der Kernsystemklasse `Double` ausgenutzt werden, um eine Endlosschleife zu erzwingen und so einen DoS-Angriff durchzuführen, wodurch das Schutzziel der Verfügbarkeit gefährdet wird (Schönefeld, 2011, Seite 121).

Festzuhalten ist somit, dass Java allgemeinhin zwar als sichere Programmiersprache gilt und dem Programmierer viele grundlegende Funktionalitäten und diverse Möglichkeiten für die Entwicklung sicherer Software bietet, allerdings müssen diese bereitgestellten Funktionalitäten auch angemessen verwendet und Logikfehler vermieden werden, um die Entstehung von Verwundbarkeiten zu vermeiden. Die bereits in Abschnitt 2.5 erwähnten Secure Coding Richtlinien können beispielsweise genutzt werden, um entsprechende Fehler zu identifizieren und zu beheben.

[21] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4476, aufgerufen am 25.10.2019

Kapitel 3

Secure Coding Richtlinien

Die Probleme der Softwaresicherheit teilen sich in etwa gleichmäßig auf die Ebene der Architektur und des Entwurfs sowie die Ebene der Implementierung auf, von denen aufgrund des Umfangs der Thematik im weiteren Text ausschließlich die Implementierungsebene näher betrachtet wird. Dieses Kapitel befasst sich näher mit den bereits in Abschnitt 2.5 erwähnten Secure Coding Richtlinien, die sich mit verschiedenen sicherheitsrelevanten Problemen auseinandersetzen und angewendet werden können, um Schwachstellen und Verwundbarkeiten in dem Quellcode einer Software zu identifizieren und zu beheben oder zu vermeiden.

3.1 Definition des Richtlinien-Begriffs

Um festzulegen welche Arten von sicherheitsrelevanten Programmierpraktiken auf der Implementierungsebene unter den Begriff der Secure Coding Richtlinien fallen, muss dieser zunächst definiert und von verwandten Begriffen abgegrenzt werden. Der Duden definiert den verwandten Begriff der Regel beispielsweise als eine „aus bestimmten Gesetzmäßigkeiten abgeleitete, aus Erfahrungen und Erkenntnissen gewonnene, in Übereinkunft festgelegte, für einen jeweiligen Bereich als verbindlich geltende Richtlinie . . . “[1]. Somit bilden Regeln dieser Definition nach eine verbindliche Unterart der Richtlinien. In der englischen Sprache definiert das Cambridge Dictionary den übersetzten Begriff „Rule“ wie Folgt: „a general standard that guides one’s actions“[2]. Das englische Wort für Richtlinie, „Guideline“, wird hier zudem folgendermaßen beschrieben: „a piece of information that suggests how something should be done“[3]. Diese Definitionen legen nahe, dass die Verbindlichkeit ein wesentliches Unterscheidungskriterium für Richtlinien und ihre Unterformen ist. Allgemeinhin können Richtlinien also verbindlich im Sinne einer Regel oder unverbindlich im Sinne einer Empfehlung kategorisiert werden.

Da die relevanten Definitionen dieser Begrifflichkeiten im Bezug auf die Softwaresicherheit insbesondere aus verschiedenen Standards hervorgehen, wird hier die Bedeutung des Begriffs der Richtlinie weiterhin im Hinblick auf seine Verwendung in entsprechenden Publikationen betrachtet. Die durch das CERT herausgegebenen

[1] https://www.duden.de/rechtschreibung/Regel, aufgerufen am 30.10.2019

[2] https://dictionary.cambridge.org/de/worterbuch/englisch-deutsch/rule, aufgerufen am 30.10.2019

[3] https://dictionary.cambridge.org/de/worterbuch/englisch/guideline, aufgerufen am 30.10.2019

Kodierungsstandards für die sichere Programmierung in Sprachen wie Java, C oder C++ definieren das Wort *Richtlinie (Guideline)* als Oberbegriff für *Regeln (Rules)* und *Empfehlungen (Recommendations)*. Diese Definitionen entsprechen damit der zuvor beschriebenen, rein sprachlichen Definition. Richtlinien werden hier als Regeln klassifiziert, wenn sie drei festgelegte Kriterien erfüllen[4]:

1. Eine Verletzung der Richtlinie resultiert in einem Defekt, der sich nachteilig auf die Sicherheit oder Zuverlässigkeit des Programms auswirken kann.

2. Die Richtlinie ist nicht von Annahmen über den Code oder Annotationen im Quellcode abhängig.

3. Die Konformität des Codes zu der Richtlinie kann durch automatisierte Analysen, formale Methoden oder manuelle Inspektionen festgestellt werden.

Im Gegensatz dazu werden Richtlinien als Empfehlungen kategorisiert, wenn ihre Anwendung zwar zu einer Verbesserung der Sicherheit und Zuverlässigkeit des Softwaresystems führt, jedoch mindestens eines der Kriterien für eine Klassifizierung als Regel nicht erfüllt wird, beispielsweise indem die Missachtung der Richtlinie nicht zwangsweise in einem Defekt im Programmcode resultiert. Zwei beispielhafte Richtlinien, an denen die Verwendung dieser Kriterien verdeutlicht wird, sind in Abschnitt 3.4 aufgelistet. Anders als Empfehlungen müssen die normativen Regeln somit eingehalten werden, damit ein Softwaresystem den Standard erfüllt, außer wenn eine für die jeweilige Regel fest definierte Ausnahme zutrifft (Seacord, 2006, Seite 2).

Sowohl die Regeln als auch die Empfehlungen der CERT-Standards verfolgen damit das Ziel, unsichere Programmierpraktiken, die in Schwachstellen und Verwundbarkeiten resultieren können zu vermeiden und unterscheiden sich auch nicht in ihrer Bedeutsamkeit für die Sicherheit einer Software oder der Schwere der Konsequenzen einer Missachtung der jeweiligen Richtlinie, sondern sind als gleich wichtig anzusehen (Long et al., 2013, Seite xiv). Thematisch behandeln sie ebenfalls größtenteils identische Felder, wie Fehler bei der Verwendung der Datentypen `character` und `String`[5,6] oder der Validierung von Eingaben[7,8]. Unterschiede im Hinblick auf die jeweils behandelten Themen ergeben sich nur auf Basis der Normativität, etwa indem Richtlinien die sich nur mit der Lesbarkeit des Quellcodes befassen nicht als normative Regeln angesehen werden können, da die schlechte Lesbarkeit, die sich durch die Missachtung einer solchen Richtlinie ergibt, nicht in einem prüfbaren Defekt resultiert.

Im Hinblick auf weitere Publikationen definiert die von der ISO veröffentlichte technische Spezifikation ISO/IEC TS 17961:2013[9] ähnlich wie der CERT-Standard

[4]https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487386, aufgerufen am 12.11.2019
[5]https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487607, aufgerufen am 12.11.2019
[6]https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487333, aufgerufen am 12.11.2019
[7]https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487865, aufgerufen am 12.11.2019
[8]https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487337, aufgerufen am 12.11.2019
[9]https://www.iso.org/obp/ui/#iso:std:61134:en, aufgerufen am 31.10.2019

einzuhaltende Regeln, die sich insbesondere dadurch auszeichnen, dass die Konformität zu jeder dieser Regeln anhand statischer Code-Analysen festgestellt werden kann.

Zwei der weltweit führenden[10] industriespezifischen Standards in diesem Bereich, die durch die Automotive Open System Architecture (AUTOSAR) und die MISRA definiert werden, befassen sich ebenfalls mit dieser Thematik. Im Vergleich zu der ISO/IEC TS 17961:2013 und den CERT-Standards definiert der AUTOSAR-Standard für C++ unterschiedlich priorisierte Regeln, von denen nur die als „erforderlich" eingestuften Regeln für die Erfüllung des Standards notwendig sind. Regeln die stattdessen in einer Hinweisfunktion aufgeführt werden entsprechen eher der Definition der Empfehlungen der CERT-Standards, da diese ebenfalls nach eigenem Ermessen anzuwenden sind (AUTOSAR, 2017, Seite 15).

Der MISRA-C-Standard bezeichnet die konformitätsrelevanten Elemente hingegen als „Richtlinien", die in unterschiedlichem Maße zu erfüllen sind (MISRA, 2016, Seite 10). Jede dieser Richtlinien gehört zunächst einer von drei Kategorien an: Verbindliche Richtlinien müssen immer eingehalten werden, während erforderliche und beratende Richtlinien unter bestimmten Bedingungen missachtet werden können. Diese Unterscheidung ist im Wesentlichen eine Weiterentwicklung der Abgrenzung zwischen den Regeln und Empfehlungen des CERT-Standards: „Earlier versions drew a simple distinction between those categorized as Required and those categorized as Advisory" (MISRA, 2016, Seite 10). Der MISRA-C-Standard trifft jedoch eine weitere Unterscheidung nach *Regeln*, welche objektiv prüfbare Richtlinien darstellen, und *Direktiven*, welche beispielsweise von subjektiven Einschätzungen abhängig sein können (MISRA, 2016, Seite 3). Beide dieser Arten von Richtlinien können dabei verbindlich, erforderlich oder beratender Natur sein. Um zu unterscheiden, ob eine Richtlinie angewendet werden muss, ist die Trennung von Regeln und Direktiven daher nicht relevant. Die subjektive Komponente der Direktiven ähnelt charakteristisch den Empfehlungen der CERT-Standards, welche ebenfalls eher subjektiv sind, da sie auch von der Intention des Programmierers abhängig sein können (Long et al., 2013, Seite xiv).

Das CERT grenzt die für jede Programmiersprache spezifischen Richtlinien zudem von *Mustern* ab, die abstraktere, weil allgemeinere und technologieagnostische Vorgehensweisen definieren (Dougherty et al., 2009, Seite 1). Weiterhin definiert das OWASP sogenannte Secure Coding *Praktiken*, die sich von Richtlinien auch dadurch unterscheiden, dass sie nicht nur während der Implementierungsphase anzuwenden sind, sondern über den gesamten Lebenszyklus der Softwareentwicklung hinweg (OWASP, 2010, Seite 3). Diese Praktiken sind genau wie die zuvor genannten Muster unabhängig von der verwendeten Programmiersprache und daher ähnlich abstrakt.

Im Hinblick darauf, ob die relevanten Begrifflichkeiten über verschiedene Standards hinweg einheitlich verwendet werden ergibt sich somit zunächst, dass eine Unterscheidung zwischen Methoden, die von der verwendeten Programmiersprache unabhängig sind, und solchen, die sprachspezifisch sind, getroffen wird. Zudem wird zwischen Richtlinien, die angewendet werden müssen und solchen, die angewendet werden sollten unterschieden. Die konkreten Begrifflichkeiten für diese Arten von

[10] https://www.synopsys.com/automotive/misra-autosar-standards.html, aufgerufen am 31.10.2019

Richtlinien werden über unterschiedliche Standards und Spezifikationen hinweg jedoch nicht einheitlich verwendet.

Da der Begriff der Richtlinie nicht einheitlich verwendet wird, muss seine Bedeutung für diese Arbeit an dieser Stelle definiert werden. Da im Folgenden ausschließlich Richtlinien für Java behandelt werden, wird die Definition der CERT-Publikationen verwendet, sodass Richtlinien in der Funktion eines Überbegriffs für normative, grundsätzlich einzuhaltende Regeln und nicht normative Empfehlungen verwendet werden, deren Sinn die gezielte Vermeidung von Schwachstellen und Verwundbarkeiten während der Implementierung ist. Die Secure Coding Richtlinien werden weiterhin im Folgenden auf der Basis ihres Abstraktionsgrades unterschieden: Richtlinien die unabhängig von der verwendeten Programmiersprache sind werden dementsprechend als *abstrakte Richtlinien* bezeichnet, während sprachspezifische Richtlinien wie etwa die der CERT-Standards unter den Begriff der *spezifischen Richtlinien* fallen. Beide Kategorien sind dabei nicht unabhängig voneinander, da es sich bei vielen spezifischen Richtlinien auch um eine Konkretisierung von abstrakten Richtlinien handelt. Diese Zusammenhänge werden zudem in Abbildung 3.1 veranschaulicht.

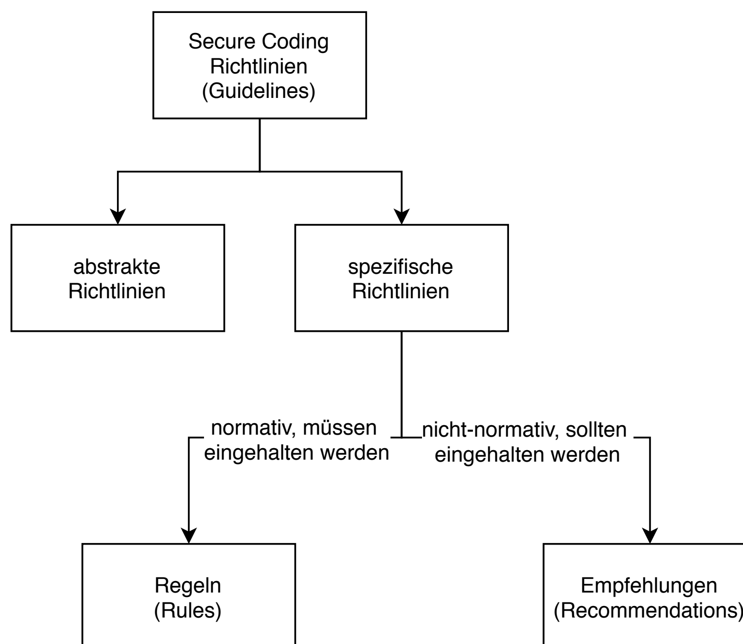

Abbildung 3.1: Unterscheidung verschiedener Formen von Secure Coding Richtlinien

3.2 Quellen für Secure Coding Richtlinien

McGraw und Felten stellten bereits 1999 zwölf Regeln für das Verfassen sicheren Quellcodes in Java vor[11]. Seitdem wurden verschiedene Secure Coding Richtlinien für Java in unterschiedlichen Publikationen veröffentlicht, Alexander, Bieman und Viega (2000), Bloch (2008), Chess und West (2007) sowie Gong, Ellison und Dageforde (2003) befassen sich beispielsweise mit unterschiedlichen Eigenschaften von

[11] http://www.securingjava.com/chapter-seven/chapter-seven-1.html, aufgerufen am 06.11.2019

Java, die zu Verwundbarkeiten in einem Softwaresystem führen können und erläutern Lösungsvorschläge, die als Secure Coding Richtlinien bezeichnet werden können. Oracle selbst stellt heute ebenfalls Secure Coding Richtlinien für Java bereit[12]. Lai (2008) beschreibt zudem ergänzende Feinheiten, die bei der Anwendung ausgewählter abstrakter Richtlinien in Java zu beachten sind. Da diese Quellen meist Java-spezifische Vorgehensweisen zur Vermeidung von Verwundbarkeiten beschreiben, fallen diese Praktiken unter die Kategorie der spezifischen Richtlinien.

Weiterhin beschreiben beispielsweise auch der Comprehensive, Lightweight Application Security Process (CLASP)[13], die Seven Pernicious Kingdoms nach McGraw (2006, Seite 274), die Common Weakness Enumeration (CWE)[14] sowie die OWASP Cheat Sheets[15] oder die zuvor genannten Secure Coding Praktiken des OWASP mögliche Ursachen für die Entstehung von Verwundbarkeiten und Ansätze, um diese zu mildern oder zu vermeiden (Secure Software, 2005). Diese Richtlinien sind dabei abstrakter Natur und benennen somit allgemeine Praktiken, die für verschiedenste Programmiersprachen wie meist auch Java anzuwenden sind.

Die genannten Veröffentlichungen beschreiben zwar eine Vielzahl unterschiedlicher und wichtiger Richtlinien, jedoch liefert keine der Quellen eine möglichst vollständige Liste an prüfbaren Richtlinien, die konkrete Programmierpraktiken vorschreiben, welche gleichmäßig bei der Entwicklung von Softwaresystemen auf der Implementierungsebene angewendet und für die Evaluierung des Softwareprojekts verwendet werden können. Diese Anforderungen werden jedoch an einen *Secure Coding Standard* gestellt (Seacord, 2006, Seite 1).

Ein solcher Kodierungsstandard existierte für Java lange Zeit nicht. Erst 2011 veröffentlichte die CERT Secure Coding Initiative, die bereits ähnliche Standards für die Sprachen C und C++ entwickelt hatte, eine erste Version des „SEI CERT Oracle Coding Standard for Java“ (Ware & Fox, 2008, Seite 1). Der CERT Secure Coding Standard für Java stellt somit aus verschiedenen Gründen einen guten Ansatzpunkt für die Betrachtung von Secure Coding Richtlinien im Rahmen dieser Arbeit dar: Durch seine Funktion als Standard beinhaltet diese Publikation viele Richtlinien, die auch in anderen Quellen beschrieben werden und stellt eine umfangreiche Liste an Ursachen für mögliche Verwundbarkeiten und entsprechenden Lösungsansätzen bereit. Zudem definiert der CERT-Standard spezifische Richtlinien, die auf die Eigenheiten von Java zugeschnitten sind und somit präzisere Hilfestellungen leisten können als vergleichbare abstrakte Richtlinien. Die Online-Version des Standards[16] wird stets aktualisiert und gewartet, wodurch die hier beschriebenen Richtlinien aktuell gehalten werden. Daher sind sie auch für neuere Versionen der Programmiersprache relevant und Fehler oder missverständliche Beschreibungen in den Richtlinien werden mit der Zeit herausgearbeitet. Weiterhin sind die hier beschriebenen Richtlinien durch ihren Fokus auf Java SE allgemein genug, um für die Entwicklung einer möglichst großen Vielfalt an unterschiedlicher Software hilfreich zu sein und dabei präzise genug formuliert,

[12]https://www.oracle.com/technetwork/java/seccodeguide-139067.html, aufgerufen am 06.11.2019

[13]https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf, aufgerufen am 06.11.2019

[14]http://cwe.mitre.org/, aufgerufen am 06.11.2019

[15]https://cheatsheetseries.owasp.org/, aufgerufen am 06.11.2019

[16]https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java, aufgerufen am 06.11.2019

um effektiv helfen zu können (Seacord, 2006, Seite 2). Aus diesen Gründen werden ausschließlich die in dem CERT-Standard beschriebenen Secure Coding Richtlinien im Folgenden betrachtet und durch die Verwendung ihrer jeweiligen Identifikatoren referenziert.

3.3 Relevanz von Secure Coding Richtlinien

Insbesondere bei spezifischen Secure Coding Richtlinien handelt es sich um ein sehr effektives Mittel zur Vermeidung von Schwachstellen und Verwundbarkeiten, da sie langjährige, konkrete Erfahrungen von anderen Programmierern zusammenfassen (Chess & West, 2007, Seite 3). Üblicherweise besteht eine Secure Coding Richtlinie aus zwei Komponenten: Eine Dokumentation des jeweiligen Fehlers kann verwendet werden, um den Fehler im Quellcode des Entwicklers zu identifizieren und es werden Lösungsansätze bereitgestellt, um den Fehler zu vermeiden oder zu beheben. Anders als etwa Penetrationstests und gemäß des Prinzips *„Building Security In“* bietet die Anwendung von Secure Coding Richtlinien damit eine Möglichkeit, um die Ursachen von Verwundbarkeiten zu vermeiden, statt ihre Symptome nachträglich aufzudecken. Die besonders wichtigen Richtlinien lassen sich dabei auf eine recht geringe Menge eingrenzen, da es sich bei den Fehlern, die zu schwerwiegenden Verwundbarkeiten führen meist um wenige, stets wieder auftretende Problemtypen handelt (Chess & West, 2007, Seite 20). Zur Veranschaulichung der Relevanz wird folgend eine Auswahl von Verwundbarkeiten in Java-basierter Software aufgeführt, welche sich durch die Anwendung von Secure Coding Richtlinien hätten vermeiden lassen.

- **CVE-2010-4476[17]:** Ein Fehler in der Methode `Double.parseDouble()` von Java konnte unter der Verwendung mit einem speziell zusammengestellten String-Argument durch die automatische Konvertierung des Strings in einen Wert vom Typ `double` für einen DoS-Angriff genutzt werden. Mit solchen Problemen befasst sich die Richtlinie *NUM54-J. Do not use denormalized numbers*[18] des CERT Kodierungsstandards für Java.
- **CVE-2015-2080[19]:** Eine Verwundbarkeit in Eclipse Jetty konnte ausgenutzt werden, um eine Fehlermeldung zu provozieren, die sensitive Informationen, wie z.B. Cookies aus vorherigen Requests enthält. Die Richtlinie *ERR01-J. Do not allow exceptions to expose sensitive information*[20] behandelt diese Thematik.
- **CVE-2006-6969[21]:** Eclipse Jetty berechnete zeitweise vorhersehbare Sitzungsidentifikatoren unter der Verwendung von `java.util.random`, anhand derer die Sitzungsauthentifizierung umgangen und Cross-Site-Request-Forgery (CSRF) Angriffe durchgeführt werden konnten. Mit der Generierung von Zufallszahlen befasst sich *MSC02-J. Generate strong random numbers*[22].

[17] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4476, aufgerufen am 16.12.2019
[18] https://wiki.sei.cmu.edu/confluence/display/java/NUM54-J.+Do+not+use+denormalized+numbers, aufgerufen am 16.12.2019
[19] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2080, aufgerufen am 16.12.2019
[20] https://wiki.sei.cmu.edu/confluence/display/java/ERR01-J.+Do+not+allow+exceptions+to+expose+sensitive+information, aufgerufen am 16.12.2019
[21] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6969, aufgerufen am 16.12.2019
[22] https://wiki.sei.cmu.edu/confluence/display/java/MSC02-J.+Generate+strong+random+numbers, aufgerufen am 16.12.2019

- **CVE-2016-4999[23]:** Die Software Dashbuilder besaß zeitweise eine Verwundbarkeit, welche SQL-Injektionsangriffe ermöglichte. Die Richtlinie *IDS00-J. Prevent SQL injection*[24] behandelt Vorgehensweisen, um diese Form von Angriffen zu verhindern.

3.4 Beispiele für Secure Coding Richtlinien

An den folgenden Beispielen wird verdeutlicht auf welcher Abstraktionsebene sich die Richtlinien des CERT-Standards mit der Vermeidung von Schwachstellen und Verwundbarkeiten befassen, wie sich Regeln in der Praxis von Empfehlungen unterscheiden und wie Richtlinien, die keine offensichtliche Relevanz für die Sicherheit eines Softwaresystems haben, diese Sicherheit dennoch beeinflussen können.

Die Regel *IDS00-J. Prevent SQL injection*[25] des CERT-Standards beschreibt, dass nicht vertrauenswürdige Elemente einer SQL-Abfrage wie z.B. Eingaben des Anwenders vor ihrer Verwendung bereinigt werden müssen, um zu vermeiden, dass im Rahmen eines Injektionsangriffs die Vertraulichkeit, Integrität oder Verfügbarkeit gespeicherter Informationen verletzt wird. Die Richtlinie führt ein Szenario an, bei dem ein Benutzer authentifiziert wird, falls die in Listing 3.1 aufgeführte SQL-Abfrage ein Ergebnis zurückliefert. Dieser Quellcode ist anfällig für eine SQL-Injektion, da für die Variable `pwd` der String „`'OR '1'= '1`“ angegeben werden könnte. Die Abfrage würde in diesem Fall immer ein Ergebnis zurückliefern, da `'1'= '1'` immer wahr ist. Da der CERT-Standard spezifische Richtlinien definiert, wird hier zur Unterbindung der Schwachstelle konkret die Verwendung der Funktion `prepareStatement()` empfohlen und es werden die Feinheiten ihrer Anwendung beschrieben. An diesem Beispiel ist klar ersichtlich, wie die Sicherheit des Softwaresystems durch den beschriebenen Fehler auf direktem Wege beeinträchtigt werden kann.

```
1 String sqlString = "SELECT * FROM db_user WHERE username = '"
2                    + name +  "' AND password = '" + pwd + "'";
3 Statement stmt = connection.createStatement();
4 ResultSet rs = stmt.executeQuery(sqlString);
```

Listing 3.1: Beispiel für Quellcode der für eine SQL-Injektion anfällig ist

Diese Richtlinie verdeutlicht auch die zuvor beschriebenen Kriterien, welche der CERT-Standard für die Unterscheidung von Regeln und Empfehlungen anwendet: Werden nicht vertrauenswürdige Komponenten einer SQL-Abfrage nicht vor ihrer Verwendung bereinigt, so liegt hier offensichtlich ein Defekt vor, der sich auf die Sicherheit des Systems auswirkt. Es gibt bei der Interpretation dieser Richtlinie keinen Spielraum, da ein Versäumnis bei der Bereinigung hier immer eine Gefahr darstellt. Abschließend kann auch die Konformität von einem Programmcode zu dieser Richtlinie problemlos durch verschiedene Analysemethoden, wie etwa eine manuelle Inspektion des Quellcodes festgestellt werden, da der Defekt so schnell ersichtlich wird.

[23] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4999, aufgerufen am 16.12.2019
[24] https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection, aufgerufen am 16.12.2019
[25] https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection, aufgerufen am 05.11.2019

Ein Gegenbeispiel ist die Empfehlung *DCL50-J. Use visually distinct identifiers*[26], die beschreibt, dass Zeichen, die nahezu identisch aussehen können, etwa die Ziffer „1“ und ein kleingeschriebenes „L“, nicht als einziges Unterscheidungsmerkmal von zwei verschiedenen Variablen verwendet werden sollen, da in diesem Fall beide Variablen leicht miteinander verwechselt werden könnten. Hier wird zunächst deutlich, dass nicht jede Richtlinie einen direkten und offensichtlichen Bezug zur Sicherheit der Software hat, stattdessen führt der hier beschrieben Fall zunächst nur zu einer verschlechterten Wartbarkeit. Wie in Abschnitt 2.3 beschrieben kann eine solche schlechte Wartbarkeit anschließend jedoch auch in der Einführung von Verwundbarkeiten in die Software resultieren. In diesem Beispiel könnte etwa eine Variable, die sensitive Informationen speichert, fälschlicherweise nicht adäquat verwendet werden, weil sie mit einer anderen Variable verwechselt wurde, die keinerlei sicherheitsrelevante Daten enthält. Weil die Verwendung leicht verwechselbarer Zeichen zudem nicht in einem formal prüfbaren Defekt resultiert, wird hier weiterhin erkenntlich, dass diese Richtlinie nicht als Regel, sondern als Empfehlung klassifiziert werden muss.

3.5 Gründe für die Missachtung von Secure Coding Richtlinien

Wie bereits in Kapitel 2 ersichtlich geworden ist, ist die Sicherheit des Quellcodes ein wichtiges Element, um die Softwaresicherheit im Allgemeinen zu gewährleisten. Mit Secure Coding Richtlinien existieren wie hier beschrieben konkrete Anleitungen, um verschiedenste Schwachstellen und Verwundbarkeiten zu identifizieren und zu vermeiden. Aus unterschiedlichen Gründen werden diese Richtlinien oft jedoch nicht angewendet. Die Studie von Xie, Lipford und Chu (2011, Seite 164) ergab, dass Programmierer die Durchsetzung der Softwaresicherheit oft nicht als ihre Verantwortlichkeit ansehen und sich stattdessen auf andere Personen, Prozesse oder Technologien verlassen. Ein weiterer Grund ist, dass Entwickler oft nicht ausreichend im Hinblick auf die Thematik der Softwaresicherheit ausgebildet sind und somit mögliche Fehlerquellen nicht erkennen (Graff & van Wyk, 2003, Seite 20). Ebenso kommt es häufig vor, dass Entwickler nicht über die Existenz von Secure Coding Richtlinien oder Standards informiert sind oder dass sie die einzelnen Richtlinien aufgrund mangelnder Fachkenntnisse nicht korrekt einschätzen können. Ergänzend produzieren jedoch auch Entwickler, die über die nötigen Kenntnisse verfügen, Quellcode mit Schwachstellen und Verwundbarkeiten, da es sich bei der Softwareentwicklung um eine komplexe Aktivität handelt und die Entwickler durch die Implementierung der funktionalen Anforderungen, die Optimierung der Performanz der Software und die Einhaltung von Fristen oft bereits kognitiv ausgelastet sind (Reason, 1991, Seite 19). Das Problem der kognitiven Auslastung bei der Softwareentwicklung wird auch im Rahmen von Donald Knuth’s Analyse seiner Fehler bei der Entwicklung des Textsatzsystems TeX sichtbar: „This seems to be one of my favourite mistakes: I often forget the most obvious things.“ (Knuth, 1989, Seite 609). Aus diesen Gründen existieren Werkzeuge, welche den Entwickler bei der Anwendung von Secure Coding Richtlinien unterstützen können. Kapitel 4 befasst sich mit dieser Thematik.

[26]https://wiki.sei.cmu.edu/confluence/display/java/DCL50-J.+Use+visually+distinct+identifiers, aufgerufen am 05.11.2019

Kapitel 4

Statische Code-Analyse

In diesem Kapitel werden die grundlegenden Eigenschaften, Vorgehensweisen und Zweckbestimmungen statischer Code-Analyse und ihrer praktischen Umsetzung in Form von automatisierten Werkzeugen sowohl im Allgemeinen als auch im Bezug auf die Softwaresicherheit beschrieben.

4.1 Grundlagen statischer Code-Analyse

Um eine hohe Qualität eines Softwaresystems zu gewährleisten und mögliche Fehler zu identifizieren, existieren verschiedenste Software-Prüftechniken, die sich zwei grundlegenden Kategorien unterordnen lassen: *dynamischen Tests* und *statischen Analysen* (Liggesmeyer, 2009, Seite 38).

Dynamische Testtechniken zeichnen sich dadurch aus, dass bei diesen Verfahren eine lauffähige Version der Software ausgeführt wird, etwa indem eine testende Person Eingabewerte angibt, mit denen die Software arbeitet, und anschließend ihre Ausgabe mit den erwarteten Ergebnissen vergleicht, um die Anwesenheit von Fehlern festzustellen (Bergeron, Debbabi, Erhioui, Lavoie & Tawbi, 2009, Seite 2). Statische Analysen finden hingegen zur Übersetzungszeit und ohne eine Ausführung der Software statt. Stattdessen wird eine Repräsentation des Programmcodes, beispielsweise der Quellcode oder der kompilierte Bytecode, näher betrachtet, um die Anwesenheit von Fehlern festzustellen (Heffley & Meunier, 2004, Seite 3). Während dynamische Testverfahren somit gemäß Dijkstra (1970, Seite 7) lediglich die Anwesenheit von Fehlern zeigen können, nicht aber ihre Abwesenheit, ist es mit statischen Analysemethoden zumindest in der Theorie und unter der Anwendung geeigneter Approximationen möglich, alle Ausführungspfade einer Software zu prüfen, um so Garantien über ihr Verhalten aufzustellen und auch ihre Fehlerfreiheit zu bestätigen (Møller & Schwartzbach, 2019, Seite iii).

Beide Testformen können zwar manuell von Personen durchgeführt werden, jedoch bietet sich die Verwendung von teil- oder vollautomatisierten Werkzeugen an, welche die Anwender bei der Inspektion der Software unterstützen, da die manuelle Analyse eines Softwareprojekts in seiner Gesamtheit ab einer gewissen Größe impraktikabel ist. Die im Rahmen dieser Arbeit besonders relevanten Werkzeuge zur statischen Code-Analyse werden im Folgenden als Static Code Analysis Tools (SCATs) bezeichnet. Zudem können entsprechende Werkzeuge den Programmcode deutlich schneller analysieren als Menschen und sind dabei konsistenter in der Fehlererkennung, da sie

nicht durch menschliche Eigenschaften wie die Konzentration des Testers beeinflusst werden (Leuer, 2019, Seite 29). Untersuchungen wie die von Gleirscher, Golubitskiy, Irlbeck und Wagner (2016), Ayewah und Pugh (2010) sowie Jaspan, Chen und Sharma (2007) haben gezeigt, dass SCATs die Qualität der Software erheblich fördern können, vor allem wenn sie kontinuierlich in den Entwicklungsprozess eingebunden werden. Baca, Carlsson und Lundberg (2008, Seite 85) stellten in ihrer Studie fest, dass durch die Anwendung von Werkzeugen zur statischen Code-Analyse eine Kostenreduktion von durchschnittlich 17% in der Wartung, einschließlich der Kosten die durch die Verwendung des SCAT entstehen, erreicht werden kann. Im Hinblick auf die Fehlerbehebung konnte zudem festgestellt werden, dass die Verwendung eines SCAT eine Kostenreduktion von 20% erzielen kann (RogueWave, 2017, Seite 19). Abschnitt 4.3 befasst sich konkreter mit den Vor- und Nachteilen der statischen Code-Analyse.

Wie auch durch die Studien von Zheng et al. (2006), Li und Cui (2010) oder Antunes und Vieira (2009) veranschaulicht, können SCATs keinen holistischen Ansatz darstellen, um die Softwaresicherheit in allen Belangen zu gewährleisten. Sie können die Sicherheit dennoch stark fördern, indem sie einen Teil dieser Gesamtlösung darstellen, der in den Test- und Implementierungsphasen der Softwareentwicklung durch den Entwickler selbst oder durch Tester angewendet werden kann. Statische Code-Analysen sind dabei besonders hilfreich, da viele Sicherheitsprobleme in Zuständen eintreten, die durch dynamische Testmethoden oft nur schwer erzielt werden können. Probleme, die durch statische Inspektionen des Codes einfacher als durch dynamische Tests festgestellt werden können, sind beispielsweise Buffer Overflows oder hartkodierte sensitive Informationen (Wagner, Foster, Brewer & Aiken, 2000, Seite 2).

Da SCATs bereits früh im Entwicklungsprozess angewendet werden können, eignen sie sich gut als Unterstützung für die Durchsetzung des *„Building Security In“*-Konzeptes des Secure Coding, um dem Entwickler dabei zu helfen, Kodierungsfehler zu erkennen und zu vermeiden, bevor diese durch einen Angreifer ausgenutzt werden können (Chess & West, 2007, Seite xxiii, 3, 13). Konkret werden SCATs oft auch verwendet, um zu überprüfen, ob verschiedene Secure Coding Richtlinien eingehalten werden, damit deren Durchsetzung auch gewährleistet werden kann, falls der Entwickler die jeweiligen Fehlertypen nicht kennt. Das Source Code Analysis Laboratory (SCALe) des CERT prüft beispielsweise die Konformität von Programmcode zu den Richtlinien der CERT Kodierungsstandards durch die kombinierte Verwendung diverser SCATs, da unterschiedliche Werkzeuge oft verschiedene Fehler feststellen können und so eine gründlichere Analyse ermöglicht wird (Plakosh, Seacord, Stoddard, Svoboda & Zubrow, 2014, Seite 1; Willis & Britton, 2011, Seite 13). Die automatisierte Validierung durch SCATs ist zudem die einzige praktische und skalierbare Methodik zur Überprüfung der Konformität eines Quellcodes zu einem Secure Coding Standard (Seacord, 2006, Seite 2). Im Hinblick auf die Anwendung von Secure Coding Richtlinien können dynamische Testverfahren hingegen keine Unterstützung leisten, da sie den Programmcode selbst nicht betrachten und so auch keine Ursachen für etwaige Verwundbarkeiten benennen können. Auch aus diesem Grund stellen SCATs einen der Schwerpunkte dieser Arbeit dar.

4.2 Typische Anwendungsgebiete

Statische Code-Analysen werden zu vielen unterschiedlichen Verwendungszwecken und auch an verschiedenen Stellen im Entwicklungsprozess genutzt. Dabei können sie als eigenes Programm umgesetzt oder in eine Integrated Development Environment (IDE) integriert werden. Einige der Verwendungszwecke werden im Folgenden näher beschrieben.

- **Typprüfungen:** Typprüfungen sind notwendig, um einen wichtigen Teil der Spezifikation vieler Programmiersprachen einzuhalten, etwa indem bei der Verwendung von Java sichergestellt werden muss, dass eine Variable vom Typ `int` nicht mit einem `String` initialisiert wird. Solche Typprüfungen werden herkömmlicherweise anhand von Compilern durchgesetzt, sind oft aber auch in moderneren IDEs implementiert, um den Entwickler frühstmöglich auf entsprechende Probleme hinzuweisen (Chess & West, 2007, Seite 24).
- **Stilüberprüfung:** Insbesondere im Hinblick auf die Lesbarkeit und Wartbarkeit des Quellcodes sind SCATs hilfreich, die Fehler bei der Einhaltung eines vorgegebenen Programmierstils mittels einer Stilüberprüfung erkennen. Sie bilden zudem auch die Basis für automatisierte Quelltextformatierungen, wie sie in vielen aktuellen IDEs vorhanden sind. Da Stilüberprüfer die Lesbarkeit und Wartbarkeit des Codes verbessern können, haben auch diese Werkzeuge gewissermaßen einen Bezug zur Softwaresicherheit.
- **Programmverständlichkeit:** Statische Code-Analysen können für die Umsetzung verschiedener Refaktorisierungsfunktionalitäten, wie beispielsweise das Umbenennen von Variablen oder das Auffinden der Definition einer Funktion, aber auch für abstraktere Aufgaben wie die Generierung von Unified Modeling Language (UML)-Diagrammen auf Basis des Programmcodes verwendet werden (Chess & West, 2007, Seite 27). Das Syntax-Highlighting verschiedener IDEs oder Texteditoren wird ebenfalls anhand einer statischen Code-Analyse realisiert, welche je nach Komplexität des Highlightings zumindest einen lexikalischen Scanner nutzt, um unterschiedliche Komponenten im Quellcode erkennen zu können[1]. Abschnitt 4.4 beschreibt unter anderem die grundlegende Funktionsweise eines lexikalischen Scanners.
- **Programmverifizierung:** Verifizierungswerkzeuge können verwendet werden, um zu prüfen, ob die Implementierung einer Software ihrer Spezifikation in ihrer Gesamtheit oder in Teilen entspricht (Chess & West, 2007, Seite 28).
- **Fehlererkennung:** Werkzeuge zur Fehlererkennung in Programmcode führen meist eine regelbasierte Mustererkennung durch, um Kodierungsfehler wie beispielsweise Endlosschleifen oder Code-Smells zu erkennen (Emanuelsson & Nilsson, 2008, Seite 8). Diese Werkzeuge weisen üblicherweise die in Abschnitt 4.4 beschriebene Funktionsweise auf.
- **Sicherheitsanalyse:** Bei Sicherheitsanalysen handelt es sich um Fehlererkennungen mit einem fokussierteren Ziel, da sie möglichst präzise Schwachstellen

[1]https://tomassetti.me/how-to-create-an-editor-with-syntax-highlighting-dsl/, aufgerufen am 03.02.2020

und Verwundbarkeiten in dem Programmcode eines Softwaresystems erkennen sollen. Abschnitt 4.4 beschreibt somit ebenfalls die grundlegende Funktionsweise von SCATs für die Erkennung von Schwachstellen und Verwundbarkeiten.

Mit dem Verwendungszweck eines SCAT geht der Umfang der anzustellenden Analysen einher, zu dem seine Laufzeit in etwa proportional wächst. Abbildung 4.1 veranschaulicht diesen Zusammenhang exemplarisch. Einfache Prüfungen einzelner Zeilen oder Funktionen liefern daher nahezu augenblicklich Ergebnisse, während komplexe Analysen vollständiger Programme deutlich mehr Zeit in Anspruch nehmen und einen hohen Speicherverbrauch aufweisen können (Chess & West, 2007, Seite 92).

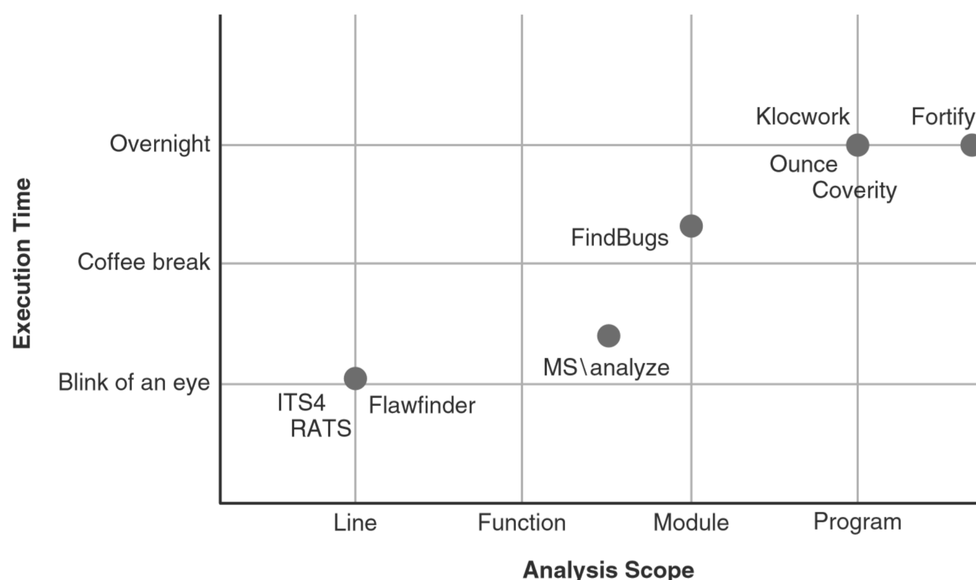

Abbildung 4.1: Exemplarische Laufzeit ausgewählter Werkzeuge zur statischen Code-Analyse. Abbildung entnommen aus Chess und West (2007, Seite 39).

Da sich diese Arbeit mit der Durchsetzung von Secure Coding Richtlinien befasst, sind SCATs, die Sicherheitsanalysen durchführen können, für den weiteren Verlauf besonders relevant. Solche Werkzeuge weisen die in Abschnitt 4.4 beschriebene Funktionsweise auf und sind nah mit Werkzeugen zur reinen Fehlererkennung verwandt. Das äußert sich auch darin, dass Sicherheitsanalysen oft durch allgemeinere Fehlererkennungswerkzeuge umgesetzt werden, etwa anhand nativ vorhandener Regeln wie in SonarQube[2] oder durch Erweiterungen wie das Plugin find-sec-bugs[3] für SpotBugs.

4.3 Vor- und Nachteile statischer Code-Analyse

Das Verfassen von Programmcode findet im Rahmen jeglicher Art von Softwareentwicklung statt, weshalb die Inspektion des produzierten Codes als effektivste Methode zur Vermeidung von Problemen angesehen werden kann (McGraw, 2006, Seite 101). SCATs können diesen Prozess unterstützen und stellen somit ein hilfreiches Werkzeug dar, um verschiedenste Arten von Implementierungsfehlern bereits frühzeitig zu

[2]https://rules.sonarsource.com/java/type/Security%20Hotspot, aufgerufen am 15.12.2019
[3]https://find-sec-bugs.github.io/, aufgerufen am 15.12.2019

erkennen und Hinweise für ihre Behebung zu geben. Bei der näheren Betrachtung stellen sich konkrete Vorteile der Verwendung von SCATs heraus, die an dieser Stelle aufgeführt werden:

- Die Verwendung von SCATs kann Code-Reviews deutlich effizienter gestalten. Chess und West (2007, Seite 57) berichten von Unternehmen, die unter der Verwendung von Analysewerkzeugen 20 Millionen, statt wie zuvor 10 Millionen Zeilen Programmcode pro Jahr inspizieren und ihre Code-Reviews in ein bis zwei Wochen statt drei bis vier Wochen durchführen konnten. Die Untersuchung von Heffley und Meunier (2004) ergab aber auch, dass insbesondere für die Analyse durch Tester, die selbst nicht mit dem Code vertraut sind, qualitativ hochwertige Werkzeuge notwendig sind, welche auch die Semantik des Codes bei der Analyse miteinbeziehen, um brauchbare Ergebnisse zu erzielen.

- Die konsistente Vermeidung verschiedener Fehlertypen stellt auch für Experten eine Herausforderung dar. Automatisierte Werkzeuge sind hierbei hilfreich, da sie anders als Menschen den gesamten Programmcode ohne Konzentrationsschwankungen betrachten und keine unbeabsichtigte Priorisierung vornehmen. So können sie auch große Codemengen deutlich homogener und schneller analysieren als Menschen (RogueWave, 2017, Seite 11). Da SCATs den Programmcode selbst betrachten, weisen sie zudem eine konstantere Fehlererkennungsrate auf als stichprobenartige Tests.

- SCATs können auch Programmierer unterstützen, die über keinerlei spezielle Kenntnisse zu den jeweiligen Fehlertypen verfügen, da sie das Wissen und die Erfahrungen anderer Entwickler bündeln und auf den Code anwenden. Eine qualitativ hochwertige Darstellung der Analyseergebnisse kann zudem in einem Lerneffekt resultieren, der für die Entwicklung sicherer Software besonders wichtig ist (Howard & Lipner, 2003, Seite 58).

- Da statische Analysen den Programmcode selbst betrachten, werden durch SCATs keine Symptome, sondern die Ursachen von Fehlern festgestellt, wodurch diese effizienter vermieden werden können (Chess & West, 2007, Seite 22).

- Indem der Quellcode für statische Analysen nicht ausgeführt werden muss, können SCATs Fehler finden, lange bevor der Entwickler dynamische Testformen anwenden kann (Nagappan & Ball, 2005, Seite 580). Dieser frühe Einsatz statischer Analysewerkzeuge bietet einen wichtigen Vorteil gegenüber dynamischer Testformen, da bei diesen der verfügbare Zeitraum für das Auffinden von Fehlern durch die Tester bis zur Veröffentlichung der Software und somit auch dem Zeitpunkt, ab dem potenzielle Angreifer versuchen Fehler zu finden, deutlich kürzer ist. Chelf und Ebert (2009, Seite 97) stellen heraus, dass die Verwendung von SCATs Fehler, die sonst erst nach der Veröffentlichung der Software entdeckt werden, um etwa 50% reduzieren kann und die damit verbundenen Entwicklungskosten um 20 bis 30% verringern kann.

- Durch die Erweiterbarkeit vieler SCATs können Ansätze für die Identifizierung einer neu entdeckten Schwachstelle oder Verwundbarkeit vergleichsweise einfach umgesetzt und schnell auf große Mengen von Programmcode angewendet

werden. Eine solche Erweiterbarkeit ist daher vor allem im Bereich der Softwaresicherheit sehr hilfreich (Bardas, 2010, Seite 4). Diese Erweiterbarkeit ermöglicht zudem die Überprüfung softwarespezifischer Anforderungen durch ein SCAT (Sun, Shu, Podgurski & Robinson, 2012, Seite 1054).

Anhand der aufgelisteten Vorteile wird schnell erkenntlich, dass SCATs sowohl im Allgemeinen als auch speziell im Hinblick auf die Softwaresicherheit eine hilfreiche Unterstützung darstellen. Sie können dabei insbesondere für die Durchsetzung des *„Building Security In“*-Prinzips genutzt werden, um Programmierfehler schon früh zu identifizieren und zu beheben, bevor sie durch Angreifer ausgenutzt werden können. Konkreter eignen sie sich auch dazu, bei der Einhaltung verschiedener Secure Coding Richtlinien zu helfen, da ihr Schwerpunkt darauf liegt, vordefinierte Fehlertypen konsistent über große Mengen Programmcode hinweg aufzufinden.

Neben diesen Vorteilen besitzen SCATs aber auch eine Reihe an inhärenten Nachteilen, die für ihre Entwicklung und Anwendung relevant sind. Das zentrale Problem von SCATs ist, dass die Ergebnisse einer Analyse oft fehlerbehaftet sind. Diese Ergebnisse können *False Positives* enthalten, also Probleme in dem analysierten Programmcode, die tatsächlich nicht existieren, oder *False Negatives* aufweisen, wenn sie fehlerbehafteten Programmcode nicht erkennen und dementsprechend diese Fehler auch dem Entwickler nicht präsentieren. Das SCAT SonarLint produziert beispielsweise die in Abbildung 4.2 dargestellte Warnung, obwohl der fragliche Quellcode fehlerfrei ist, da die Methode `super.clone()` hier absichtlich nicht aufgerufen wird. Stattdessen wird eine entsprechende `CloneNotSupportedException` erzeugt, weil Instanzen der Klasse `SensitiveClass` in diesem Beispiel gemäß der Richtlinie *OBJ07-J. Sensitive classes must not let themselves be copied*[4] nicht kopiert werden dürfen.

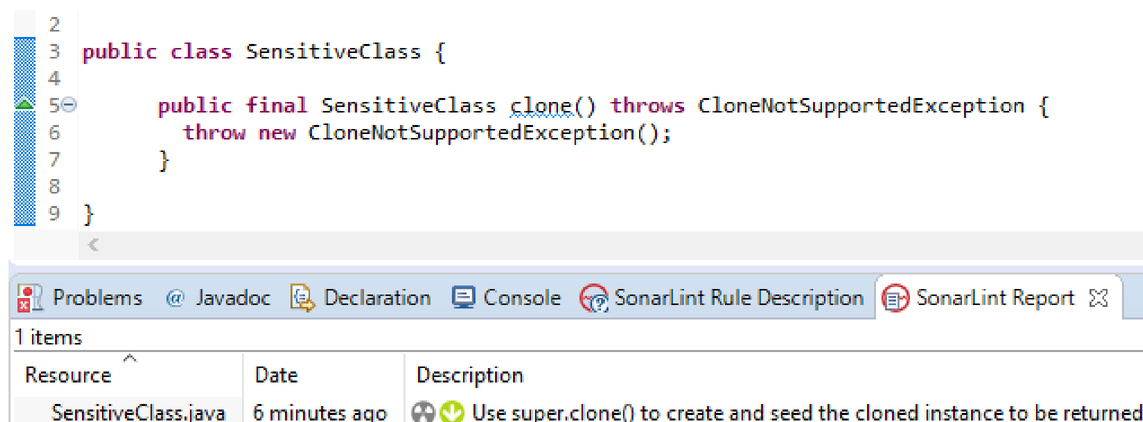

Abbildung 4.2: Beispiel für ein False-Positive Ergebnis in SonarLint

Beide Fehlertypen wirken sich negativ auf die Effizienz und Effektivität des Analyseprogramms aus. False Positives führen dazu, dass der Entwickler bei der Inspektion der Analyseergebnisse unnötig große Mengen an berichteten Fehlern untersuchen muss, um festzustellen, ob es sich jeweils tatsächlich um einen Fehler handelt oder nicht. Dies kann zum einen in einem großen Zeitverlust resultieren und zum anderen können auch *True Positives* in einer großen Menge an False Positives untergehen

[4] https://wiki.sei.cmu.edu/confluence/display/java/OBJ07-J.+Sensitive+classes+must+not+let+themselves+be+copied, aufgerufen am 03.02.2020

(Shen, Fang & Zhao, 2011, Seite 305). Im Hinblick auf Werkzeuge, die sicherheitsrelevante Schwachstellen aufspüren sollen, ist die Existenz von False Negatives jedoch relevanter als etwa bei der Auffindung von Code-Smells. Zum einen können durch False Negatives tatsächliche Probleme übersehen werden, die später in schwerwiegenden Verwundbarkeiten resultieren und zum anderen hält der Entwickler seine Software möglicherweise für sicherer als sie tatsächlich ist. Der Grund dafür ist, dass die Analyseergebnisse oft so interpretiert werden, als ob sie eine vollständige Liste an Fehlern berichten, obwohl sie eher als Hilfestellung für die Identifizierung potenziell gefährlichen Programmcodes zu verstehen sind (Chess & West, 2007, Seite 23, 34). Ware und Fox (2008, Seite 16) stellten bei einer Studie beispielsweise fest, dass selbst bei der Verwendung von acht verschiedenen SCATs nur knapp die Hälfte aller aufzufindenden Codeprobleme vollautomatisch erkannt werden konnte. Im Bezug darauf befanden Baca et al. (2008, Seite 87) für allgemeine Fehlertypen eine False Positive-Rate von bis zu 22% für akzeptabel, während eine False Negative-Rate bereits zwischen 2 und 10% als signifikant gilt (Cova, Kruegel & Vigna, 2010, Seite 288). In einer weiteren Studie stellten Baca, Petersen, Carlsson und Lundberg (2009) zudem fest, dass die Analyseergebnisse durch den Entwickler meist nicht hinterfragt werden.

Der Grund dafür, dass eine fehlerfreie statische Analyse nicht realisierbar ist, wird auf der theoretischen Ebene durch den Satz von Rice veranschaulicht. Dieser besagt, dass für jede nicht-triviale Eigenschaft eines beliebigen Programms in einer Turing-vollständigen Sprache kein Algorithmus entworfen werden kann, der feststellt, ob das Programm über diese Eigenschaft verfügt (Rice, 1953). Die Beweisführung für den Satz von Rice kann auf das Halteproblem reduziert werden, das sich mit der Frage auseinandersetzt, ob ein beliebiger Algorithmus mit einer beliebigen Eingabe endlos läuft oder jemals terminiert. Turing zeigte anhand eines Widerspruchs, dass ein Algorithmus, der das Halteproblem für alle beliebigen Algorithmen und Eingaben löst, nicht existieren kann und das Halteproblem selbst somit unentscheidbar ist (Møller & Schwartzbach, 2019, Seite 4; Turing, 1936, Seite 246). Da es sich mit der Prüfung von Algorithmen durch Algorithmen befasst, kann das Halteproblem als ein Problem der statischen Analytik angesehen werden. Entsprechend des Satzes von Rice ist auch die statische Code-Analyse im Allgemeinen somit ein unentscheidbares Problem.

Die Konsequenz dieses theoretischen Problems für die Praxis ist, dass SCATs, die fehlerfreie Analyseergebnisse liefern, nicht realisierbar sind. Verschiedene Studien wie die von Ayewah, Pugh, Hovemeyer, Morgenthaler und Penix (2008) sowie Zitser, Lippmann und Leek (2004) zeigen jedoch, dass die Ergebnisse von SCATs dennoch sehr hilfreich bei der Verbesserung der Codequalität sein können. Konkret äußert sich der Effekt der Unentscheidbarkeit bei SCATs durch die Entstehung der False Positives und False Negatives, deren Existenz bei der manuellen Überprüfung der Analyseergebnisse beachtet werden muss - das Experiment von Baca et al. (2009, Seite 808) zeigt hierbei jedoch, dass die Erkennung von False Positives vielen Entwicklern schwerfällt. False Positives und False Negatives sind Ausprägungen von zwei zentralen Eigenschaften statischer Analysewerkzeuge: *Korrektheit (Soundness)* und *Vollständigkeit (Completeness)*. Ein Analysewerkzeug, das als korrekt bezeichnet werden kann, würde keinerlei False Negatives produzieren, während ein vollständiges Werkzeug keine False Positives erzeugen würde (Long et al., 2011, Seite xxvi). Gemäß des Satzes von Rice wird jedoch ersichtlich, dass ein Werkzeug zur statischen Code-Analyse nie korrekt und vollständig sein kann und somit lediglich eine Approximation darstellt (Møller & Schwartzbach, 2019, Seite iii).

Für die praxisorientierte Verwendung von SCATs sind diese Eigenschaften gewissermaßen relevant, da dem Anwender bei der Nutzung eines solchen Werkzeugs bewusst sein muss, dass die gelieferten Ergebnisse fehlerbehaftet und unvollständig sein können. Diese Eigenschaften sind jedoch eher abstrakt und es existieren einige praktische Faktoren, deren Einschränkungen für die Brauchbarkeit von SCATs meist deutlich früher in Kraft treten. Chess und West (2007, Seite 36) führen die folgende Analogie an: Für die schnellstmögliche Geschwindigkeit eines neuen Autos stellt die Lichtgeschwindigkeit zwar eine potenzielle Einschränkung dar, diverse ingenieurtechnische Schwierigkeiten schränken die maximal erreichbare Geschwindigkeit jedoch deutlich früher ein. Dementsprechend benennen Chess und West (2007, Seite 36) für SCATs die folgenden praktischen Einschränkungen:

- Die Abwägung von Präzision und Skalierbarkeit. Je nach Anwendungszweck des Werkzeugs muss die Tiefe der Analyse des Quellcodes und der damit verbundene Speicherverbrauch sowie die entstehende Laufzeit gegeneinander abgewogen werden, da Analysen mit einer zu hohen Laufzeit oder einem zu hohen Speicherverbrauch ab einem gewissen Punkt unpraktikabel werden.

- Das Maß in dem ein Werkzeug das zu analysierende Programm verstehen kann, um den Kontext in die Analyse miteinzubeziehen und so präzisere Ergebnisse produzieren zu können. Diese Eigenschaft wird durch die Qualität des generierten Programmmodells und die verwendeten Analysealgorithmen festgelegt und bildet die Grundlage für jegliche Fehlererkennung, siehe Abschnitt 4.4.

- Die Anzahl und Definition der Fehlertypen, die das Werkzeug erkennen kann. Konkret beispielsweise in Form eines allgemeinen Regelwerks sowie durch weitere Regeln, die speziell für die jeweilige Software definiert werden, da Clients beispielsweise anfällig für andere Fehlertypen sind als Server und SCATs nur Fehlertypen identifizieren können, über die sie zuvor unterrichtet wurden. Abschnitt 4.4 beschreibt dies anhand der Komponente „Security Knowledge“.

- Die Gebrauchstauglichkeit des Werkzeugs, um den Entwickler nicht von seiner eigentlichen Arbeit abzulenken und die Ergebnisse verständlich darzustellen, damit er effizient und effektiv unterstützt werden kann. Die Gebrauchstauglichkeit eines SCAT schlägt sich beispielsweise in den in Abschnitt 4.4 beschriebenen Komponenten „Einlesen des Quellcodes“ und „Darstellung der Analyseergebnisse“ nieder.

Ergänzend dazu zeigte das Experiment von Baca et al. (2009, Seite 810), dass die bloße Verwendung von Analysewerkzeugen noch keine idealen Ergebnisse liefert. Sowohl die Kenntnisse des Entwicklers über die Vermeidung von Schwachstellen und Verwundbarkeiten, aber vor allem auch seine Kompetenz im Umgang mit dem jeweiligen Analysewerkzeug sind relevant, um gute Ergebnisse erzielen zu können. Gleichzeitig verdeutlichen Untersuchungen jedoch auch, dass insbesondere kommerzielle SCATs selten genutzt werden (Xie et al., 2011). Praktische Gründe aus denen SCATs häufig nicht verwendet werden sind somit in erster Linie, dass Entwickler durch die fehlerhaften Ergebnisse abgeschreckt werden, sowie dass die Gebrauchstauglichkeit der Werkzeuge oft als nicht gut genug gilt. Beispielsweise, weil sie sich nicht einfach genug in den Arbeitsablauf des Entwicklers integrieren lassen oder weil

die Art und Weise der Ergebnispräsentation nicht verständlich genug ist. Zusätzlich empfinden Entwickler die Hilfestellung durch SCATs bei der Behebung der gefundenen Probleme häufig als nicht ausreichend.

Für die Entwicklung und Anwendung von SCATs ist im Kontext dieser Arbeit somit festzuhalten, dass diverse einschränkende Faktoren für ihre Effizienz und Effektivität existieren, die Ergebnisse nicht immer akkurat sind und sie keine vollständige Lösung zur Sicherstellung der Softwaresicherheit leisten können. SCATs stellen dennoch ein mächtiges Hilfsmittel für die Durchsetzung der Sicherheit dar, indem sie eine effiziente Unterstützung bei der Identifizierung von sicherheitsrelevanten Problemen bieten. Bei der Anwendung eines SCAT muss sich der Anwender vor allem über die Existenz von False Positives und False Negatives bewusst sein und sollte dementsprechend die Ergebnisse eines Analysewerkzeugs nicht als vollständige Fehlerliste, sondern als Hilfestellung für die weitere manuelle Inspektion möglicher problematischer Stellen des Programmcodes interpretieren (Chess & West, 2007, Seite 33).

4.4 Allgemeine Funktionsweise von Analysewerkzeugen

Die grundlegende Funktionsweise von Werkzeugen zur statischen Code-Analyse ist vergleichbar mit jener von Programmen zur Rechtschreibprüfung. Rechtschreibprüfungen erkennen zwar falschgeschriebene Worte, indem sie diese mit einem festen Regelwerk in Form der korrekten Schreibweisen abgleichen, ein semantischer Fehler kann jedoch nicht erkannt werden, etwa wenn der Benutzer „Bach“ statt „Buch“ eingibt. SCATs verwenden üblicherweise ebenfalls feste Regelwerke, welche sie auf den Programmcode anwenden, um typische Problemfälle zu erkennen und sind gleichermaßen nicht in der Lage, die Intention des Entwicklers miteinzubeziehen. Wenn ein solches Analysewerkzeug keine Fehler im Programmcode findet, bedeutet dies ähnlich wie bei Rechtschreibprüfungen nicht, dass der Code fehlerfrei ist, sondern nur dass die Fehler, welche das jeweilige SCAT kennt und erkennen kann, nicht gefunden wurden (Chess & West, 2007, Seite 21).

Im weiteren Sinne kann auch der Unix-Befehl `grep` als ein simples Werkzeug zur statischen Code-Analyse verwendet werden, etwa um Quellcodedateien nach relevanten Begriffen wie „password“ oder den Namen sicherheitskritischer Funktionen zu durchsuchen und die Fundstellen anschließend manuell zu überprüfen (Viega, Bloch, Kohno & McGraw, 2000, Seite 258). Grep ist ein vergleichsweise einfaches Werkzeug, da es jegliche Eingaben als reine Zeichenfolge interpretiert und bei der Analyse des Programmcodes den verschiedenen Grundelementen wie Kommentaren, Stringliteralen, Deklarationen oder Funktionsaufrufen keinerlei besondere Bedeutung zukommen lässt, weshalb es keine komplexeren und präziseren Analysen durchführen kann. Dies kann in der Praxis beispielsweise auch dazu führen, dass auskommentierte Funktionsaufrufe als problematisch erkannt werden (McGraw & Chess, 2004, Seite 77).

Frühe SCATs für das Auffinden sicherheitsrelevanter Programmierfehler wie ITS4[5], RATS[6] oder Flawfinder[7] arbeiten nach ähnlichen Prinzipien. Sie gehen dabei aber

[5] https://securiteam.com/tools/6e00l0k01k/, aufgerufen am 09.12.2019
[6] https://github.com/andrew-d/rough-auditing-tool-for-security, aufgerufen am 09.12.2019
[7] https://dwheeler.com/flawfinder/, aufgerufen am 12.12.2019

intelligenter vor, indem sie Elemente wie Leerzeichen oder Kommentare auslassen und anhand einer vordefinierten Liste typische Funktionsaufrufe finden, die leicht in Verwundbarkeiten resultieren können, damit deren Verwendung anschließend durch den Anwender kontrolliert werden kann (Viega et al., 2000). Modernere SCATs sind deutlich komplexer und versuchen unter anderem auch anhand verschiedener Analysealgorithmen den Kontext des jeweiligen Codes in Betracht zu ziehen, um Verwundbarkeiten aufzudecken.

Eines der bekanntesten Werkzeuge zur statischen Code-Analyse ist *Lint*, welches bereits 1978 entwickelt wurde, um verschiedene Fehler und Probleme in C-Quellcode festzustellen und um unterschiedliche Probleme damaliger Compiler auszugleichen (S. C. Johnson, 1978). Viele der Prinzipien die Lint anwendet wurden später auch durch verschiedene Compiler übernommen (Chess & West, 2007, Seite 27). Allgemein hat die Funktionsweise moderner Werkzeuge zur statischen Analyse viel mit der von Compilern gemein, dies zeigt sich insbesondere bei einer näheren Betrachtung der Implementierungsprinzipien von SCATs. Abbildung 4.3 stellt die wesentlichen Komponenten von modernen SCATs dar, die im Folgenden näher beschrieben werden, um zu veranschaulichen, welche grundlegenden Analysemöglichkeiten diese Werkzeuge bieten. Diese Komponenten finden insbesondere auch in Werkzeugen Verwendung, die Fehlererkennungen oder Sicherheitsanalysen vornehmen.

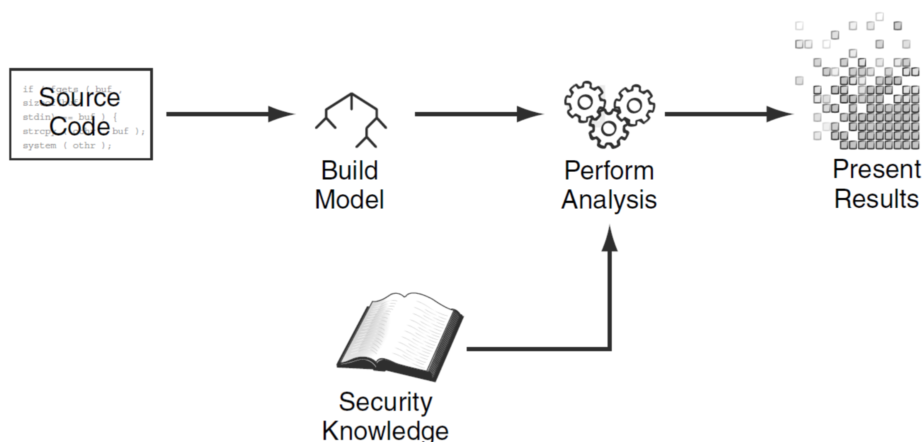

Abbildung 4.3: Die wesentliche Funktionsweise von Werkzeugen zur statischen Code-Analyse. Abbildung entnommen aus Chess und West (2007, Seite 71).

Zunächst liest jedes SCAT den zu analysierenden Programmcode ein, entweder in der Form des Quellcodes oder des kompilierten Bytecodes (Novak, Krajnc & Žontar, 2000). Die Analyse des Bytecodes hat einige Vorteile, beispielsweise kann sie auch durchgeführt werden, wenn der Quellcode selbst nicht verfügbar ist oder nicht eindeutig ist, ob ein bestimmter Quellcode zu einem Binärprodukt gehört (Schönefeld, 2011, Seite 231). Bei einer Analyse des kompilierten Codes können zudem auch Schwachstellen und Verwundbarkeiten in importierten Bibliotheken aus anderen Quellen erkannt werden (Long et al., 2011, Seite xxvii). Nachteile einer Bytecode-Analyse sind insbesondere, dass der Bytecode in der Regel deutlich komplexer und schwerer verständlich ist als der Quellcode und dass die Intention des Programmierers durch Optimierungen im Rahmen der Kompilierung verloren gehen kann. Zusätzlich ist eine Analyse, bei welcher der Quellcode ggf. auch ergänzend zum Bytecode vorliegt für den Ent-

wickler hilfreicher, da die Analyseergebnisse hierbei mit einem konkreten Bezug auf die relevanten Stellen im Quellcode dargestellt werden können (McGraw & Chess, 2004, Seite 76; Chess & West, 2007, Seite 43). Anders als bei der Verwendung des Quellcodes können Bytecode-Analysen zudem nicht durchgeführt werden, während der Quellcode gerade noch geschrieben wird oder sich in einem nicht kompilierbaren Zustand befindet. Dementsprechend basieren beispielsweise die Hinweise, die eine IDE zu dem Typ einer Variable liefert, meist auf einer statischen Analyse des Quellcodes.

In dem darauffolgenden Schritt transformieren SCATs den rein textbasierten Programmcode üblicherweise in ein besser analysierbares *Programmmodell* in Form verschiedener, leichter auswertbarer Datenstrukturen. Der genaue Aufbau des Modells hängt dabei von der Art der durchzuführenden Analysen ab. Folgend werden die Prinzipien einiger typischer Schritte zum Aufbau eines Programmmodells veranschaulicht.

Zunächst wird der Programmcode üblicherweise anhand eines lexikalischen Scanners in einzelne lexikalische Einheiten, sogenannte *Tokens*, unterteilt, wobei unwichtige Bestandteile wie Leerzeichen entfernt werden (Chess & West, 2007, Seite 72). Anhand der in Listing 4.2 dargestellten Regeln könnte eine lexikalische Analyse beispielsweise den folgenden Token-Stream für den in Listing 4.1 dargestellten Java-Quellcode produzieren[8]:

```
IF LPAREN ID(value) RPAREN ID(x) ASSIGN ID(y) SEMI
```

```
1 if(value)
2   x = y;
```

Listing 4.1: Java-Quellcode zur Veranschaulichung einer lexikalischen Analyse

```
1 if            { return IF; }
2 (             { return LPAREN; }
3 )             { return RPAREN; }
4 =             { return ASSIGN; }
5 ;             { return SEMI; }
6 /[ \t\n]+/    { /* ignore whitespace */ }
7 /[a-zA-Z]+/   { return ID; }
```

Listing 4.2: Beispielhafte Regeln für eine lexikalische Analyse

In dem darauffolgenden Schritt wird der produzierte Token-Stream von einem Sprachparser durch die Verwendung einer kontextfreien Grammatik in ein erstes Programmmodell in Form eines *Parsebaums* übertragen, welcher sich aufgrund seiner Struktur für die maschinelle Weiterverarbeitung und grundlegende Analysen bereits deutlich besser eignet als die ursprüngliche reine Textrepräsentation. Anhand der in Listing 4.3 aufgeführten Produktionsregeln könnte beispielsweise der in Abbildung 4.4 dargestellte Parsebaum[9] für den zuvor angegebenen Token-Stream generiert werden.

[8] Bei dem hier angegebenen Beispiel handelt es sich um eine vereinfachte Abwandlung des Beispiels von Chess und West (2007, Seite 72).

[9] Die hier verwendeten Begrifflichkeiten wie „IfStatement“ „Assignment“ oder „Simplename“ sind an das Werkzeug Eclipse AST View angelehnt, https://www.eclipse.org/jdt/ui/astview/, aufgerufen am 03.02.2020.

```
1 Statement    →  IfStatement | Assignment
2 IfStatement  →  IF LPAREN Simplename RPAREN Statement
3 Simplename   →  ID
4 Assignment   →  Simplename ASSIGN Simplename SEMI
```

Listing 4.3: Beispielhafte Produktionsregeln für einen Sprachparser

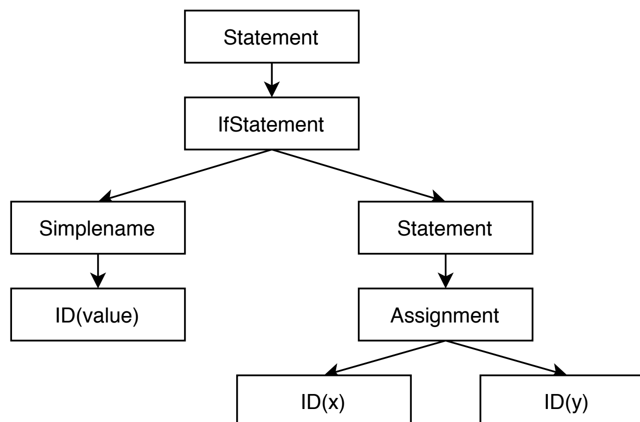

Abbildung 4.4: Beispiel für einen Parsebaum

Der Parsebaum ist eine sehr direkte Repräsentation des ursprünglichen Programmcodes, die sich bereits für verschiedene Analysen eignet. Eine weitere Abstrahierung in Form eines *Abstract Syntax Tree (AST)* bietet sich oft jedoch an, um eine standardisierte und leichter verständliche Repräsentation zu erreichen, bei der beispielsweise syntaktischer Zucker entfernt und For-Schleifen in While-Schleifen umgewandelt werden, um das ursprüngliche Programm zu simplifizieren und so leichter analysieren zu können (Chess & West, 2007, Seite 75). Durch diesen Standardisierungsprozess können jedoch auch Informationen über die Intention des Entwicklers verloren gehen. Abbildung 4.5 veranschaulicht einen möglichen AST, auffällig ist hierbei, dass die Bedingung der If-Abfrage verallgemeinert und ein leerer Zweig für den nicht vorhandenen Else-Teil der Abfrage ergänzt wird. Der AST entspricht somit weniger exakt dem ursprünglichen Quellcode und weist stattdessen eine standardisiertere Form auf.

Neben diesem grundlegenden Programmmodell werden bei modernen SCATs noch verschiedene weitere Komponenten wie beispielsweise *Symboltabellen*, die unter anderem Typinformationen bereitstellen, verwendet, um erste semantische Informationen erfassen, sowie Typanalysen, Kontrollflussanalysen oder Pointeranalysen durchführen zu können (Møller & Schwartzbach, 2019, Seite 14; McGraw & Chess, 2004, Seite 77). Diese Schritte entsprechen im Wesentlichen auch der Funktionsweise moderner Compiler. Die weitere Vorgehensweise für den Aufbau des Programmmodells ist in der Regel abhängig von dem Funktionszweck des Analysewerkzeugs: Das Aufstellen von *Kontrollflussgraphen* ermöglicht etwa Analysen der verschiedenen Ausführungspfade auf intraprozeduraler Ebene, während *Aufrufgraphen* den Kontrollfluss zwischen einzelnen Methoden und Funktionen repräsentieren (Chess & West, 2007, Seite 77, 78). Abbildung 4.6 veranschaulicht einen simplen Aufrufgraphen für den in Listing 4.4 dargestellten Quellcode.

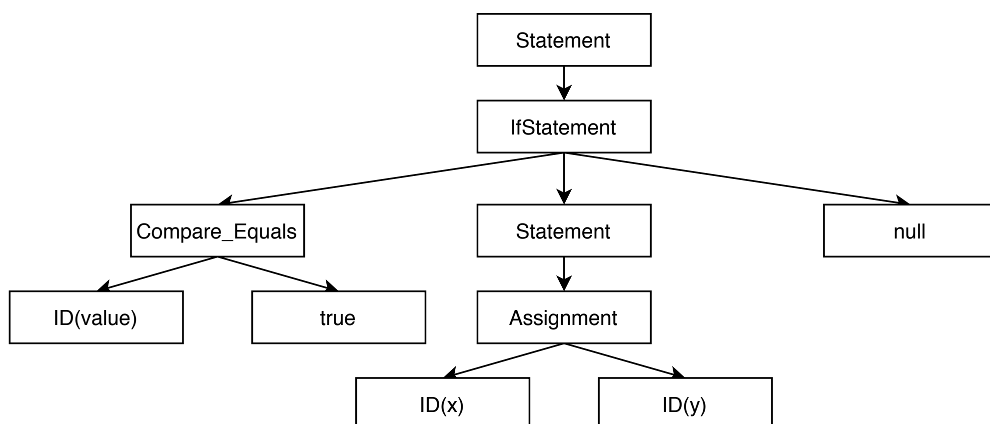

Abbildung 4.5: Beispiel für einen Abstract Syntax Tree

Kontrollflussgraphen ermöglichen zudem vor allem *Datenflussanalysen*, um festzustellen, wo Daten generiert und verwendet werden. Die *Taint Propagation* ist dabei eine für die Softwaresicherheit besonders wichtige Unterart der Datenflussanalysen, anhand derer identifiziert werden kann, welche Daten in einem Programm durch Angreifer kontrolliert werden können und wie diese sich durch das Programm ausbreiten (Chess & West, 2007, Seite 82). Für die in Listing 4.4 dargestellte Funktion `run()` könnte beispielsweise der in Abbildung 4.7 dargestellte Kontrollflussgraph generiert werden. Eine Taint Propagation Analyse könnte diesen Graphen verwenden, um zu erkennen, dass die Variable, die der sicherheitskritischen Methode `Runtime.exec()` übergeben wird, nur auf dem Pfad [a, x1, x2] bereinigt wird und daher im Falle des Pfades [a, y] eine Verwundbarkeit vorliegt[10,11].

```
1  int run(int a, String userInput) {
2    if (a == 1) {
3      String input = sanitize(userInput);
4      runtime.exec(input);
5    } else {
6      runtime.exec(userInput);
7    }
8    return 0;
9  }
10
11 String sanitize(String userInput) {
12   if(Pattern.matches("[0-9A-Za-z@.]+", userInput)) {
13     return userInput;
14   } else {
15     return DEFAULT;
16   }
17 }
```

Listing 4.4: Quellcode ohne garantierte Bereinigung einer Benutzereingabe

[10] https://wiki.sei.cmu.edu/confluence/display/java/IDS07-J.+Sanitize+untrusted+data+passed+to+the+Runtime.exec%28%29+method, aufgerufen am 03.02.2020

[11] Beispiel angelehnt an Chess und West (2007, Seite 78).

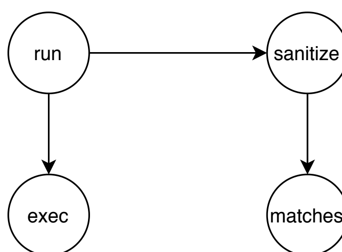

Abbildung 4.6: Beispiel für einen Aufrufgraphen

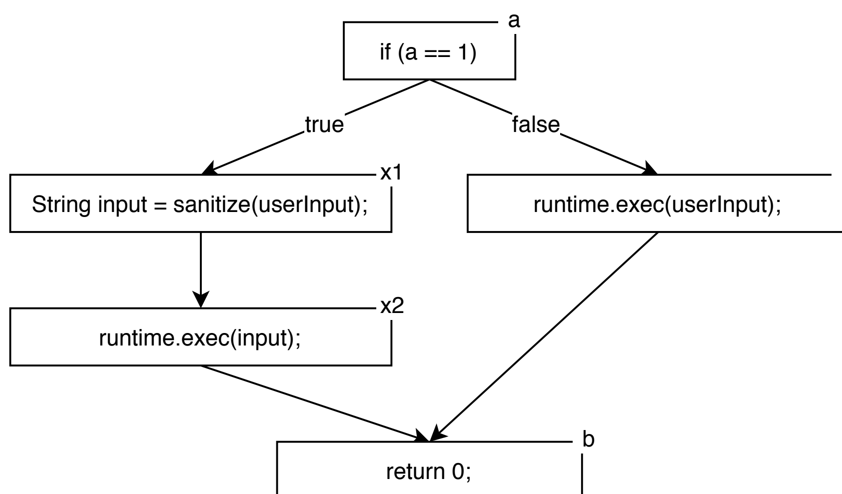

Abbildung 4.7: Beispiel für einen Kontrollflussgraphen

Nachdem ein Programmmodell des zu analysierenden Codes generiert wurde, werden verschiedene Analysetechniken anhand der zuvor erstellen Datenstrukturen angewendet, um Aussagen über den eingelesenen Programmcode zu treffen. Einfachere Werkzeuge führen etwa Mustererkennungen durch, um festzustellen an welchen Stellen möglicherweise problematische Funktionen aufgerufen werden, beispielsweise wenn eine vorhersehbare Zufallszahl per `new Random()` in Java generiert wird. Frühe und vergleichsweise simple Werkzeuge wie ITS4, RATS oder Flawfinder führen solch eine statische Analyse etwa anhand des Token-Streams durch (Viega et al., 2000, Seite 259; Karim, 2015, Seite 26). Komplexere Analysealgorithmen zeichnen sich im Vergleich dazu insbesondere durch ihre Kontextsensitivität aus, beispielsweise indem sie versuchen zu erkennen, ob der jeweilige Aufruf überhaupt sicherheitsrelevant ist. Solche Analyseverfahren werden dabei auf zwei unterschiedlichen Ebenen durchgeführt: Zum einen kann eine *lokale Analyse* der einzelnen Funktionen durchgeführt werden, wobei üblicherweise der vorher aufgestellte Abstract Syntax Tree und der Kontrollflussgraph verwendet werden, und zum anderen können in einer *globalen Analyse* die Interaktionen der einzelnen Funktionen miteinander unter der Verwendung des generierten Aufrufgraphen betrachtet werden. Abbildung 4.8 veranschaulicht den Zusammenhang zwischen lokaler und globaler Analyse bei ihrer Verwendung in einem Analysealgorithmus sowie die wesentlichen Datenstrukturen, die bei der jeweiligen Analyseform verwendet werden.

Weitere Details für die Durchführung und Implementierung statischer Analysen sind für die hier beschriebene grundlegende Funktionsweise nicht relevant. Chess und

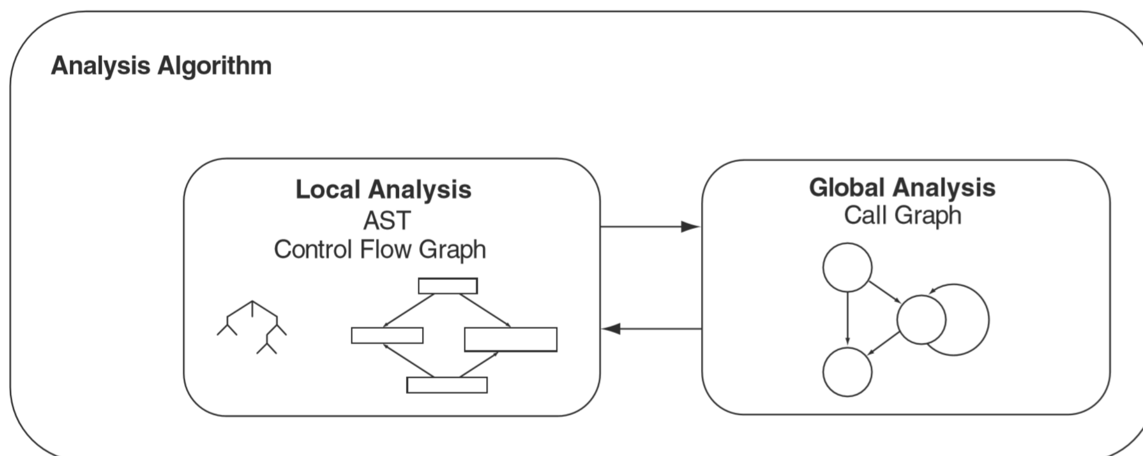

Abbildung 4.8: Zusammenhang von lokaler und globaler Analyse in einem Analysealgorithmus. Abbildung entnommen aus Chess und West (2007, Seite 83).

West (2007) sowie Møller und Schwartzbach (2019) liefern beispielsweise einen detaillierteren Überblick über diese Themen.

Nach der Durchführung der Analyse müssen die gefundenen Probleme dem Benutzer in einer adäquaten Art und Weise als Ergebnisse präsentiert werden. Dieser Punkt ist ebenso wichtig wie das Aufstellen des Programmmodells und die Analyse selbst, da der Entwickler die Relevanz der Ergebnisse korrekt einschätzen und seinen eigenen Programmcode basierend darauf überarbeiten muss. Daher muss das Analysewerkzeug die Probleme und ihren Einfluss auf die Sicherheit der Software gut verständlich darstellen und auch ganz allgemein eine möglichst gute Gebrauchstauglichkeit aufweisen, um den Anwender effektiv und effizient unterstützen zu können.

Das Einlesen des zu analysierenden Programmcodes, das Generieren eines Programmmodells sowie die verwendeten Analysealgorithmen und die Darstellung der Ergebnisse ergeben die grundlegende Funktionsweise von Werkzeugen zur statischen Code-Analyse. Das Erstellen eines geeigneten Programmmodells, sowie die Wahl und Umsetzung geeigneter Analysealgorithmen bilden dabei zwar das Grundgerüst für die Analyse, dem Werkzeug muss zudem jedoch auch das Wissen über die konkret aufzufindenden Probleme vermittelt werden.

Chelf und Ebert (2009) benennen zu diesem Zweck sogenannte *Fehlerklassen* welche diese Fehlertypen, etwa eine Division durch Null oder einen Buffer Overflow, formal beschreiben. Die praktische Umsetzung dieser Fehlerklassen findet sich in dem definierten Regelwerk wieder, welches ein Analysewerkzeug für die eigentliche Fehlererkennung nutzt (Chess & West, 2007, Seite 96). Diese *Regeln* werden in entsprechenden SCATs meist in externen Dateien bereitgestellt und können so angepasst, erweitert oder entfernt werden, damit das jeweilige Werkzeug auf die Bedürfnisse des Entwicklers bzw. des zu überprüfenden Softwaresystems angepasst werden kann, ohne dass die Implementierung des Werkzeugs selbst modifiziert werden muss (Chess & West, 2007, Seite 97). Da die Regeln des Analysewerkzeugs das Wissen über die aufzufindenden Fehlerklassen für das SCAT verständlich beschreiben, muss der Verfasser dieser Regeln über gute Kenntnisse der jeweiligen Thematik verfügen (Ersoy & Sözer, 2016, Seite 30). In der zuvor dargestellten Abbildung 4.3 wird dieser Punkt durch die für Sicherheitsprobleme spezifische Komponente *„Security Knowledge“* festgehalten. Im Hinblick auf sicherheitsrelevante Fehler können die verschiedenen Secure

Coding Richtlinien typischerweise als das Äquivalent zu den allgemeineren Fehlerklassen angesehen werden. Dementsprechend können Regeln für SCATs aus den einzelnen Richtlinien abgeleitet werden. Die unter anderem in dem SCAT SonarQube verwendete Regel *Formatting SQL queries is security-sensitive*[12] ist beispielsweise eine praktische Umsetzung der Secure Coding Richtlinie *IDS00-J. Prevent SQL injection*[13] des CERT-Kodierungsstandards für die sichere Programmierung in Java.

Da fehlerhafte oder nicht vorhandene Eingabevalidierungen einen besonders weit verbreiteten Fehlertyp darstellen, der beispielsweise auch SQL-Injektionen behandelt, wird an dieser Stelle anhand des in Listing 4.5 dargestellten Quellcodes ein Beispiel für das Prinzip von Taint Propagation Regeln beschrieben, die für das Auffinden vieler Fehler dieser Art genutzt werden können.

```
1 public static void main(String[] args) {
2   String param = getParam(args);
3   param = sanitize(param);
4   use(param);
5 }
```

Listing 4.5: Quellcodebeispiel für das Prinzip von Taint Propagation Regeln

Regeln zur Definition von Taint Propagation Problemen definieren üblicherweise zunächst *Quellen (Sources)*, an denen Benutzereingaben in das Programm eintreten, in diesem Fall der Parameter der Funktion `main()`. Anhand von *Durchgaben (Pass-throughs)* wird beschrieben, wie sich der Zustand einer Benutzereingabe bei einem Funktionsaufruf auf andere Variablen auswirkt. In diesem Fall sollte eine entsprechende Regel für die Funktion `getParam()` definiert werden, die dem Analysewerkzeug mitteilt, dass der Rückgabewert dieser Funktion ebenfalls als potenziell schädlich anzusehen ist. Regeln für *Reinigungen (Cleanses)* können Funktionen definieren, welche die Eingabe validieren oder bereinigen, wie in diesem Beispiel die Funktion `sanitize()`. *Senken (Sinks)* beschreiben Funktionen, die keine schädlichen Eingaben erhalten dürfen, hier die Funktion `use()`. Über *Einstiegspunkte (Entry-Points)* wird dem Analyseprogramm zudem mitgeteilt, welche weiteren Funktionen durch einen Angreifer aufgerufen werden können (Chess & West, 2007, Seite 101, 102). Anhand dieser Regeln und einer Datenflussanalyse könnte ein automatisiertes Werkzeug in diesem Beispiel feststellen, dass keine Verwundbarkeit vorliegt, da der Wert der Variable `param` vor seiner Verwendung in der Senke `use()` bereinigt wird.

Neben der Anpassung anhand von Regeln ermöglichen einige Analysewerkzeuge, wie beispielsweise SpotBugs, auch einen Zugriff auf die zur Analyse verwendeten Datenstrukturen und Algorithmen auf der Ebene des Quellcodes und ermöglichen so die Entwicklung von Plugins. Ebenso wie bei der Aufstellung der Regeln sind auch bei der Entwicklung von Plugins detaillierte Kenntnisse über die aufzufindenden Fehler essenziell, da ein Analysewerkzeug grundsätzlich nur die ihm bekannten Fehlertypen erkennen kann (Chess & West, 2007, Seite 96). Anhand von Regeln und auch programmatischen Zugriffsmöglichkeiten auf die interne Umsetzung des SCAT

[12] https://rules.sonarsource.com/java/RSPEC-2077, aufgerufen am 03.12.2019

[13] https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection, aufgerufen am 03.12.2019

bieten qualitativ hochwertigere Analysewerkzeuge Möglichkeiten, um die Analyse zu individualisieren und so für den jeweiligen Anwendungszweck zu optimieren.

Durch ihre Funktionsweise sind SCATs praktisch ausschließlich auf die Implementierungsdetails des zu analysierenden Softwaresystems fokussiert und können lediglich Defekte erkennen, die durch eine Inspektion des Programmcodes ersichtlich werden können. Dies verdeutlicht einen Aspekt, aufgrund dessen die statische Code-Analyse prinzipiell keine holistische Lösung für die Softwaresicherheit bieten kann, da Verwundbarkeiten, die durch Fehler in dem Entwurf und der Architektur der Software entstehen, üblicherweise nicht betrachtet werden (Novak et al., 2000, Seite 2). SCATs können jedoch einen sehr hilfreichen Ansatz für die Durchsetzung verschiedener Secure Coding Richtlinien darstellen, da diese sich wie in Kapitel 3 beschrieben ebenfalls ausschließlich mit der Vermeidung von Schwachstellen und Verwundbarkeiten auf der Implementierungsebene befassen.

4.5 Beispiele für traditionelle Analysewerkzeuge

Es existiert eine große Menge an verschiedenen Werkzeugen, welche statische Code-Analysen zu unterschiedlichen Verwendungszwecken durchführen[14]. Werkzeuge, die allgemeine oder sicherheitsspezifische Codeprobleme auffinden, funktionieren meist nach dem in Abschnitt 4.4 beschriebenen traditionellen Ansatz. SCATs die diesen Ansatz verwenden, der sich durch die zwei zentralen Schritte der vollautomatischen Erkennung von Fehlern und der Ausgabe der gefundenen Ergebnisse in Form einer Liste definiert, werden im Folgenden als **traditionelle SCATs** bezeichnet.

Einige dieser Werkzeuge werden als eigenes Programm ausgeführt, wie beispielsweise SpotBugs, SonarQube, CodeSonar, JLint, PMD oder ErrorProne. In diesem Fall muss der Entwickler den zu analysierenden Programmcode dem SCAT manuell übergeben. Abbildung 4.9 stellt den Aufbau des SCAT SpotBugs und die Darstellung der potenziellen Fehler exemplarisch dar. Diese Abbildung veranschaulicht zudem, dass dieser traditionelle Ansatz dem Entwickler lediglich bei dem Verbessern der Sicherheit des Quellcodes hilft, indem eine Beschreibung des Problems und ein Lösungsansatz angegeben wird. Die entsprechenden Änderungen müssen durch den Entwickler basierend auf dieser Beschreibung zunächst ggf. abstrahiert und dann manuell vorgenommen werden. Die Regeln, die mit einem SCATs standardmäßig ausgeliefert werden, sind zudem oft auch für den Entwickler verständlich dokumentiert, wie beispielsweise in SonarLint und SonarQube[15] oder SpotBugs[16] und können genutzt werden, um mehr über mögliche Schwachstellen und Verwundbarkeiten zu erfahren.

Diverse Werkzeuge zur Unterstützung bei dem Verfassen sicheren Quellcodes werden nicht als eigenes Programm ausgeführt, sondern können als Teil der IDE des Entwicklers genutzt werden. Beispiele hierfür sind SonarLint, ReSharper, Klocwork oder Fortify Static Code Analyzer. Einige eigenständige SCATs wie beispielsweise SpotBugs können zudem anhand eines Plugins in eine IDE eingebunden werden. Dies hat den Vorteil, dass das Werkzeug einfacher in den Arbeitsablauf des Entwicklers integriert werden kann. Fehler können so außerdem bereits während der Entwickler an

[14]https://en.wikipedia.org/wiki/User:Nickj/List_of_tools_for_static_code_analysis, aufgerufen am 02.02.2020

[15]https://rules.sonarsource.com/, aufgerufen am 04.02.2020

[16]https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html, aufgerufen am 04.02.2020

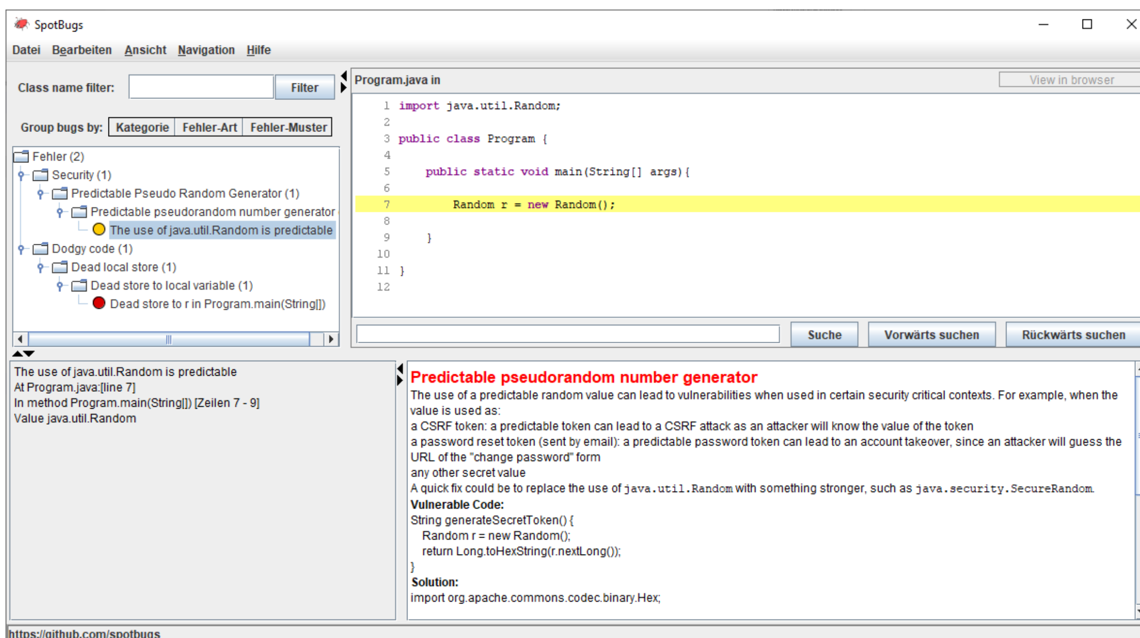

Abbildung 4.9: Aufbau und Ergebnisdarstellung des Analysewerkzeugs SpotBugs

dem Quellcode arbeitet, an den relevanten Stellen im Code angezeigt und behoben werden, da das Einlesen des Quellcodes, die Durchführung der Analyse und die Ergebnisausgabe in der Regel automatisch geschieht, sobald der Code gespeichert wird. Abbildung 4.10 veranschaulicht einen durch das Werkzeug SonarLint gefundenen Fehler in der IDE Eclipse. Da in eine IDE integrierte SCATs genauso wie eigenständige Werkzeuge in der Regel den zuvor beschriebenen traditionellen Ansatz zur Unterstützung des Entwicklers umsetzen, wird auch hier eine Hilfestellung zur Behandlung des gefundenen Fehlers lediglich in textueller Form bereitgestellt.

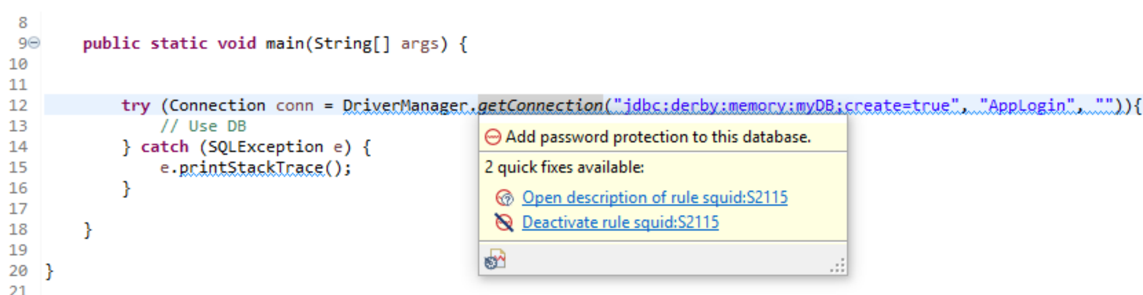

Abbildung 4.10: Fehlerdarstellung in dem Analysewerkzeug SonarLint

Kapitel 5

Eingrenzung des Schwerpunktes für die Entwicklung neuer Ansätze

Dieses Kapitel befasst sich mit der Betrachtung bereits existierender nicht-traditioneller Ansätze zur Unterstützung des Entwicklers bei dem Verfassen sicheren Quellcodes unter der Verwendung statischer Code-Analysen sowie der Priorisierung verschiedener Arten von Schwachstellen und Verwundbarkeiten, um den Schwerpunkt für die in dieser Arbeit zu entwickelnden Ansätze zu definieren.

5.1 Vorgehensweise

In den vorherigen Kapiteln wurde veranschaulicht, dass das Verfassen sicheren Quellcodes essenziell für die Gewährleistung der Softwaresicherheit ist sowie dass Secure Coding Richtlinien verschiedenste Schwachstellen und Verwundbarkeiten beschreiben und klare Vorgehensweisen für ihre Vermeidung definieren. Häufig werden diese Richtlinien jedoch nicht angewandt, etwa weil sie den Entwicklern nicht bekannt sind oder weil der Entwickler durch die Implementierung der funktionalen Anforderungen des jeweiligen Softwareprojekts bereits kognitiv ausgelastet ist. Aus diesem Grund existieren SCATs, die statische Analysen des Programmcodes durchführen, um den Entwickler auf potenzielle Sicherheitsprobleme aufmerksam zu machen. Bei der näheren Betrachtung der Vor- und Nachteile solcher traditioneller SCATs wurde ersichtlich, dass diese Werkzeuge bereits eine gute Hilfestellung für die Verbesserung der Sicherheit des Quellcodes bieten, indem sie unterschiedlichste Fehlertypen automatisch erkennen, die gefundenen Probleme beschreiben und oft auch Hinweise für ihre Behebung geben. Gleichzeitig wurde aber auch deutlich, dass traditionelle SCATs noch keine ideale Lösung für das Verfassen sicheren Quellcodes darstellen, beispielsweise weil die Analysen oft auch fehlerhafte Ergebnisse in der Form von False Positives oder False Negatives produzieren, weil die Erklärungen der gefundenen Fehler nicht verständlich sind oder weil die Werkzeuge aufgrund unzureichender Gebrauchstauglichkeit von den Softwareentwicklern nicht akzeptiert werden. Zudem leisten solche traditionellen SCATs abseits einer Beschreibung der gefundenen Fehler und möglicher Vorgehensweisen für ihre Vermeidung auch keine direkte Hilfestellung bei ihrer Behebung. Dementsprechend stellt sich die Frage, ob und wie der Entwickler bei dem Verfassen sicheren Quellcodes besser unterstützt werden kann als durch traditionelle SCATs.

Goseva-Popstojanova und Perhinschi (2015, Seite 32) stellen fest, dass sowohl die Umsetzung der Werkzeuge selbst, als auch die von ihnen genutzten Techniken über Verbesserungspotenzial verfügen. Um eine bessere Unterstützung des Entwicklers zu ermöglichen, können dementsprechend die folgenden Schwerpunkte näher betrachtet werden:

1. Eine Verbesserung der traditionellen SCATs. Beispielsweise durch die Entwicklung neuer, oder die Verbesserung bestehender Algorithmen oder Regeln, um die Anzahl an False Positives und False Negatives zu reduzieren. Auch die Optimierung der Gebrauchstauglichkeit ist hier relevant, damit die Zielgruppe diese Art von Werkzeugen eher akzeptiert und in ihren Arbeitsablauf integriert und die berichteten Fehler besser versteht, um sie korrigieren zu können.

2. Eine Entwicklung von nicht-traditionellen Ansätzen zur Unterstützung bei dem Verfassen sicheren Quellcodes, die sich von dem Ansatz traditioneller SCATs unterscheiden, welche wie in Abschnitt 4.4 beschrieben den Programmcode einlesen, vollautomatisch analysieren und anschließend eine Liste potenzieller Fehler präsentieren. So kann untersucht werden, inwiefern nicht-traditionelle Ansätze den Entwickler bei der Umsetzung sicherer Software effizienter und effektiver unterstützen können als die traditionellen SCATs.

Im Hinblick auf diese Schwerpunkte wird in dieser Arbeit die Entwicklung nicht-traditioneller Ansätze für ausgewählte Secure Coding Richtlinien näher betrachtet. Zu diesem Zweck werden zwei grundsätzliche Forschungsfragen definiert:

> **Forschungsfrage 1:** Welche neuen nicht-traditionellen Ansätze sind für die Unterstützung des Entwicklers bei der Anwendung spezifischer Secure Coding Richtlinien denkbar?
>
> **Forschungsfrage 2:** Inwiefern sind diese Ansätze sinnvoll und inwiefern können sie den Entwickler effektiver und effizienter bei der Anwendung spezifischer Secure Coding Richtlinien unterstützen als traditionelle SCATs?

Wie zuvor beschrieben werden traditionelle SCATs üblicherweise in der Implementierungsphase durch die Programmierer oder in der Testphase durch die Tester angewendet. Für die hier entworfenen Ansätze liegt der Fokus auf der Unterstützung der Entwickler, damit diese die verschiedenen Secure Coding Richtlinien direkt während der Implementierung anwenden und so die Sicherheit des Programmcodes möglichst früh gewährleisten können. Dies kann auch die nötige Zusammenarbeit zwischen den Entwicklern und den Testern verringern und in einer Kosten- und Aufwandsreduktion resultieren, da Fehler, die bereits möglichst früh ausgeschlossen werden, nicht erst später in dem Entwicklungszyklus durch Tester gefunden und dann durch die Entwickler behoben werden müssen (Zhu, Xie, Lipford & Chu, 2013, Seite 2).

Um diese neuen Ansätze zu entwickeln, werden zunächst in Abschnitt 5.2 verschiedene nicht-traditionelle Ansätze aufgeführt, welche den Entwickler bei der Umsetzung sicheren Quellcodes im Zusammenhang mit einer statischen Code-Analyse unterstützen. So wird ein Überblick über bereits existierende Ansätze sowie ihre Konzepte und mögliche Forschungslücken geschaffen. Darauffolgend werden in Abschnitt 5.3 einige

Secure Coding Richtlinien basierend auf ihrer Priorität für die nähere Betrachtung in dieser Arbeit ausgewählt. Dies ist notwendig, um neue Ansätze entwickeln zu können, welche auf die Eigenheiten der jeweiligen Richtlinien zugeschnitten sind. Abschnitt 5.4 definiert basierend auf den Erkenntnissen dieses Kapitels den Schwerpunkt für die in Kapitel 6 entwickelten, neuen und nicht-traditionellen Ansätze zur Unterstützung des Entwicklers bei der Anwendung der ausgewählten Secure Coding Richtlinien.

5.2 Existierende nicht-traditionelle Ansätze

In diesem Abschnitt werden einige bereits existierende und besonders relevante nicht-traditionelle Ansätze für die Unterstützung des Programmierers bei der Entwicklung sicherer Software beschrieben, die im Zusammenhang mit einer statischen Code-Analyse stehen. Diese zeichnen sich dadurch aus, dass sich ihr Konzept von den zuvor dargestellten traditionellen Ansätzen unterscheidet, beispielsweise indem sie andere Vorgehensweisen verfolgen oder eine Ergänzung des traditionellen Ansatzes darstellen. Die an dieser Stelle beschriebenen Ansätze wurden exemplarisch ausgewählt, da sie grundlegende Prinzipien veranschaulichen, die bei der Recherche zu nicht-traditionellen Möglichkeiten besonders auffallend waren. Einen offensichtlichen Bezug zur Softwaresicherheit müssen diese Ansätze nicht aufweisen, da es hier in erster Linie darum geht, einen Überblick über weitere Möglichkeiten im Zusammenhang mit statischer Code-Analyse zu schaffen.

5.2.1 ASIDE

Ansatz

Lipford et al. (2014) beschreiben mit dem Plugin ASIDE für die IDE Eclipse einen interaktiven Ansatz unter der Verwendung von grafischen Annotationen in der Form von Textmarkierungen und statischer Code-Analyse für Java und PHP. Der entwickelte Prototyp hilft dem Entwickler bei dem Verfassen sicheren Quellcodes im Hinblick auf die Umsetzung adäquater Zugriffskontrollen und die Vermeidung von CSRF.

Das grundlegende Konzept dieses Ansatzes sieht vor, dass zunächst ähnlich wie bei den Regeln traditioneller SCATs einige sicherheitssensitive Operationen (Security Sensitive Operations (SSOs)), wie beispielsweise Funktionen die Datenbankzugriffe tätigen, definiert werden. Während der Implementierung führt ASIDE statische Analysen des Quellcodes durch, um Aufrufe der zuvor definierten SSOs aufzuspüren. Wird eine solche Operation gefunden, erwartet ASIDE, dass der Entwickler die Stellen in seinem Quellcode anhand einer Selektion markiert, welche für die Sicherheit der jeweiligen Operation relevant sind, wie beispielsweise die nötigen Zugriffskontrollen für einen Datenbankzugriff. Außerdem kann eine Beschreibung des Problems eingesehen oder die Warnung entfernt werden. So wird der Entwickler ggf. zunächst daran erinnert, fehlende Zugriffskontrollen zu implementieren.

Die anhand der grafischen Annotationen markierten Zusammenhänge im Quellcode werden zudem als Basis für weitere statische Code-Analysen verwendet (Lipford et al., 2014, Seite 18). ASIDE durchsucht nach dem Hinzufügen einer Annotation den restlichen Quellcode nach weiteren Vorkommnissen derselben SSO. Falls beispielsweise der gleiche Datenbankzugriff an einer anderen Stelle im Quellcode mit identischen Zugriffskontrollen erfolgt, werden dieselben Annotationen hier automatisch ergänzt

und der Entwickler wird zur manuellen Kontrolle dieser Ergänzung informiert. Wenn eine identische SSO stattdessen mit anderen Zugriffskontrollen annotiert wurde, wird der Entwickler über diese Inkonsistenz informiert und die fraglichen Stellen werden als potenzielle Verwundbarkeiten markiert. Zusätzlich werden alle Pfade von den Einstiegspunkten des Programmcodes zu der jeweiligen SSO anhand einer statischen Analyse überprüft, um sicherzustellen, dass die annotierten Zugriffskontrollen jeden möglichen Pfad abdecken.

Beispiel
Zhu, Chu, Lipford und Thomas (2015, Seite 202) veranschaulichen die Idee dieses Ansatzes an einem Beispiel des Open Source Projektes Moodle im Bezug auf die Funktion `chat_login_user()`, welche sicherheitsrelevante Datenbankzugriffe durchführt. Gemäß des Konzeptes von ASIDE muss diese Funktion zunächst für das Plugin als SSO angegeben werden. Da ASIDE den Quellcode während der Implementierung analysiert und über die Sicherheitsrelevanz dieser Funktion informiert ist, benachrichtigt das Plugin den Entwickler über jeden Aufruf dieser Funktion anhand einer Warnung in der IDE. Abbildung 5.1 stellt die verschiedenen Interaktionsmöglichkeiten mit einer solchen Warnung dar.

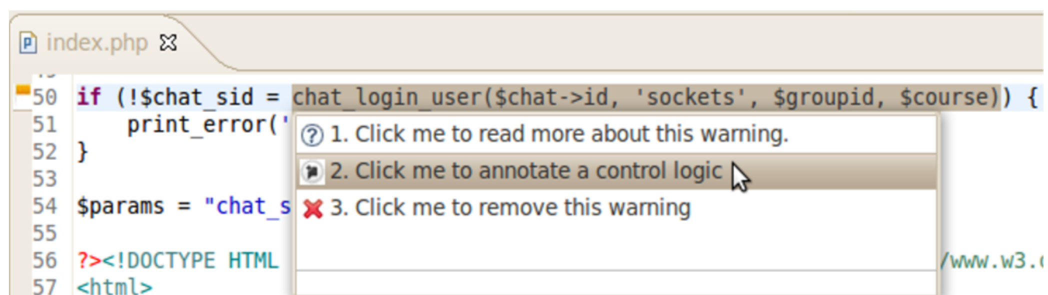

Abbildung 5.1: Das Kontextmenü für eine durch ASIDE generierte Warnung. Abbildung entnommen aus Zhu, Chu, Lipford und Thomas (2015, Seite 202).

Die *interaktive Annotierung* anhand der zweiten Option ist eines der Kernelemente dieses Ansatzes und ermöglicht es dem Entwickler in diesem Beispiel den Quellcode zu markieren, welcher die notwendige Zugriffskontrolllogik definiert, um sicherzustellen, dass der Benutzer seiner Software dazu berechtigt ist, die mit der Funktion `chat_login_user()` verbundenen Datenbankzugriffe durchzuführen. Abbildung 5.2 stellt die durch den Entwickler grün annotierten Zugriffskontrollen für die automatisch erkannte Verwendung der SSO in Zeile 50 dar.

Durch eine zusätzliche statische Code-Analyse auf Basis der durch diese grafischen Annotationen definierten Zusammenhänge im Quellcode stellt ASIDE zudem fest, dass der Entwickler bei einem Aufruf der Funktion `chat_login_user()` an einer anderen Stelle im Quellcode andere Zugriffskontrollen mit Annotationen versehen hat. ASIDE weist auf diese Inkonsistenz hin, indem der Funktionsaufruf in Zeile 50 als potenzielle Verwundbarkeit rot markiert wird.

Abbildung 5.2: Darstellung der grafischen Annotationen für eine potenzielle Verwundbarkeit in ASIDE. Abbildung entnommen aus Zhu, Chu, Lipford und Thomas (2015, Seite 202).

5.2.2 CQUAL

Ansatz

CQUAL setzt einen nicht-traditionellen Ansatz zur statischen Code-Analyse um, der dem Entwickler dabei hilft, Fehler in C-Quellcode zu finden (Foster, 2007). Die Programmiersprache C stellt standardmäßig einige Datentyp-Qualifikatoren zur Verfügung, die genutzt werden, um zusätzliche Informationen über eine Variable anzugeben, wie beispielsweise `const` oder `volatile`. Der Ansatz von CQUAL basiert auf der Definition zusätzlicher Qualifikatoren und ihrer Beziehungen durch den Benutzer. Diese Annotationen können entweder direkt im Code oder in einer *Prelude*-Datei definiert werden. Eine solche Datei, die einige Standardfunktionen von C mit Annotationen versieht, wird grundsätzlich mit CQUAL ausgeliefert. Abgesehen davon stellt das Werkzeug jedoch keine weiteren Annahmen über den Quellcode an und weitere Annotationen müssen durch den Entwickler selbst definiert werden.

Die definierten Annotationen können dann im Quellcode angegeben werden, damit CQUAL sie im Rahmen einer statischen Code-Analyse verwendet, um eine Datentyp-Qualifikator-Inferenz durchzuführen (Greenfieldboyce & Foster, 2004, Seite 2). Diese traversiert den Quellcode des Programms, wobei nicht-markierten Funktionen und Variablen die passenden Qualifikatoren automatisch zugeordnet und die Beziehungen zwischen den verschiedenen Identifikatoren auf Basis der Qualifikatoren evaluiert werden. Falls CQUAL während dieser Analyse einen Widerspruch in den zuvor definierten Beziehungen erkennt, informiert das Werkzeug den Entwickler über einen möglichen Fehler. Prinzipiell handelt es sich bei CQUAL somit um einen leichtge-

wichtigen, interaktiven Ansatz, dessen grundlegendes Prinzip dem der Regeln herkömmlicher SCATs entspricht.

Beispiel
Listing 5.1 definiert die Qualifikatoren `$untainted` und `$tainted` zur Markierung von nicht vertrauenswürdigen und vertrauenswürdigen Daten. In Zeile fünf wird ihre Beziehung zueinander festgelegt, die besagt, dass `$untainted` ein Subtyp von `$tainted` ist (Greenfieldboyce & Foster, 2004, Seite 2). Deshalb dürfen die als `$untainted` markierten Daten an allen Stellen im Quellcode verwendet werden, an denen als `$tainted` markierte Daten akzeptiert werden. Die als `$tainted` markierten Daten dürfen dementsprechend nur dort verwendet werden, wo sie explizit erlaubt sind.

```
1 partial order {
2   $untainted [level=value, sign=neg]
3   $tainted [level=value, sign=pos]
4
5   $untainted < $tainted
6 }
```

Listing 5.1: Definition von Qualifikatoren in CQUAL

Darauffolgend muss der Benutzer die Signaturen von Funktionen anhand der Qualifikatoren annotieren, um festzulegen, ob diese vertrauenswürdige oder nicht vertrauenswürdige Daten produzieren oder akzeptieren, siehe Listing 5.2. In diesem Beispiel wird CQUAL darüber informiert, dass der Rückgabewert der Funktion `getenv()` als nicht vertrauenswürdig anzusehen ist und dass die Funktion `printf()` ausschließlich vertrauenswürdige Daten als erstes Argument erhalten darf.

```
1 $tainted char *getenv(const char *name);
2 int printf($untainted const char *fmt, ...);
```

Listing 5.2: Annotieren von Signaturen in CQUAL

Bei der Analyse des in Listing 5.3 dargestellten Quellcodes erkennt CQUAL auf der Basis einer statischen Code-Analyse, dass die Variable `s` den Rückgabewert der Funktion `getenv()` erhält und ihr Wert somit nicht vertrauenswürdig ist. Der Wert der Variable `t` ist ebenfalls nicht vertrauenswürdig, da ihr der Wert von `s` zugewiesen wird. In Zeile sechs erkennt CQUAL einen Fehler, da die Funktion `printf()` einen vertrauenswürdigen Wert als Argument erwartet, aber mit `t` einen nicht vertrauenswürdigen Wert erhält und dies der zuvor definierten Beziehung widerspricht.

```
1 int main(void)
2 {
3   char *s, *t;
4   s = getenv("LD_LIBRARY_PATH");
5   t = s;
6   printf(t);
7 }
```

Listing 5.3: Beispiel für die Verwendung von CQUAL in Quellcode

Ähnliche Ansätze
Mit dem Checker Framework[1] existiert ein ähnlicher Ansatz für Java, welcher durch Annotationen definierte Einschränkungen anhand von Compiler Plugins (*Checkers*) überprüft und um neue Annotationen und Checker ergänzt werden kann. Mit JQUAL gibt es zudem Variante von CQUAL für Java (Greenfieldboyce & Foster, 2007).

5.2.3 SpotBugs Annotationen

Ansatz
Neben seiner Funktionalität als traditionelles SCAT stellt SpotBugs eine Reihe von Annotationen für Java bereit, die durch das Werkzeug ausgewertet werden können[2]. Anders als in CQUAL können neue Annotationen hier nicht durch den Benutzer definiert werden. Sie stellen auch nicht die Grundlage für eine durchzuführende Analyse dar, sondern ermöglichen es, die Intention des Entwicklers für das Werkzeug verständlich zu beschreiben. Beispielsweise indem ihm zusätzliche Informationen mitgeteilt werden, die durch eine vollautomatische Analyse nicht erkenntlich werden können oder um die Analyse zu präzisieren, indem etwa False Positives unterdrückt werden.

Beispiel
Einige der in SpotBugs enthaltenen Annotationen eignen sich auch für die Unterstützung bei der Anwendung von Secure Coding Richtlinien. Im Folgenden wird ein Beispiel für die Durchsetzung der Richtlinie *MSC59-J. Limit the lifetime of sensitive data*[3] beschrieben. Listing 5.4 enthält den hierfür relevanten Quellcode.

```
1  @CleanupObligation
2  public class Program {
3    static char[] c = new char[14];
4
5    public static void main(String args[]) {
6      generateSensitiveData();
7      // use sensitive data...
8      cleanUp();
9    }
10
11   @CreatesObligation
12   static void generateSensitiveData() {
13     c = "password123456".toCharArray();
14   }
15
16   @DischargesObligation
17   static void cleanUp() {
18     Arrays.fill(c, ' ');
19   }
20 }
```

Listing 5.4: Beispiel für die Verwendung von Annotationen in SpotBugs

[1] https://checkerframework.org/, aufgerufen am 26.02.2020
[2] https://spotbugs.readthedocs.io/en/latest/annotations.html, aufgerufen am 04.01.2020
[3] https://wiki.sei.cmu.edu/confluence/display/java/MSC59-J.+Limit+the+lifetime+of+sensitive+data, aufgerufen am 04.01.2020

In diesem Beispiel wird anhand der Annotation `@CreatesObligation` festgelegt, dass die Funktion `generateSensitiveData()` hier eine Ressource erstellt, die nach ihrer Verwendung wieder zu entfernen ist, damit der Zeitraum, in dem sie aus dem Speicher ausgelesen werden kann, möglichst kurz gehalten wird. Die Annotation `@DischargesObligation` teilt SpotBugs mit, dass die Funktion `cleanUp()` für die Entfernung einer Ressource zuständig ist. Für die Verwendung dieser Annotationen muss die Klasse zusätzlich mit `@CleanupObligation` markiert werden. Falls die Funktion `cleanUp()` in dem Quellcode nicht aufgerufen wird, gibt die Analyse von SpotBugs die in Abbildung 5.3 dargestellte Warnung aus, um den Entwickler daran zu erinnern, den Funktionsaufruf zu ergänzen[4].

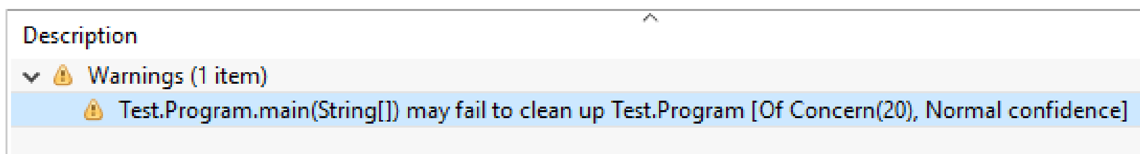

Abbildung 5.3: Beispiel für eine durch die Verwendung von Annotationen in SpotBugs generierte Warnung

Ähnlich wie bei vielen anderen SCATs stellt SpotBugs anhand der Annotation `@SuppressFBWarnings(value, justification)` zudem eine Möglichkeit bereit, um Fehler des Typs `value` unter Angabe einer optionalen Begründung zu unterdrücken, damit der Entwickler False Positives vermeiden kann. Da dieser Verwendungszweck von Annotationen jedoch eher für eine Präzisierung der Analyse genutzt wird, als um dem Entwickler bei der Vermeidung von Schwachstellen und Verwundbarkeiten zu helfen, wird dieser Verwendungszweck im Folgenden nicht näher betrachtet.

Ähnliche Ansätze

Textuelle Annotationen zur Unterstützung von SCATs werden auch durch einige andere Systeme bereitgestellt oder genutzt. Im Rahmen der Spezifikation JSR 305[5] wurde beispielsweise die Entwicklung einiger standardisierter Annotationen für Java vorgeschlagen, die durch SCATs verwendet werden könnten, um bei der Erkennung von Taint Propagation Problemen wie SQL-Injektionen zu helfen oder um Aussagen darüber zu treffen, ob eine bestimmte Variable den Wert `NULL` annehmen darf. Einige vergleichbare Annotationen zu diesem Zweck werden auch standardmäßig mit der IDE IntelliJ IDEA ausgeliefert[6]. Flanagan et al. (2002) sowie Fähndrich und Leino (2003) beschreiben ähnliche Ansätze. Die von Microsoft bereitgestellte Annotationssprache SAL[7] stellt ebenfalls Annotationen für C und C++ bereit, um die Einhaltung bestimmter Einschränkungen verifizieren zu können.

[4]SpotBugs kann als eigenständiges Programm oder als Plugin verwendet werden. In den hier dargestellten Abbildungen wird das SpotBugs Plugin für die IDE Eclipse verwendet.
[5]https://jcp.org/en/jsr/detail?id=305, aufgerufen am 26.02.2020
[6]https://www.jetbrains.com/help/idea/annotating-source-code.html, aufgerufen am 26.02.2020
[7]https://docs.microsoft.com/de-de/cpp/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects?view=vs-2019, aufgerufen am 26.02.2020

5.2.4 Composable Thread Coloring

Ansatz

Mit „Composable Thread Coloring“ stellen Sutherland und Scherlis (2010) einen Ansatz vor, um basierend auf einer statischen Code-Analyse Fehler bei der Verwendung von mehreren Threads in Java zu erkennen. Dieser Ansatz stellt gewissermaßen eine Kombination der Prinzipien von CQUAL und der zuvor beschriebenen Annotationen in SpotBugs dar, da hierbei Annotationen verwendet werden, welche zum einen die Grundlage für eine durchzuführende Analyse darstellen und zum anderen Informationen für ein SCAT bereitstellen, die automatisch nicht erkannt werden können.

Das grundlegende Prinzip dieses Ansatzes basiert auf der Verwendung einer Annotationssprache, um bestimmte Anforderungen für Threads auszudrücken. Eine statische Analyse stellt anschließend fest, ob die Implementierung der durch die Annotationen ausgedrückten Anforderungen an die Threads entspricht. Falls dies nicht der Fall ist, erhält der Entwickler ähnlich wie bei traditionellen SCATs eine Beschreibung des potenziellen Fehlers sowie Hinweise zur Behebung in textueller Form.

Beispiel

Listing 5.5 zeigt ein stark vereinfachtes Beispiel für die Verwendung dieses Ansatzes basierend auf dem von Sutherland und Scherlis (2010, Seite 236) beschriebenen Beispiel. In Zeile zwei werden mit DBChanger und DBExaminer zwei Rollen („Colors“) definiert, die Threads annehmen können. Zeile drei legt die Einschränkung fest, dass diese beiden Rollen nicht gleichzeitig durch einen Thread angenommen werden dürfen, während Zeile vier die Einschränkung definiert, dass lediglich ein Thread zu jedem Zeitpunkt die Rolle DBChanger übernehmen darf.

Eine statische Analyse des Quellcodes erkennt zudem anhand der Annotation in Zeile sieben, dass diese Methode nur durch einen Thread aufgerufen werden darf, welchem zuvor die Rolle DBChanger oder DBExaminer zugewiesen wurde und kann damit bereits Fälle identifizieren, in denen die Methode durch einen Thread ohne eine der beiden Rollen aufgerufen wird. Die Annotationen in den Zeilen neun und zehn besagen weiterhin, dass dem jeweiligen Thread an dieser Stelle die Rolle DBExaminer abgenommen und die Rolle DBChanger hinzugefügt wird. Diese Annotationen werden im Rahmen einer statischen Analyse genutzt, um Inkonsistenzen zu erkennen, falls ein Thread zwei inkompatible Rollen erhält, wie zuvor in Zeile drei definiert.

```
1 /**
2  * @ColorDeclare DBChanger, DBExaminer
3  * @IncompatibleColors DBChanger, DBExaminer
4  * @MaxColorCount DBChanger 1
5  */
6 public abstract class Job implements ..., Runnable {
7   @Color("DBChanger | DBExaminer")
8   public void run() {
9     // @Revoke DBExaminer
10    // @Grant DBChanger
11    accessDB();
12  }
13 }
```

Listing 5.5: Quellcodebeispiel für Composable Thread Coloring

5.2.5 SpongeBugs

Ansatz

Bei SpongeBugs handelt es sich um einen Ansatz, der dem Entwickler bei der Behebung einiger ausgewählter Fehlertypen in Java hilft, welche durch die SCATs SpotBugs und SonarQube gefunden werden können (Marcilio, Furia, Bonifácio & Pinto, 2019). Bei seiner Verwendung ergänzt SpongeBugs das Programm an den fehlerhaften Stellen um einen korrigierten Vorschlag, sodass der Entwickler seinen eigenen, fehlerhaften Quellcode durch den vollautomatisch generierten Code ersetzen kann. SpongeBugs ist in der Lage, Korrekturen für elf verschiedene Fehlertypen zu generieren. Diese Fehlertypen sind ausschließlich syntaktischer Natur, sodass eine Korrektur vollautomatisch generiert werden kann, ohne die Verhaltensweise des Programmes zu verändern, weshalb der Fokus hier in erster Linie auf Code-Smells wie beispielsweise der Regel *String literals should not be duplicated*[8] liegt (Marcilio, Furia et al., 2019, Seite 4). Dieses Prinzip entspricht den Quick-Fixes, die durch viele moderne IDEs angeboten werden.

Beispiel

Einer der Fehler, die SpongeBugs vollautomatisch korrigieren kann, basiert auf der Regel *Collections.EMPTY_LIST, EMPTY_MAP, and EMPTY_SET should not be used*[9], welche festlegt, dass generischen Collections keine Raw-Types anhand der Felder `EMPTY_LIST`, `EMPTY_MAP` oder `EMPTY_SET` zugewiesen werden dürfen, da dies nicht typsicher ist. Stattdessen sind die Methoden `emptyList()`, `emptyMap()` und `emptySet()` zu verwenden. Da es sich hierbei um eine Korrektur handelt, welche die Logik des Programmes nicht abändert, ist dies einer der Fehlertypen, welche durch SpongeBugs korrigiert werden. Abbildung 5.4 zeigt den fehlerhaften rot markierten Teil im Quellcode, sowie die automatische Korrektur durch SpongeBugs in blau.

```
public Collection<Binding> getSequencesFor(ParameterizedCommand
     ↪ command) {
   ArrayList<Binding> triggers = bindingsByCommand.get(command);
- return (Collection<Binding>) (triggers == null ?
     ↪ Collections.EMPTY_LIST : triggers.clone());
+ return (Collection<Binding>) (triggers == null ?
     ↪ Collections.emptyList() : triggers.clone());
}
```

Abbildung 5.4: Beispiel für eine vollautomatische Korrektur durch SpongeBugs. Abbildung entnommen aus Marcilio, Furia, Bonifácio und Pinto (2019, Seite 7).

Ähnliche Ansätze

Bei WalkMod[10] handelt es sich um ein Werkzeug, das ebenfalls die automatisierte Anwendung von Quick Fixes für SCATs wie PMD, CheckStyle oder SonarQube in Java ermöglicht.

[8] https://rules.sonarsource.com/java/RSPEC-1192, aufgerufen am 26.02.2020
[9] https://rules.sonarsource.com/java/RSPEC-1596, aufgerufen am 26.02.2020
[10] http://walkmod.com/, aufgerufen am 27.02.2020

5.2.6 FixBugs

Ansatz

Im Gegensatz zu der vollautomatischen Korrektur durch SpongeBugs stellen Barik, Song, Johnson und Murphy-Hill (2016) mit FixBugs einen Ansatz zur interaktiven Behebung komplexerer Fehler in Java vor, die sie als *Slow-Fixes* bezeichnen. Dies ist insbesondere hilfreich, um Probleme zu beheben, die auch von der Intention des Entwicklers abhängig sein können oder deren Lösung sich auf die Funktionsweise der Software auswirkt. Der Prototyp dieses Ansatzes ist als Ergänzung des SCATs FindBugs, des Vorgängers von SpotBugs, für die IDE Eclipse implementiert und kann verwendet werden, um Fehler im Hinblick auf das Auffangen von Exceptions, das Konkatenieren von Strings und das fehlende Schließen eines Streams zu beheben.

Beispiel

Ein Fehlertyp, der von FindBugs erkannt werden kann, ist das Auffangen zu allgemeiner statt spezifischer Exceptions[11]. Für die Behebung dieses Fehlers setzt FixBugs einen interaktiven Ansatz um, bei dem neben der Möglichkeit zur Verwendung einer standardmäßigen Lösung für das Problem nach dem Prinzip eines Quick-Fixes auch eine Drag-and-Drop-Funktionalität bereitgestellt wird, die es dem Entwickler ermöglicht, die jeweils geworfenen Exceptions an festgelegten Stellen im Quellcode wie catch-Blöcken oder der Signatur der Methode abzulegen, woraufhin der Quellcode automatisch angepasst wird. Abbildung 5.5 veranschaulicht den Drag-and-Drop-Prozess. Relevante Stellen im Quellcode werden hierbei farblich markiert, sodass ersichtlich wird, welcher Funktionsaufruf welche Exceptions werfen kann (grün und blau), sowie an welchen Stellen eine Exception per Drag-and-Drop abgelegt werden kann (grau).

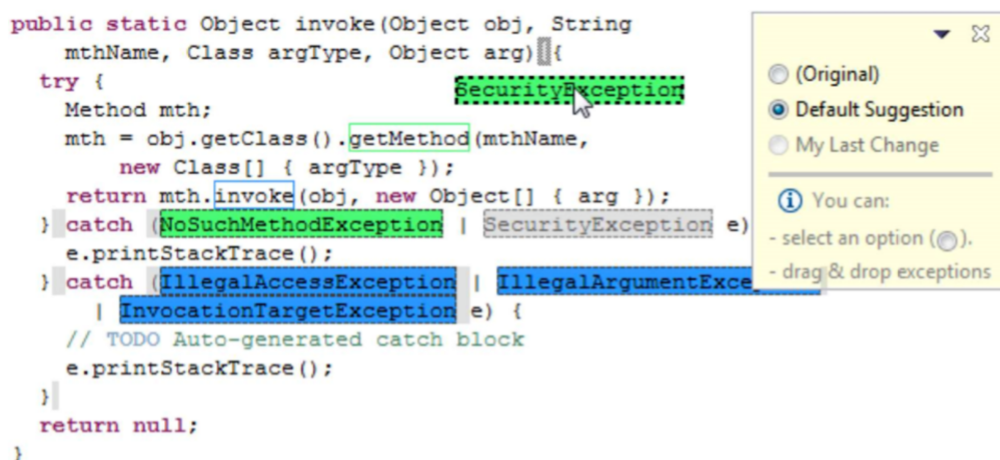

Abbildung 5.5: Beispiel für eine interaktive Korrektur für Exception-Handling anhand von FixBugs. Abbildung entnommen aus Barik, Song, Johnson und Murphy-Hill (2016, Seite 2).

[11] http://findbugs.sourceforge.net/bugDescriptions.html#REC_CATCH_EXCEPTION, aufgerufen am 26.02.2020

Für Fehler im Hinblick auf das ineffiziente Konkatenieren von Strings innerhalb einer Schleife und das Schließen eines Streams bietet FixBugs lediglich eine Auswahl zwischen verschiedenen automatischen Korrekturen des Quellcodes an, ohne die Verwendung eines Drag-and-Drop-Systems. Abbildung 5.6 zeigt die verschiedenen Möglichkeiten, die FixBugs bietet, um sicherzustellen, dass ein Stream auch im Falle einer Exception geschlossen wird. Der grün markierte Quellcode wird hierbei automatisch ergänzt, um den gelb hinterlegten Stream auch im Fehlerfall zu schließen.

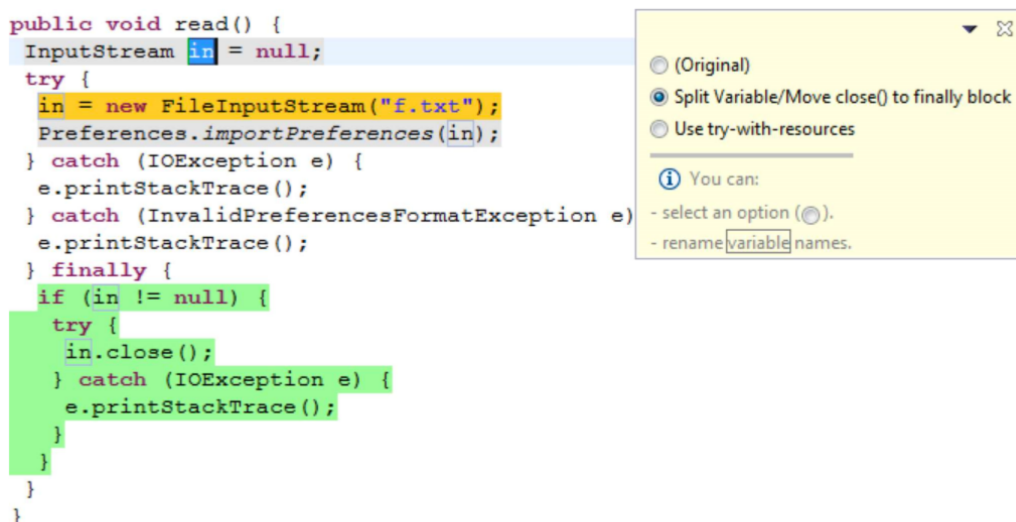

Abbildung 5.6: Beispiel für eine interaktive Korrektur für Streams anhand von FixBugs. Abbildung entnommen aus Barik, Song, Johnson und Murphy-Hill (2016, Seite 4).

Ähnliche Ansätze

Mit „Active Code Completion“ beschreiben Omar, Yoon, LaToza und Myers (2012) einen Ansatz für die Unterstützung bei der Instanziierung von Objekten in Java durch eine Erweiterung der IDE Eclipse, der ebenfalls je nach Problemstellung einen hochgradig individuellen Lösungsmechanismus definiert. Die Instanziierung der Klasse `Color` wird beispielweise durch ein Pop-up unterstützt, in dem der Entwickler die zu erstellende Farbe anhand einer Palette auswählen kann, während bei der Instanziierung eines regulären Ausdrucks Testfälle angegeben werden können, die der Ausdruck akzeptieren muss oder nicht akzeptieren darf, um seine Korrektheit zu prüfen.

5.2.7 CogniCrypt

Ansatz

Bei CogniCrypt[12] handelt es sich um Plugin für die IDE Eclipse, das auf die Unterstützung bei der Verwendung von kryptographischen APIs für Java, wie beispielsweise

[12]https://www.eclipse.org/cognicrypt/, aufgerufen am 31.12.2019

der Java Cryptography Architecture (JCA)[13] oder der Bouncy Castle Crypto APIs[14], spezialisiert ist[15]. CogniCrypt besteht zu diesem Zweck aus zwei Komponenten:

1. **Statische Code-Analyse:** Sobald der Entwickler seinen Quellcode speichert, wird eine statische Code-Analyse durchgeführt, welche den Code im Hinblick auf die Verwendung von kryptographischen APIs betrachtet. Falls hierbei eine potenzielle Verwundbarkeit festgestellt wird, erzeugt das Plugin eine Fehlermeldung, die den Entwickler auf das Problem hinweist. Dies entspricht der Vorgehensweise traditioneller SCATs.

2. **Codegenerierung:** In der Form eines Assistenten unterstützt CogniCrypt den Programmierer anhand von abstrahierten Fragen hinsichtlich des vorliegenden Anwendungsfalles bei der Auswahl eines adäquaten Kryptographiealgorithmus. Anschließend generiert CogniCrypt den nötigen Quellcode sowie ein Beispiel für dessen Anwendung automatisch.

Beispiel
Abbildung 5.7 veranschaulicht die Hilfestellung, die CogniCrypt bei einer fehlerhaften Nutzung einer kryptographischen API anhand einer vollautomatischen statischen Analyse des Programms bietet. In diesem Fall wurde die kryptographische Hashfunktion MD5 verwendet. Da MD5 nicht mehr als kollisionsresistent gilt[16], weist CogniCrypt hier darauf hin, dass eine andere Hashfunktion verwendet werden sollte.

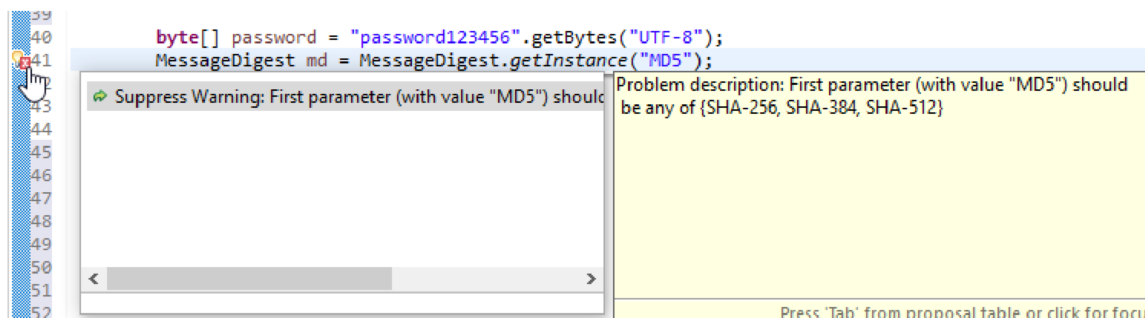

Abbildung 5.7: CogniCrypt markiert fehlerhafte Verwendungen von kryptographischen APIs in der IDE und gibt einen kurzen Hinweis zur Korrektur

Um den Quellcode für eine bestimmte Aufgabe durch CogniCrypt automatisch generieren zu lassen, muss der Entwickler einen Assistenten aufrufen und dort zunächst den jeweiligen Verwendungszweck angeben, siehe Abbildung 5.8. Nach der Auswahl einer Aufgabe, beispielsweise „Encrypting Data“, stellt der Assistent dem Benutzer weitere problembezogene Fragen, wie beispielsweise welche Art von Daten verschlüsselt werden soll, sowie ob und auf welchem Wege ein Schlüsselaustausch stattfinden soll, siehe Abbildung 5.9.

[13] https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html, aufgerufen am 31.12.2019
[14] https://www.bouncycastle.org/, aufgerufen am 31.12.2019
[15] https://projects.eclipse.org/projects/technology.cognicrypt, aufgerufen am 31.12.2019
[16] https://www.mscs.dal.ca/~selinger/md5collision/, aufgerufen am 28.01.2020

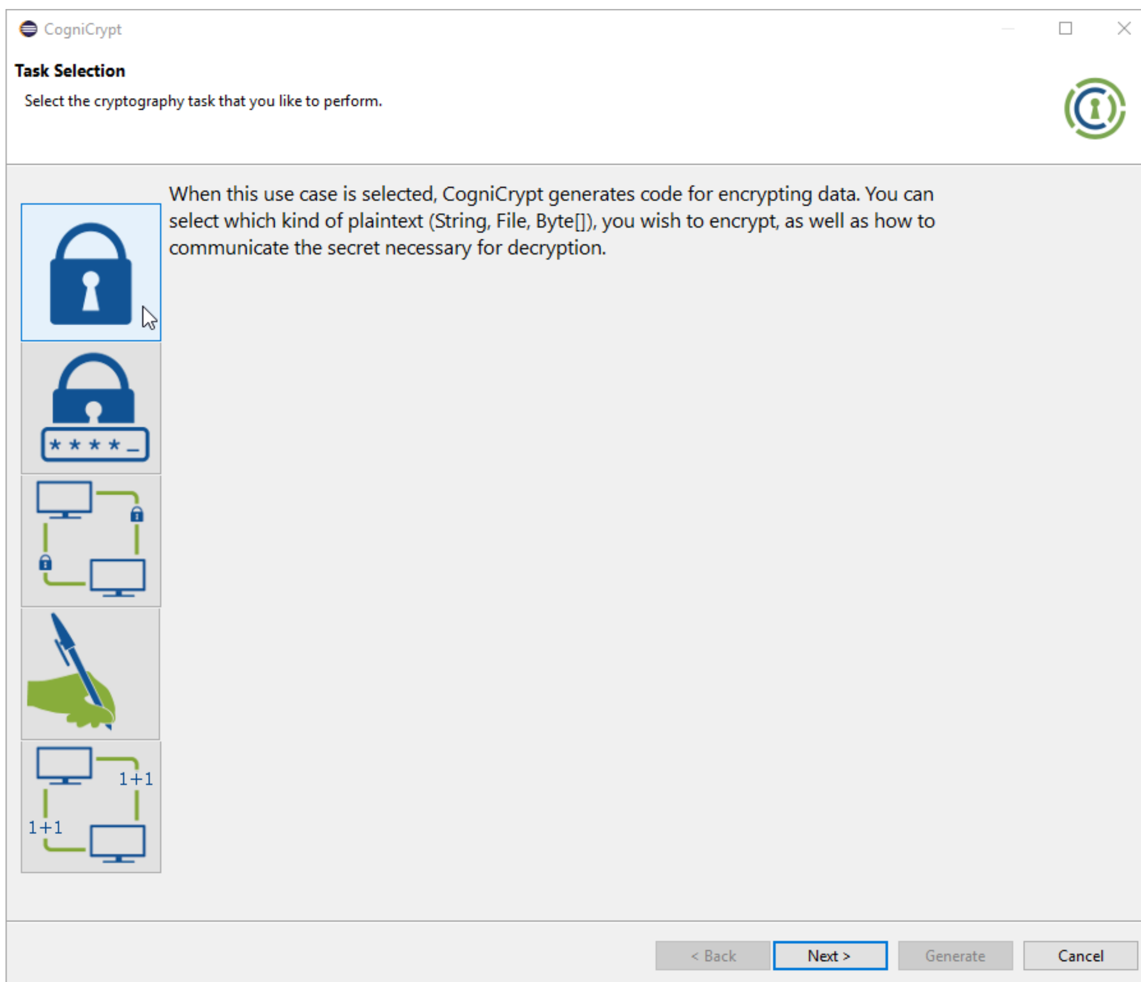

Abbildung 5.8: Die Auswahl der Aufgabe in dem Assistenten zur Codegenerierung in CogniCrypt

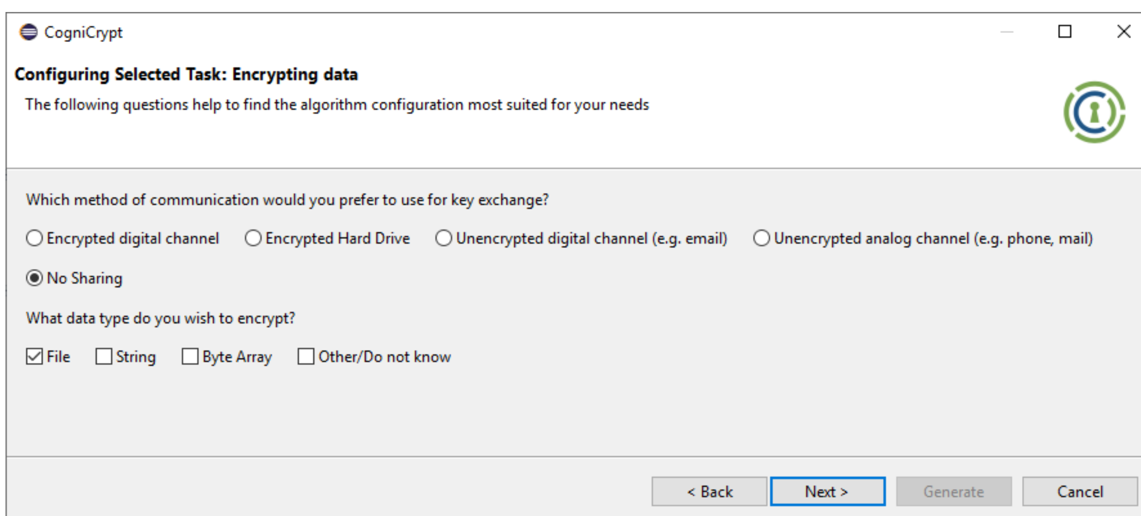

Abbildung 5.9: Die Fragen des Assistenten zur Codegenerierung in CogniCrypt im Hinblick auf die Verschlüsselung von Daten

5.2.8 Beobachtungen

Durch die Betrachtung der bereits existierenden nicht-traditionellen Ansätze werden verschiedene Beobachtungen deutlich. In erster Linie wird schnell erkenntlich, dass diese Ansätze im Wesentlichen zwei Ziele verfolgen: ASIDE, CQUAL, die durch SpotBugs bereitgestellten Annotationen und Composable Thread Coloring beschreiben nicht-traditionelle Ansätze, um Fehler aufzufinden, indem sie auf unterschiedliche Art und Weise interaktiv sind. So kann der Entwickler Informationen bereitstellen, die durch eine vollautomatische statische Code-Analyse nicht ersichtlich werden können. Diese Informationen werden in den betrachteten Ansätzen entweder anhand von textuellen oder grafischen Annotationen festgehalten und können dann im Rahmen der statischen Analyse verwendet werden, um Aussagen über potenziell vorhandene Fehler in dem Programmcode zu treffen. Konkret etwa indem verifiziert wird, dass bestimmte Einschränkungen eingehalten werden oder indem Inkonsistenzen zwischen der Implementierung und dem gewollten Zustand erkannt werden.

Im Gegensatz dazu verfolgen Ansätze wie SpongeBugs, FixBugs oder CogniCrypt das Ziel, den Entwickler bei der Behebung oder Vermeidung von Problemen zu unterstützen, indem Teile des Quellcodes verändert oder generiert werden. Dies erfolgt entweder auf einem vollautomatischen Wege, wie bei SpongeBugs, oder interaktiv, wie in FixBugs und CogniCrypt. Hierbei wird schnell erkenntlich, dass vollautomatische Korrekturen sich lediglich für Probleme eignen, deren Behebung nicht in einer Änderung der Funktionsweise der Software resultiert. Interaktive Ansätze eignen sich stattdessen besser für komplexere Probleme.

Im Hinblick auf die erste Forschungsfrage kann somit zunächst festgehalten werden, dass nicht-traditionelle Ansätze für die Verwendung statischer Code-Analyse zur Unterstützung des Entwicklers bei der Anwendung von Secure Coding Richtlinien entweder interaktiv bei der Fehlerfindung oder vollautomatisch oder interaktiv bei der Fehlerbehebung helfen müssen.

5.3 Betrachtete Richtlinien

Um individuelle Ansätze für die Unterstützung des Entwicklers bei der Anwendung von Secure Coding Richtlinien entwickeln zu können, müssen einige exemplarisch zu betrachtende Richtlinien ausgewählt werden. Diesbezüglich werden in diesem Abschnitt zunächst unterschiedliche Kategorien von Schwachstellen und Verwundbarkeiten identifiziert und priorisiert. Weiterhin werden Kriterien aufgestellt, die zur Auswahl konkreter Richtlinien innerhalb dieser Themenfelder genutzt werden. Abschließend werden basierend auf der vorherigen Priorisierung eine möglichst relevante Kategorie sowie einige für diese Kategorie besonders wichtige Secure Coding Richtlinien anhand der aufgestellten Kriterien ausgewählt, die im weiteren Verlauf dieser Arbeit näher betrachtet werden.

5.3.1 Bildung relevanter Kategorien

Wie in Abschnitt 3.2 beschrieben existiert eine Reihe unterschiedlicher Quellen, die Richtlinien für die sichere Programmierung in Java beschreiben. Die Entwicklung neuer Ansätze für die Anwendung jeder dieser Richtlinien ist im Rahmen dieser Arbeit sowohl unrealistisch als auch nicht notwendig, da an dieser Stelle kein funktionsreiches

Werkzeug umgesetzt werden soll, das eine vollständige Lösung zur Durchsetzung der Softwaresicherheit während der Implementierung darstellt. Stattdessen sollen neue nicht-traditionelle Ansätze unter der Verwendung statischer Code-Analyse entwickelt werden, die dem Entwickler möglichst effektiv und effizient bei der Anwendung besonders relevanter Secure Coding Richtlinien zur Identifizierung, Vermeidung oder Behebung von Schwachstellen und Verwundbarkeiten helfen.

Eine zentrale Beobachtung für die Auswahl relevanter Secure Coding Richtlinien ist, dass eine recht geringe Menge an stets gleichen Sicherheitsproblemen mit vergleichsweise schwerwiegenden Folgen besonders häufig auftritt (Hovemeyer, Spacco & Pugh, 2005, Seite 1). Ein Bericht des Unternehmens Veracode zum Stand der Softwaresicherheit im Jahr 2019 hält beispielsweise fest, dass 24% der 85.000 untersuchten Anwendungen mindestens eine Verwundbarkeit beinhalten, die eine SQL-Injektion ermöglicht (Veracode, 2019, Seite 19). McGraw (2006, Seite 274) klassifiziert und priorisiert die verschiedenen Arten von Schwachstellen und Verwundbarkeiten als *Seven Pernicious Kingdoms* wie folgt:

1. Input Validation and Representation
2. API Abuse
3. Security Features
4. Time and State
5. Error Handling
6. Code Quality
7. Encapsulation

- Environment

Um zu prüfen, inwiefern diese Kategorisierung und Priorisierung der aktuellen Situation entspricht, werden die Seven Pernicious Kingdoms mit den aktullen Werten der *OWASP Top 10* von 2017[17] sowie denen der *CWE Top 25* von 2019[18] abgeglichen. Bei diesem Prozess werden zudem einige Unterkategorien innerhalb der durch die Seven Pernicious Kingdoms definierten Kategorien erkenntlich. Abbildung 5.10 stellt die Ergebnisse dieses Vergleichs dar.

Durch diesen Vergleich wird erkenntlich, dass die Kategorien „Input Validation and Representation“, „Security Features“, „Time and State“ sowie „Code Quality“ auch aktuell besonders relevante Themenfelder sind, zu denen teilweise auch weitere Unterkategorien gebildet werden können. Durch den Vergleich des in der jeweiligen Liste niedrigsten Ranges kann weiterhin eine Priorisierung dieser Kategorien und Unterkategorien gebildet werden. Tabelle 5.1 stellt diese Priorisierung dar.

[17] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, aufgerufen am 09.01.2020

[18] https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html, aufgerufen am 09.01.2020

CWE Top 25	OWASP Top 10	7 Pernicious Kingdoms	Unterkategorien
14. CWE-476 NULL Pointer Dereference	-		
20. CWE-400 Uncontrolled Resource Consumption	-		
21. CWE-772 Missing Release of Resource after Effective Lifetime	-		
-	A9 Using Components with Known Vulnerabilities		
7. CWE-416 Use After Free	-	6. Code Quality	
12. CWE-787 Out-of-bounds Write	-		
5. CWE-125 Out-of-bounds Read	-		
1. CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer	-		Speichersicherheit
8. CWE-190 Integer Overflow or Wraparound	-		
3. CWE-20 Improper Input Validation	-		
22. CWE-426 Untrusted Search Path	-		
17. CWE-611 Improper Restriction of XML External Entity Reference	A4 XML External Entities (XXE)		
2. CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	A7 Cross-Site Scripting (XSS)		
6. CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')			
11. CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')			
16. CWE-434 Unrestricted Upload of File with Dangerous Type			
18. CWE-94 Improper Control of Generation of Code ('Code Injection')	A1 Injection	1. Input Validation and Representation	Injektionsangriffe
4. CWE-200 Information Exposure			
25. CWE-295 Improper Certificate Validation	A3 Sensitive Data Exposure		Vertraulichkeit sensibler Daten
24. CWE-269 Improper Privilege Management			
10. CWE-22 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	A5 Broken Access Control		Fehlerhafte Zugriffskontrollen
15. CWE-732 Incorrect Permission Assignment for Critical Resource	A6 Security Misconfiguration		
9. CWE-352 Cross-Site Request Forgery (CSRF)	-	3. Security Features	
13. CWE-287 Improper Authentication			
19. CWE-798 Use of Hard-coded Credentials	A2 Broken Authentication	4. Time and State	
23. CWE-502 Deserialization of Untrusted Data	A8 Insecure Deserialization	-	
-	A10 Insufficient Logging & Monitoring	-	
-	-	2. API Abuse	
-	-	5. Errors	
-	-	7. Encapsulation	
-	-	*. Environment	

Abbildung 5.10: Zusammenhänge zwischen den in den Seven Pernicious Kingdoms, OWASP Top 10 und CWE Top 25 festgehaltenen Schwachstellen und Verwundbarkeiten

Priorität	*Kategorie*	*CWE Top25 Rang*	*OWASP Top10 Rang*	*Seven Pernicious Kingdoms Rang*
1	Input Validation and Representation	1	A1	1
	· Injektionsangriffe	6	A1	1
	· Speichersicherheit	1	-	1
2	Security Features	4	A3	3
	· Vertraulichkeit sensibler Daten	4	A3	3
	· Fehlerhafte Zugriffskontrollen	10	A5	3
3	Time and State	13	A2	4
4	Code Quality	7	A9	6

Tabelle 5.1: Priorisierung relevanter Kategorien und Unterkategorien für Schwachstellen und Verwundbarkeiten

5.3.2 Kriterien zur Auswahl der Richtlinien

Auf Basis der Erkenntnisse aus Abschnitt 3.2 sollen im Folgenden nur die Richtlinien des CERT Secure Coding Standards für Java betrachtet werden, insbesondere da dieser eine sehr vollständige Liste an spezifischen Richtlinien für Java bereitstellt, die neben einer Beschreibung des Fehlers auch Lösungsansätze bereitstellen. Der aktuelle Stand des CERT-Standards umfasst knapp 210 Regeln und 80 Empfehlungen. Aufgrund dieser großen Menge an Richtlinien ist eine weitere Priorisierung innerhalb der einzelnen Kategorien notwendig, um die Richtlinien herauszufiltern, deren Einhaltung als besonders wichtig gilt. Das CERT weist den Richtlinien zu diesem Zweck eine Priorität basierend auf drei Faktoren zu[19]:

1. **Schwere:** Wie schwerwiegend sind die Konsequenzen, falls die Richtlinie missachtet wird?

2. **Wahrscheinlichkeit:** Wie wahrscheinlich ist es, dass das Missachten der Richtlinie zu einer ausnutzbaren Verwundbarkeit führt?

3. **Anpassungskosten:** Wie aufwändig ist die Anpassung bereits geschriebenen Quellcodes, um die Einhaltung der Richtlinie im Nachhinein umzusetzen?

Diese Metrik ist aufgrund des Kriteriums *Anpassungskosten* nur im Bezug auf die nachträgliche Überarbeitung von Softwareprojekten anwendbar. Eine solche Unterscheidung ist für die hier zu entwickelnden Ansätze zur Unterstützung bei der Anwendung der Richtlinien zunächst jedoch nicht relevant, da es für die Entwicklung der Ansätze vorerst nicht von Bedeutung ist, ob sie während der ersten Programmierung oder einer nachträglichen Anpassung angewendet werden.

[19]https://wiki.sei.cmu.edu/confluence/display/java/Rule%3A+Priority+and+Levels, aufgerufen am 07.11.2019

Da die anderen Kriterien hier dennoch relevant sind, kann zunächst eine alternative Priorisierung ohne den Faktor *Anpassungskosten* aufgestellt werden, indem dieser bei der Berechnung der Priorität ausgelassen wird. Diese Abwandlung entspricht damit der Definition des Risikobegriffs und bildet so das erste Kriterium zur Auswahl der hier zu betrachtenden Richtlinien:

$$Risikofaktor = Schwere \cdot Wahrscheinlichkeit$$

Da die Richtlinien für die exemplarische Entwicklung nicht-traditioneller Ansätze zur Unterstützung bei dem Verfassen sicheren Quellcodes genutzt werden sollen, müssen einige weitere Kriterien für die Auswahl relevanter Richtlinien definiert werden:

1. **Allgemeingültigkeit:** Es sollen nur Richtlinien betrachtet werden, die für jede Art von Java-basierter Software und im weiteren Sinne auch für unterschiedliche Programmiersprachen gültig sind. Damit können beispielsweise die Richtlinien der Kategorie *Rule 50. Android (DRD)* ausgeschlossen werden, da diese sich mit den Spezifika der Softwareplattform Android auseinandersetzen.

2. **Risikofaktor:** Anhand des zuvor beschriebenen Risikofaktors wird eine Priorisierung durchgeführt, mittels derer die wichtigeren Richtlinien herausgestellt werden können. Dies setzt voraus, dass den jeweiligen Richtlinien eine Priorität durch das CERT zugewiesen ist. Richtlinien die sich beispielsweise noch im Aufbau befinden sind meist nicht mit einer Priorität versehen.

3. **Vollständigkeit:** Richtlinien, die als „überholt“ markiert sind oder sich noch im Aufbau befinden und somit unvollständig sind, sollen nicht behandelt werden. Somit wird verhindert, dass die entwickelten Ansätze in naher Zukunft obsolet werden, und es wird gewährleistet, dass die Unterstützung bei der Anwendung der Richtlinien zu einer möglichst korrekten Lösung führt.

4. **Thematische Nähe:** Bei thematisch ähnlichen Richtlinien soll nur eine der Richtlinien exemplarisch betrachtet werden.

5. **Codenähe:** Aufgrund des Zusammenhangs mit statischer Code-Analyse sollen nur Richtlinien ausgewählt werden, die relevant im Hinblick auf den Code sind. Richtlinien wie *ENV04-J. Do not disable bytecode verification*[20] oder *ENV00-J. Do not sign code that performs only unprivileged operations*[21] können damit beispielsweise ausgeschlossen werden.

5.3.3 Auswahl der Richtlinien

Durch die Bildung der Kategorien in Abschnitt 5.3.1 wird ersichtlich, dass die Validierung Eingaben die momentan wichtigste Kategorie von Schwachstellen und Verwundbarkeiten darstellt. Der Kodierungsstandard des CERT beschreibt verschiedene

[20] https://wiki.sei.cmu.edu/confluence/display/java/ENV04-J.+Do+not+disable+bytecode+verification, aufgerufen am 20.04.2020

[21] https://wiki.sei.cmu.edu/confluence/display/java/ENV00-J.+Do+not+sign+code+that+performs+only+unprivileged+operations, aufgerufen am 20.04.2020

Regeln und Empfehlungen für die Validierung von Eingaben in den Kategorien *Rule 00. Input Validation and Data Sanitization (IDS)* und *Rec. 00. Input Validation and Data Sanitization (IDS)*. Da bereits durch die Menge an Einträgen für die Thematik der Eingabevalidierung in den zuvor dargestellten Priorisierungen ersichtlich wird, dass es sich hierbei um ein komplexes und großes Themenfeld handelt, müssen an dieser Stelle einige konkrete Richtlinien ausgewählt werden, die exemplarisch näher zu betrachten sind. Weil die Regeln des CERT-Standards, wie in Abschnitt 3.1 beschrieben, grundsätzlich einzuhalten sind und ihre Missachtung in einem prüfbaren Defekt resultiert, werden im weiteren Verlaufe ausschließlich Regeln für die Validierung von Eingaben betrachtet. Empfehlungen werden hier nicht behandelt.

Der Standard umfasst 17 verschiedene Regeln, welche direkt in die Kategorie der Eingabevalidierung fallen[22]. Unter der Anwendung der in Abschnitt 5.3.2 definierten Kriterien wird Tabelle 5.2 für diese 17 Regeln aufgestellt, in der insbesondere ersichtlich wird, welche Regeln aufgrund ihres *Risikofaktors* besonders relevant sind und bei welchen das Kriterium *Vollständigkeit* verletzt wird. Die Kriterien *Allgemeingültigkeit* und *Codenähe* werden durch alle aufgeführten Regeln erfüllt, während das Kriterium *Thematische Nähe* nicht anwendbar ist, da alle aufgeführten Regeln relevante Eigenschaften aufweisen.

Regel	*Information*	*Risikofaktor*
IDS02-J	Überholt, siehe FIO16-J	-
IDS05-J	Überholt, siehe FIO99-J	-
IDS09-J	Überholt, siehe STR02-J	-
IDS10-J	Überholt, siehe STR01-J	-
IDS13-J	Überholt, siehe STR04-J	-
IDS15-J	Platzhalter	6
IDS00-J		6
IDS01-J		6
IDS07-J		6
IDS11-J		6
IDS14-J		6
IDS16-J		6
IDS03-J		4
IDS17-J		4
IDS04-J		2
IDS06-J		2
IDS08-J		2

Tabelle 5.2: Priorisierung von CERT-Richtlinien für die Thematik der Eingabevalidierung unter Beachtung der Auswahlkriterien

Damit werden basierend auf den aufgestellten Kriterien an dieser Stelle die sechs folgenden, besonders wichtigen Secure Coding Richtlinien aus dem Kodierungsstandard des CERT im Bezug auf die Validierung von Eingaben ausgewählt, die im weiteren Verlauf dieser Arbeit näher zu betrachten sind.

[22] https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487865, aufgerufen am 26.02.2020

- **IDS00-J.** Prevent SQL injection
- **IDS01-J.** Normalize strings before validating them
- **IDS07-J.** Sanitize untrusted data passed to the Runtime.exec() method
- **IDS11-J.** Perform any string modifications before validation
- **IDS14-J.** Do not trust the contents of hidden form fields
- **IDS16-J.** Prevent XML Injection

5.4 Definition des Schwerpunktes

Anhand der Erkenntnisse dieses Kapitels wird an dieser Stelle der Schwerpunkt der im Folgenden zu entwickelnden Ansätze definiert. Durch die Betrachtung existierender nicht-traditioneller Ansätze in Abschnitt 5.2 wurde insbesondere erkenntlich, dass diese jeweils mindestens eines von zwei grundlegenden Zielen verfolgen: Sie befassen sich entweder mit nicht-traditionellen Vorgehensweisen, um Fehler in dem Programmcode zu erkennen, oder mit Methoden, um Fehler zu beheben bzw. zu vermeiden.

Im Rahmen der Auswahl der näher zu betrachtenden Richtlinien in Abschnitt 5.3 wurde festgestellt, dass vier wesentliche Kategorien festgehalten werden können, die einen Großteil der aktuell besonders wichtigen Schwachstellen und Verwundbarkeiten umfassen. Das wichtigste dieser Themenfelder ist dabei die korrekte Validierung von Eingaben. Im Hinblick auf diese Thematik definiert der Kodierungsstandard des CERT für Java verschiedene Richtlinien, von denen sechs besonders relevante Regeln für die nähere Betrachtung im Rahmen dieser Arbeit ausgewählt wurden.

Basierend auf diesen Beobachtungen wird der Schwerpunkt auf die Entwicklung neuer nicht-traditioneller Ansätze gelegt, welche den Entwickler dabei unterstützen sollen, Fehler im Hinblick auf die Umsetzung von Eingabevalidierungen zu beheben bzw. zu vermeiden, konkret unter Beachtung der in Abschnitt 5.3.3 ausgewählten Secure Coding Regeln. Der Fokus wird hierbei nicht auf die Entwicklung neuer Ansätze für das Auffinden von Problemen im Hinblick auf Eingabevalidierungen gelegt, da dieser Bereich durch die weit verbreiteten Taint Propagation Analysen bereits gut erforscht ist, sowohl in einer vollautomatischen Form wie in traditionellen SCATs als auch in einer interaktiveren Form in nicht-traditionelle Ansätze wie etwa CQUAL oder dem Checker Framework. Methoden für die Korrektur bzw. Vermeidung entsprechender Fehler im Zusammenhang mit statischer Code-Analyse konnten hingegen nicht gefunden werden.

Kapitel 6

Entwicklung individueller Ansätze zur Anwendung von Richtlinien

Dieses Kapitel befasst sich mit der Entwicklung von neuen Ansätzen, welche auf die individuellen Eigenschaften ausgewählter Secure Coding Richtlinien für die Validierung von Eingaben zugeschnitten sind, um den Entwickler basierend auf statischer Code-Analyse bei ihrer Anwendung zu unterstützen.

6.1 Einleitung

Traditionelle SCATs sind primär auf das Auffinden von Fehlern in dem Code eines Programms ausgelegt. Im Hinblick auf die Vermeidung oder Behebung gefundener Fehler unterstützen sie den Entwickler üblicherweise nur anhand von Beschreibungen, die sich oft auch nicht auf den konkreten Code beziehen, sondern allgemeiner Natur sind. SpotBugs liefert beispielsweise sehr grobe Beschreibungen, siehe Abbildung 6.1, während SonarQube meist ausführlichere Erklärungen mit möglichen Lösungsansätzen angibt, wie durch Abbildung 6.2 veranschaulicht.

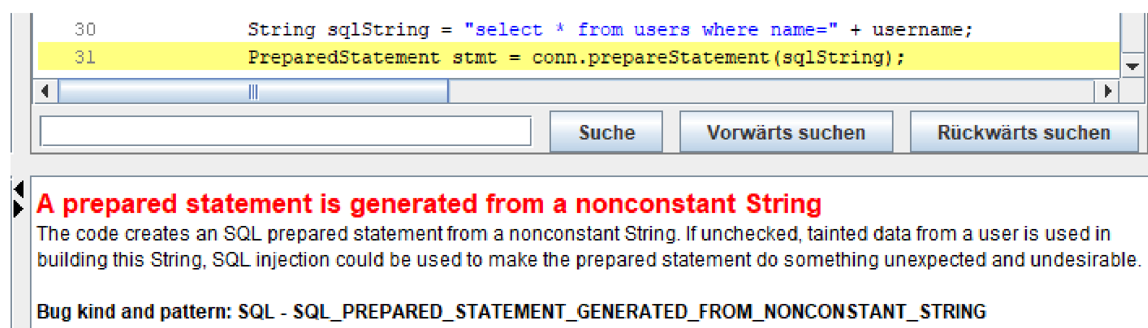

Abbildung 6.1: Beschreibung einer Verwundbarkeit für SQL-Injektionen in SpotBugs

Die Studie von Marcilio, Bonifacio et al. (2019, Seite 5) zeigt, dass Entwickler mit knapp 10% meist nur einen sehr geringen Anteil der berichteten Fehler korrigieren. Dies lässt sich auf einige zentrale Gründe zurückführen. Habib und Pradel (2018, Seite 324) führen beispielsweise an, dass bei der ersten Analyse durch ein SCAT je nach Umfang des analysierten Programms eine große Menge an potenziellen Fehlern berichtet werden kann, die nur schwer zu handhaben ist. Marcilio, Bonifacio et al.

What's the risk? | Are you at risk? | How can you fix it?

Recommended Secure Coding Practices

- Avoid building queries manually using formatting technics. If you do it anyway, do not include user input in this building process.
- Use parameterized queries, prepared statements, or stored procedures whenever possible.
- You may also use ORM frameworks such as Hibernate which, if used correctly, reduce injection risks.
- Avoid executing SQL queries containing unsafe input in stored procedures or functions.
- Sanitize every unsafe input.

You can also reduce the impact of an attack by using a database account with low privileges.

Compliant Solution

```
public User getUser(Connection con, String user) throws SQLException {

  Statement stmt1 = null;
  PreparedStatement pstmt = null;
  String query = "select FNAME, LNAME, SSN " +
                 "from USERS where UNAME=?"
  try {
    stmt1 = con.createStatement();
    ResultSet rs1 = stmt1.executeQuery("GETDATE()");

    pstmt = con.prepareStatement(query);
    pstmt.setString(1, user);  // Good; PreparedStatements escape their inputs.
```

Abbildung 6.2: Beschreibung einer Verwundbarkeit für SQL-Injektionen in SonarQube

(2019, Seite 10) merken zudem an, dass Entwickler die Fehler häufig nicht beheben, da sie unter Zeitdruck stehen oder weil sie sich nicht darüber im Klaren sind, wie der jeweilige Fehler zu korrigieren ist. Die Untersuchung von B. Johnson, Song, Murphy-Hill und Bowdidge (2013, Seite 6) stellt weiterhin fest, dass sich viele Entwickler zusätzlich zu der Funktionalität traditioneller SCATs für das Auffinden von Fehlern auch eine bessere Unterstützung bei der Behebung dieser Fehler wünschen, beispielsweise indem Lösungen vollautomatisch generiert werden, Beispiele für ein besseres Verständnis des Problems geliefert werden oder kontextbezogene Quellcodevorlagen zur Korrektur des Problems angeboten werden. Diese Unterstützung bei der Behebung von Problemen wird von den Entwicklern als ebenso wichtig angesehen, wie die Hilfe bei dem Auffinden der Probleme. Die Studien von Xie et al. (2011, Seite 164), Nadi, Krüger, Mezini und Bodden (2016, Seite 943) sowie Meng et al. (2017, Seite 10) kommen ebenfalls zu dem Ergebnis, dass eine bessere Unterstützung bei der Vermeidung und Behebung von Fehlern nötig ist.

Diese Erkenntnisse sowie die bereits zuvor genannte Beobachtung, dass eine große Anzahl von Schwachstellen und Verwundbarkeiten in der Praxis aus einer vergleichsweise kleinen Menge immer wiederkehrender Fehler resultiert, legen nahe, dass Ansätze, die SCATs ergänzen und individuell auf die Eigenschaften dieser jeweiligen Fehlertypen hin entwickelt werden, Entwicklern sehr effektiv bei der Behebung oder Vermeidung vieler Fehler helfen können. Um solche Ansätze entwickeln zu können

werden in Abschnitt 6.2 zunächst die Grundlagen der Thematik der Eingabevalidierung und in Abschnitt 6.3 die bereits existierenden Konzepte für die Umsetzung korrekter Eingabevalidierungen beschrieben. Im Anschluss daran wird in Abschnitt 6.4 die Motivation für die Wahl der Konzepte der neuen Ansätze erläutert und in den Abschnitten 6.5, 6.6 und 6.7 werden die hier betrachteten Ansätze für die Unterstützung des Entwicklers bei der Umsetzung korrekter Eingabevalidierung durch die Anwendung von Secure Coding Richtlinien und anhand von statischer Code-Analyse erläutert.

6.2 Eingabevalidierung

In diesem Abschnitt werden die Grundlagen der Eingabevalidierung beschrieben und am Beispiel der sechs zuvor ausgewählten Secure Coding Richtlinien veranschaulicht. Abschließend wird die Bedeutung der Beobachtungen dieses Abschnitts für die Entwicklung der neuen Ansätze zusammengefasst.

6.2.1 Grundlagen der Eingabevalidierung

Eingaben an ein Programm, die durch einen Benutzer oder andere nicht vertrauenswürdige Quellen kontrolliert werden können, sind grundsätzlich selbst als nicht vertrauenswürdig anzusehen. Diese Eingaben können an offensichtlichen Stellen in das Programm eintreten, etwa anhand eines `args`-Parameters einer Main-Methode, oder an weniger offensichtlichen Stellen, wie Cookies oder einem `GET`-Parameter in einer URL, dessen Veränderung durch den Anwender nicht vorgesehen ist[1]. Durch die Validierung dieser Eingaben ist sicherzustellen, dass sie wie erwartet durch das Programm verwertet werden können, ohne dabei in Nebenwirkungen zu resultieren. Dies kann erzielt werden, indem als ungültig erkannte Eingaben abgewiesen oder in einen gültigen Zustand transformiert werden.

Typische Verwundbarkeiten, die durch nicht vorhandene oder fehlerhafte Eingabevalidierungen entstehen können, sind Gelegenheiten für SQL-Injektionen, Cross-Site-Scripting Angriffe oder Buffer Overflows (McGraw, 2006, Seite 275). Fehlerhafte Eingabevalidierungen können somit in einer Gefährdung jedes der drei Schutzziele der Informationssicherheit resultieren. Ein Angreifer kann SQL-Injektionen beispielsweise verwenden, um sowohl die Vertraulichkeit, die Integrität als auch die Verfügbarkeit zu gefährden, indem er Zugriff auf Daten erlangt, für die er nicht berechtigt ist, Daten verändert oder löscht[2].

Die genauen Techniken, anhand derer Eingaben mit unterschiedlichen Verwendungszwecken und in unterschiedlichen Situationen zu validieren sind, sind nicht klar klassifiziert[3] und eine Ausarbeitung dieser Feinheiten würde den Rahmen dieser Arbeit sprengen. Stattdessen werden an dieser Stelle einige grundlegende Prinzipien beschrieben, die für die Eingabevalidierung besonders relevant sind und auch durch die sechs zuvor ausgewählten Secure Coding Richtlinien erkenntlich werden.

[1] https://cwe.mitre.org/data/definitions/472.html, aufgerufen am 28.02.2020
[2] https://cwe.mitre.org/data/definitions/20.html, aufgerufen am 28.02.2020
[3] https://cwe.mitre.org/data/definitions/20.html, aufgerufen am 28.02.2020

- Eine Eingabe sollte immer **syntaktisch** und **semantisch** validiert werden[4]. Eine syntaktische Validierung stellt sicher, dass die Syntax der Eingabe, beispielsweise eines Datums, korrekt ist, während eine semantische Validierung prüft, ob der Wert der Eingabe im Rahmen der Logik des Programms korrekt ist, etwa indem ein Programm für die Buchung eines Hotelzimmers sicherstellt, dass das Abreisedatum nicht vor dem Ankunftsdatum liegen kann.

- Die Art und Weise der Umsetzung des Validierungsprozesses ist stark von dem Verwendungszweck der Eingabe abhängig. Für die Vorgehensweise zur Prüfung, ob eine Zahl in einem bestimmten Bereich liegt, existiert beispielsweise keine Secure Coding Richtlinie und es genügt ein einfacher Vergleich der Eingabe mit den jeweiligen Minimal- und Maximalwerten. Für die Vermeidung von SQL-Injektionen definiert die Richtlinie *IDS00-J. Prevent SQL injection* hingegen feste Vorgehensweisen anhand der Funktionen der jeweils verwendeten Datenbankschnittstelle (z.B. Java Database Connectivity (JDBC), Java Persistence API (JPA) oder Hibernate). Im Folgenden bezeichnen daher **Eingaben mit einem allgemeinen Verwendungszweck** solche Eingaben, für deren standardmäßige Validierung keine Richtlinien existieren, während **Eingaben mit einem spezifischen Verwendungszweck** solche bezeichnen, deren Validierungsprozess entsprechend einer Richtlinie immer gleich ablaufen muss.

6.2.2 Eingabevalidierung am Beispiel von Richtlinien

An den sechs zuvor ausgewählten Secure Coding Richtlinien können einige für die Entwicklung der neuen Ansätze besonders relevante Eigenschaften, sowie die Prinzipien der Eingabevalidierung verdeutlicht werden. Zu diesem Zweck werden die ausgewählten Richtlinien an dieser Stelle grob beschrieben. Weitere relevante Details zu diesen Richtlinien werden im Rahmen der Beschreibung der entwickelten Ansätze erwähnt.

IDS00-J. Prevent SQL injection
Diese Richtlinie befasst sich mit dem spezifischen Verwendungszweck der Nutzung einer Eingabe in der Konstruktion einer SQL-Abfrage. Im Falle einer fehlerhaften oder nicht vorhandenen Validierung einer solchen Eingabe kann ein Angreifer einen schädlichen Wert angeben, um beispielsweise Informationen aus einer Datenbank zu löschen. Um eine solche Verwundbarkeit zu vermeiden legt der CERT-Standard fest, dass spezielle Funktionalitäten der jeweils verwendeten Datenbankschnittstelle für die Konstruktion der SQL-Abfrage zu verwenden sind, die SQL-Injektionen unterbinden.

IDS01-J. Normalize strings before validating them
Ob ein String mit einem allgemeinen Verwendungszweck zulässig ist, wird oft anhand eines regulären Ausdrucks geprüft. Erfüllt der String das durch den Ausdruck definierte Muster, kann er verwendet werden. Falls nicht, muss der String entweder in einen zulässigen Zustand transformiert oder zurückgewiesen werden. Da Zeicheninformationen in Java auf dem Unicode Standard basieren, existieren unterschiedliche Zeichenfolgen, die identisch interpretiert werden können. Der Unicode-Text „\uFE64script\uFE65“ sowie die Zeichenfolge „<script>“ werden beispielsweise

[4]https://owasp.org/www-project-cheat-sheets/cheatsheets/Input_Validation_Cheat_Sheet.html, aufgerufen am 28.02.2020

gleich interpretiert. Um sicherzustellen, dass ein regulärer Ausdruck, der spitze Klammern in einer Eingabe erkennen soll, diese auch in dem zuvor genannten Unicode-Text findet, muss die String-basierte Eingabe gemäß dieser Richtlinie zuvor auf eine normalisierte Form reduziert werden.

IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method
Eingaben können als Argumente an einen Aufruf der Funktion `Runtime.exec()` übergeben werden, um externe Programme auszuführen. Werden diese Eingaben nicht validiert, so kann es zu Command- oder Argument-Injektionen kommen. Diese Richtlinie befasst sich mit diesem spezifischen Verwendungszweck und besagt, dass entsprechende Eingaben beispielsweise durch typische String-Validierungen anhand regulärer Ausdrücke zu überprüfen sind oder dass stattdessen vordefinierte, auswählbare Werte bereitzustellen sind. Falls möglich sollte der Aufruf der `Runtime.exec()`-Methode idealerweise vermieden werden, etwa indem die Methode `File.list()` statt eines Aufrufes von `runtime.exec("cmd.exe /C dir")` genutzt wird.

IDS11-J. Perform any string modifications before validation
Gemäß dieser Richtlinie darf ein String nach seiner Validierung nicht mehr verändert werden. Im Hinblick auf die Vermeidung von Cross-Site-Scripting Angriffen könnte eine Eingabe beispielsweise validiert werden, indem sichergestellt wird, dass sie die Zeichenfolge „<script>“ nicht enthalten darf. Die Eingabe „<scr!ipt>“ würde dementsprechend als zulässig gelten. Falls im Anschluss an die Validierung alle Ausrufezeichen aus der Eingabe entfernt würden, käme es in diesem Fall zu einer potenziellen Verwundbarkeit, da der entstandene String den Validierungsprozess nicht mehr bestehen würde.

IDS14-J. Do not trust the contents of hidden form fields
Diese Richtlinie besagt, dass auch die Werte von Formularfeldern, die für den Benutzer der Software nicht sichtbar sind, validiert werden müssen. Dies ist notwendig, da Angreifer diese Werte dennoch modifizieren könnten, beispielsweise indem der Wert eines entsprechenden `GET`-Parameters in einer URL verändert wird. Der anzuwendende Validierungsprozess hängt hierbei von dem Verwendungszweck der Eingabe ab. Falls die Eingabe beispielsweise in einer SQL-Abfrage verwendet wird, muss sie wie zuvor beschrieben anders validiert werden als wenn sie als Argument für den Aufruf von `Runtime.exec()` genutzt wird.

IDS16-J. Prevent XML Injection
Diese Richtlinie beschreibt den spezifischen Anwendungsfall von Eingaben, welche in ein XML-Dokument integriert werden. Diese Eingaben müssen validiert werden, da sonst Werte verwendet werden können, die dazu führen, dass das XML-Dokument falsch interpretiert wird. Durch die Integration von XML-Tags in die Eingabe könnten beispielsweise andere Werte in identischen Tags, die bereits in dem Dokument enthalten sind, überschrieben werden. Numerische Eingaben können hierbei etwa durch eine Konvertierung in einen passenden Datentyp validiert werden, während bei der Verwendung komplexerer, String-basierter Eingaben anhand eines XML-Schemas zu prüfen ist, ob das generierte XML-Dokument der erwarteten Struktur entspricht oder ob eine XML-Injektion vorliegt.

6.2.3 Folgerungen für die Entwicklung neuer Ansätze

Bereits durch die grobe Zusammenfassung der sechs ausgewählten Richtlinien für die Eingabevalidierung in dem vorherigen Abschnitt wird ersichtlich, dass es sich hierbei um eine komplexe Thematik handelt, da je nach Datentyp und Verwendungszweck der Eingabe völlig unterschiedliche Vorgehensweisen für die Validierung anzuwenden sind. Zudem bestehen verschiedene Zusammenhänge zwischen den einzelnen Richtlinien. Die Richtlinie *IDS01-J* hält beispielsweise fest, dass Strings vor ihrer Validierung normalisiert werden müssen, während *IDS11-J* besagt, dass Strings nach ihrer Validierung nicht mehr verändert werden dürfen.

Diese Beobachtungen legen nahe, dass Ansätze nötig sind, welche den Entwickler bei dem Vermeiden oder Beheben von Fehlern bei der Eingabevalidierung unterstützen, da basierend auf dem jeweiligen Kontext verschiedenste Vorgehensweisen anzuwenden und Feinheiten zu beachten sind. Entsprechende Ansätze können also besonders hilfreich sein, wenn sie den Verwendungszweck der Eingabe miteinbeziehen, um sicherzustellen, dass die Vorgehensweisen und Feinheiten der jeweils relevanten Richtlinien eingehalten werden, ohne dass der Entwickler diese selbst kennen und anwenden muss. Da die Richtlinien teils miteinander zusammenhängen und sehr feingranulare Probleme beschreiben, sollten zudem Ansätze entwickelt werden, welche nicht nur bei der isolierten Einhaltung der einzelnen Richtlinien helfen, sondern diese Problematiken auch im größeren Kontext der Thematik der Eingabevalidierung betrachten. Die Anforderungen an die Kontextsensitivität der Ansätze werden bei bestimmten Verwendungszwecken, wie etwa der Vermeidung von SQL-Injektionen, weiter verschärft, da hier eine spezifische Hilfestellung unter Beachtung der verwendeten Datenbankschnittstelle nötig ist. Diese Erkenntnis entspricht auch der Beobachtung von Omar et al. (2012, Seite 859), dass Werkzeuge, die eine Hilfestellung für die Umsetzung korrekten Programmcodes leisten sollen, individuell auf die Eigenheiten des jeweiligen Verwendungszweckes zugeschnitten sein müssen.

Im Hinblick auf die Entwicklung neuer Ansätze und unter Beachtung der Erkenntnisse der in Kapitel 5.2 beschriebenen, bereits existierenden nicht-traditionellen Ansätze kann zudem gefolgert werden, dass eine vollautomatische Lösung des Problemes der Eingabevalidierung nur schwer realisierbar ist, da es sich hierbei nicht um ein syntaktisches, sondern um ein semantisches Problem handelt, dessen Behebung sich stark auf die Funktionsweise des jeweiligen Programmes auswirken kann. Dementsprechend ist hier die Umsetzung teilautomatischer, interaktiver Ansätze, welche etwa mit FixBugs und CogniCrypt vergleichbar sind, naheliegend, um den Entwickler bei der Umsetzung einer adäquaten Validierung zu unterstützen. Dies deckt sich mit der Annahme von Xie et al. (2011, Seite 161), dass interaktive Ansätze relevant sind, um sicherheitsbezogene Fehler in der Softwareentwicklung zu vermeiden. Für die Entwicklung der Ansätze in dieser Arbeit stellt die Idee teilautomatischer, interaktiver Vorgehensweisen, um eine korrekte Eingabevalidierung umzusetzen somit die Grundlage dar. Die wesentlichen Rahmenbedingungen für die Entwicklung der neuen Ansätze werden im Folgenden zusammengefasst:

1. Die zu entwickelnden Ansätze müssen den Entwickler bei der Vermeidung oder Behebung von Fehlern im Hinblick auf die Eingabevalidierung, speziell durch die Anwendung ausgewählter Secure Coding Richtlinien, unterstützen können.

2. Die Ansätze müssen einen Bezug zur statischen Code-Analyse haben. Entweder indem sie die Vorgehensweisen der statischen Code-Analyse verwenden oder indem sie von der Kombination mit traditionelleren SCATs profitieren, um Fehler zu erkennen, zu vermeiden oder zu beheben.

3. Es sollen ausschließlich teilautomatische bzw. interaktive Ansätze untersucht werden, da es sich bei der Eingabevalidierung um ein semantisches Problem handelt, dessen vollautomatische Behandlung vergleichsweise umständlich ist. Methoden wie die von Alkhalaf (2014) oder Scholte, Robertson, Balzarotti und Kirda (2012) beschriebenen sind hier damit nicht relevant.

Aufgrund der Komplexität der Thematik können im Rahmen dieser Arbeit keine vollständigen Lösungen für das Problem der Eingabevalidierung umgesetzt werden. Dies wird auch dadurch verdeutlicht, dass Spezialformen verschiedener Secure Coding Richtlinien existieren, wie etwa die Notwendigkeit zur Vermeidung von Hibernate Query Language (HQL)-Injektionen als Abwandlung der Richtlinie *IDS00-J. Prevent SQL injection*. Stattdessen werden mit den hier entwickelten Ansätzen die grundlegenden Ideen, um den Entwickler bei der Einhaltung einiger exemplarischer Richtlinien zu unterstützen in theoretischer Form, sowie in der Form praktischer Proof of Concepts festgehalten, um die technische Realisierbarkeit und Praxistauglichkeit zu veranschaulichen. Die Richtlinien *IDS14-J. Do not trust the contents of hidden form fields* und *IDS11-J. Perform any string modifications before validation* sind im weiteren Verlauf zudem nicht relevant, da es sich hierbei um Fehlerformen handelt, welche vor allem durch traditionelle statische Code-Analysen gefunden werden und nicht durch Herangehensweisen zur Implementierung korrekter Eingabevalidierungen vermieden werden können. Die Ergebnisse der hier entwickelten Ansätze können in der weiteren Forschung als Grundlage verwendet werden, um vollständigere Lösungen umzusetzen, die den Entwickler bei der Implementierung korrekter Eingabevalidierungen unterstützen.

6.3 Existierende Konzepte zur Umsetzung der Eingabevalidierung

Zuvor wurden in Abschnitt 5.2 verschiedene bereits existierende, nicht-traditionelle Ansätze im Zusammenhang mit statischer Code-Analyse unabhängig von der jeweiligen Problemform, die sie behandeln, betrachtet. Interaktive Ansätze, welche sich mit der Thematik der Eingabevalidierung befassen, konnten hierbei nicht gefunden werden. An dieser Stelle müssen daher zusätzlich allgemeine, bereits existierende und interaktive Konzepte für die Hilfe bei der Implementierung von Eingabevalidierungen betrachtet werden, die keinen Bezug zur statischen Code-Analyse haben. So sind Prinzipien zu identifizieren, welche für die Entwicklung der neuen Ansätze unter der Verwendung statischer Code-Analyse relevant sein können. Bei der Recherche wurden zwei wesentliche Konzepte erkenntlich, die im Folgenden beschrieben werden.

6.3.1 Einschränkungsbasierte Konzepte

Vergleichsweise weit verbreitet ist die Verwendung von Bibliotheken, die Annotationen bereitstellen, um Einschränkungen (*Constraints*) an die Werte, die eine Variable

annehmen kann, zu definieren und Funktionen für die Prüfung dieser Einschränkungen beinhalten. Eine Umsetzung dieses Konzeptes für Java ist *Bean Validation*[5], welche auf den Spezifikationen JSR 380[6], 349[7] und 303[8] basiert und für die verschiedene Implementierungen existieren, wie der Hibernate Validator[9] oder Apache BVal[10].

Die grundlegende Funktionsweise dieses Konzeptes sieht vor, dass die zu validierenden Informationen in einer JavaBean gekapselt werden, um eine Instanz der Klasse anschließend validieren zu können. Dies wird durch die Definition der JavaBean in Listing 6.1 veranschaulicht. Anhand der Annotationen werden hier die Einschränkungen festgelegt, deren Einhaltung durch den Validierungsprozess zu prüfen ist. Die Implementierung stellt für einige Verwendungszwecke fest definierte Annotationen bereit, wie `@Email` für die Prüfung der Struktur einer E-Mail-Adresse, während andere Annotationen konfiguriert werden können, etwa indem die Annotation `@Pattern` genutzt werden kann, um eigene reguläre Ausdrücke anzugeben.

```
1 public class User {
2   @Size(min = 5, max = 20,
3         message = "Name must be between 5 and 20 characters")
4   @Pattern(regexp="[a-zA-Z]+",
5         message = "Name can only consist of one word")
6   private String name;
7
8   @Email(message = "Email should be valid")
9   private String email;
10
11  // standard setters and getters
12 }
```

Listing 6.1: Definition einer validierbaren JavaBean

Die Instanziierung und programmatische Validierung dieser JavaBean wird in Listing 6.2 anhand des Hibernate Validators durchgeführt. Dabei werden die Verletzungen der Einschränkungen durch Objekte der Klasse `ConstraintViolation` festgehalten, wodurch der Entwickler seinen Quellcode so verfassen kann, dass das Programm auf die unterschiedlichen Fehlerformen reagieren kann.

```
1 User user = new User();
2 user.setName("ValidName");
3 user.setEmail("InvalidEmail");
4
5 Set<ConstraintViolation<User>> violations = validator.validate
    (user);
6 violations.forEach(v -> System.err.println(v.getMessage()));
```

Listing 6.2: Durchführung einer programmatischen Validierung einer JavaBean

[5]https://beanvalidation.org/, aufgerufen am 02.03.2020
[6]https://www.jcp.org/en/jsr/detail?id=380, aufgerufen am 02.03.2020
[7]https://www.jcp.org/en/jsr/detail?id=349, aufgerufen am 02.03.2020
[8]https://www.jcp.org/en/jsr/detail?id=303, aufgerufen am 02.03.2020
[9]https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/?v=5.3, aufgerufen m 02.03.2020
[10]http://bval.apache.org/, aufgerufen am 02.03.2020

Diese Annotationen können auch für die Parameter und Rückgabewerte von Funktionen verwendet werden, um anzugeben, welche Anforderungen eine aufrufende Methode erfüllen muss oder um Garantien über die Rückgabe bieten zu können[11] und das Konzept der Bean Validation kann durch die Definition eigener Annotationen und Validierungsmethoden erweitert und spezialisiert werden[12]. Bean Validation findet zudem insbesondere in Frameworks für die Entwicklung von Web-Applikationen wie Struts[13] oder Spring[14] Verwendung, sodass eine manuelle, programmatische Validierung nicht durchgeführt werden muss.

Einschränkungsbasierte Konzepte existieren auch abseits der hier beschriebenen Bean Validation. Struts setzt beispielsweise auch einen eigenen Mechanismus um, bei dem die Einschränkungen entweder anhand von XML-Dokumenten oder ebenfalls durch Annotationen umgesetzt werden können[15] und das Apache Commons Validator Projekt[16] stellt einen vergleichbaren, XML-basierten Mechanismus bereit.

6.3.2 API-basierte Konzepte

Es existieren verschiedene APIs, die Funktionen bereitstellen, um Eingaben für ihren jeweiligen Verwendungszweck zu validieren, etwa indem sie etwa das Format eines Datums prüfen, eine Konvertierung in einen gültigen Pfadnamen durchführen oder sicherstellen, dass eine Eingabe bei ihrer Verwendung in einer SQL-Abfrage nicht zu einer SQL-Injektion führen kann. Ein solches Konzept unterstützt den Entwickler insofern bei der Umsetzung korrekter Validierungen, als dass er die Feinheiten des jeweiligen Validierungsprozesses nicht selbst kennen und umsetzen muss und stattdessen lediglich eine einfacher zu verwendende Schnittstelle nutzen kann.

Die OWASP Enterprise Security API (ESAPI)[17] stellt verschiedenste Funktionen zu diesem Zweck bereit und Validierungsprozesse, welche diese API nicht abdeckt, können durch die eigene Implementierung der bereitgestellten Interfaces realisiert werden. Listing 6.3 veranschaulicht, wie SQL-Injektionen bei der Integration von Eingaben in eine Abfrage anhand von ESAPI vermieden werden können. Vergleichbare Funktionen sind teils auch in Architekturframeworks wie Django[18] oder Vaadin[19] enthalten.

```
1 String encodedQuery = "SELECT * FROM user WHERE name = '"+
    encoder.encodeForSQL(mysqlCodec, name) + "'";
```

Listing 6.3: Unterbindung von SQL-Injektionen anhand von ESAPI

[11] https://docs.jboss.org/hibernate/validator/5.1/reference/en-US/html/chapter-method-constraints.html, aufgerufen am 02.03.2020
[12] https://docs.jboss.org/hibernate/validator/5.0/reference/en-US/html/validator-customconstraints.html, aufgerufen am 02.03.2020
[13] https://struts.apache.org/plugins/bean-validation/, aufgerufen am 02.03.2020
[14] https://spring.io/guides/gs/validating-form-input/, aufgerufen am 02.03.2020
[15] https://struts.apache.org/core-developers/validation.html, aufgerufen am 02.03.2020
[16] https://commons.apache.org/proper/commons-validator/, aufgerufen am 02.03.2020
[17] https://owasp.org/www-project-enterprise-security-api/, aufgerufen am 02.03.2020
[18] https://docs.djangoproject.com/en/1.11/ref/validators/, aufgerufen am 02.03.2020
[19] https://vaadin.com/docs/flow/binding-data/tutorial-flow-components-binder-validation.html, aufgerufen am 02.03.2020

6.3.3 Beobachtungen

Im Hinblick auf die existierenden Konzepte zur Unterstützung des Entwicklers bei der Umsetzung korrekter Eingabevalidierungen können einige Feststellungen getroffen werden, welche für die Entwicklung der neuen Ansätze unter der Verwendung statischer Code-Analyse relevant sind.

Die erste wesentliche Beobachtung ergibt sich im Hinblick auf den Verwendungszweck der jeweiligen Eingabe. Sowohl einschränkungsbasierte als auch API-basierte Konzepte bieten Möglichkeiten, um Daten mit einem allgemeinen Verwendungszweck zu validieren, beispielsweise indem ein Maximalwert für eine Variable vom Typ `int` festgelegt oder ein String auf ein beliebiges Format hin validiert werden kann. Für spezifische Verwendungszwecke, welche auch durch die ausgewählten Secure Coding Richtlinien behandelt werden, wie etwa das Vermeiden von SQL-Injektionen, konnten nur die Funktionen der ESAPI gefunden werden. Diese Funktionen führen jedoch eine manuelle Maskierung potenziell gefährlicher Symbole durch, statt die bereitgestellten Funktionalitäten der jeweiligen Datenbankschnittstelle zu verwenden[20]. Dies widerspricht der Vorgehensweise der Richtlinie *IDS00-J. Prevent SQL injection* und ist vergleichsweise fehleranfällig, da die Wartung der Funktionen der API in der Regel weniger gut gewährleistet wird als die Wartung der Datenbankschnittstelle. Zudem muss auch die Konsistenz zwischen der Funktionsweise der API und der Funktionsweise des verwendeten Datenbankverwaltungssystemes garantiert werden. Dies bestärkt die zuvor aufgestellte Rahmenbedingung, dass neue Ansätze die konkreten Vorgehensweisen der Richtlinien des CERT-Standards umsetzen müssen.

Die zweite wichtige Beobachtung ist, dass sowohl das einschränkungsbasierte Konzept als auch das API-basierte Konzept den Entwickler grundsätzlich bei der Durchführung verschiedener Validierungen unterstützen können. Fälle, die durch die ausgelieferten Funktionalitäten nicht abgedeckt werden, müssen jedoch auch hier selbst implementiert werden, beispielsweise durch die manuelle Implementierung neuer Annotationen und Validatoren bei der Verwendung von Bean Validation. Dabei sind die je nach Verwendungszweck relevanten Richtlinien wiederum manuell einzuhalten und es entsteht ein zusätzlicher Aufwand. Abgesehen davon müssen auch bei der Verwendung der hier beschriebenen Konzepte verschiedene Richtlinien wie etwa *IDS01-J. Normalize strings before validating them* durch den Entwickler selbst eingehalten werden, um eine korrekte Validierung zu ermöglichen. Dies bestärkt ebenfalls die zuvor aufgestellte Rahmenbedingungen, dass dem Entwickler bei der Einhaltung verschiedener Secure Coding Richtlinien geholfen werden muss.

6.4 Auswahl grundlegender Konzepte

Grundlegende Konzepte, die neue Ansätze zur Fehlerkorrektur und Fehlervermeidung verfolgen sollten, werden durch verschiedene Studien vorgeschlagen. Untersuchungen wie die von Xie et al. (2011), Xie, Lipford und Chu (2012), B. Johnson et al. (2013) oder Nadi et al. (2016) zeigen, dass sicherheitsrelevanter Programmcode möglichst automatisiert bereitgestellt werden sollte, um effiziente und effektive Hilfe zu leisten. Krüger et al. (2017) beschreiben zudem, dass ein solches Konzept keine vollständigen

[20] https://github.com/ESAPI/esapi-java-legacy/blob/develop/src/main/java/org/owasp/esapi/reference/DefaultEncoder.java, aufgerufen am 03.03.2020

Lösungen generieren muss und dass bereits Code-Vorlagen hilfreich sein können, die durch den Entwickler angepasst werden müssen. Meng et al. (2017) schlagen vergleichbar dazu die Untersuchung von Ansätzen vor, um Fehler von vornherein zu vermeiden. Für die Unterstützung des Entwicklers bei der Behebung von Validierungsfehlern müssen somit Ansätze entwickelt werden, welche für den jeweiligen Kontext korrekten Programmcode generieren oder den Programmcode in einen Zustand transformieren, in dem eine korrekte Validierung gewährleistet werden kann.

Im Hinblick auf diese Vorschläge sowie die zuvor in Abschnitt 5.2 beschriebenen nicht-traditionellen Ansätze unter der Verwendung statischer Code-Analyse und die in Abschnitt 6.3 erläuterten existierenden Konzepte für die Unterstützung bei der Umsetzung von Eingabevalidierungen, wurden im Rahmen dieser Arbeit drei neue Ansätze betrachtet, die im Folgenden beschrieben werden. Zur Verdeutlichung des Prinzips der Ansätze und ihrer Brauchbarkeit sowie Realisierbarkeit und um eine praktische Grundlage für die weitere Forschung und Entwicklung zu schaffen, werden diese an Implementierungen im Sinne eines Proof of Concepts veranschaulicht. Diese Proof of Concepts wurden für die Hilfe bei der Anwendung der zuvor ausgewählten Richtlinien *IDS01-J. Normalize strings before validating them*, *IDS00-J. Prevent SQL injection* und *IDS16-J. Prevent XML Injection* entwickelt, da an diesen besonders gut veranschaulicht werden kann, wie die jeweiligen Ansätze bei der Durchführung syntaktischer sowie semantischer Validierungen von Eingaben mit allgemeinen und spezifischen Verwendungszwecken helfen können. Weiterhin wird im Hinblick auf die Richtlinie *IDS01-J* zudem besser erkenntlich, warum Ansätze für die Anwendung isolierter Richtlinien nicht sinnvoll sind und wie stattdessen eine Hilfestellung für die übergeordnete Problematik der Eingabevalidierung zu leisten ist. Hinweise zu der Ausführung der Implementierungen sind in Anhang B zu finden.

6.5 Ansatz 1: Annotationsbasierte Validierung

Dieser Abschnitt beschreibt den ersten Ansatz, der im Rahmen dieser Arbeit näher betrachtet wurde. Er basiert auf der Idee des zuvor in Abschnitt 6.3.1 beschriebenen Konzeptes, um Annotationen zu verwenden, anhand derer Einschränkungen definiert werden, die anschließend bei der Durchführung des Validierungsprozesses geprüft werden. Diese Idee entspricht damit auch dem Vorschlag von Meng et al. (2017), Ansätze zu entwickeln, die zwischen Implementierungen deklarativer und imperativer Sicherheit transformieren.

6.5.1 Konzept

Die notwendigen Komponenten werden in Abbildung 6.3 veranschaulicht. Ebenso wie bei dem zuvor in Abschnitt 6.3.1 beschriebenen Konzept für die Verwendung von Annotationen, um Eigenschaften für den Validierungsprozess festzuhalten, werden auch hier eine Annotationssprache und eine API genutzt. Das Konzept wird dabei jedoch um die Verwendung eines SCATs erweitert.

Ein SCAT kann hierbei einen Mehrwert bieten, indem es durch Taint Propagation Analysen feststellt, ob eine Eingabe vor ihrer Verwendung in einer Senke mit Annotationen versehen und durch die API validiert wird. Der annotationsbasierte Validierungsprozess kann somit durch das SCAT als Reinigung einer Eingabe inter-

pretiert werden. Dementsprechend würde das Analysewerkzeug ausschließlich Fehler berichten, falls es keinen solchen Reinigungsprozess zwischen der Quelle und der Senke erkennt. Der eigentliche Validierungsprozess könnte etwa wie bei dem zuvor beschriebenen Konzept der Bean Validation durch entsprechende Funktionen einer API umgesetzt werden. Diese betrachten die im Quellcode definierten Annotationen und stellen die Einhaltung von Richtlinien wie etwa *IDS01-J. Normalize strings before validating them* sicher. Diese Komponenten könnten somit eine sinnvolle Ergänzung eines traditionellen SCAT darstellen, da existierende Werkzeuge Fehler üblicherweise gut erkennen können, jedoch nicht in der Lage sind, den Programmcode zur Validierung zu erfassen (Zhu et al., 2013, Seite 2).

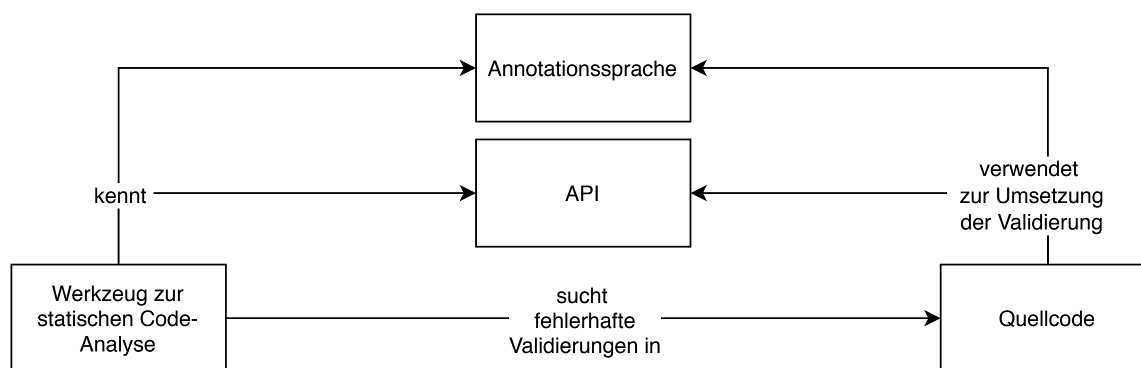

Abbildung 6.3: Komponenten für einen Ansatz zur annotationsbasierten Validierung

Die zentrale Komponente dieses Ansatzes ist somit eine Annotationssprache, die zusätzlich durch eine API ergänzt wird. Beide dieser Komponenten werden durch den Entwickler im Quellcode angewendet und müssen dem SCAT bekannt sein.

6.5.2 Erste Beobachtungen

Bei der Betrachtung dieses Ansatzes stellt sich zunächst die Frage, wie Annotationen konkret verwendet werden können, um den Validierungsprozess zu vereinfachen. Syntaktische und semantische Validierungen einer Eingabe mit einem allgemeinen Verwendungszweck können dabei genauso realisiert werden, wie bei dem Konzept der Bean Validation: Die bereitgestellte API setzt den Validierungsprozess um und bezieht die nötigen Informationen zur Laufzeit per Reflection aus den angegebenen Annotationen und prüft, ob die jeweilige Eingabe diese Anforderungen erfüllt.

Listing 6.4 veranschaulicht das Prinzip dieses Ansatzes an einem theoretischen Minimalbeispiel. In Zeile zwei definiert die Annotation `@Size` die Eigenschaften für den Validierungsprozess, während in Zeile fünf anhand der beigelegten API die Validierung durchgeführt wird. Ein SCAT, dem die API und die Annotationen bekannt sind und das eine Regel kennt, die besagt, dass die Funktion `sink()` als Senke definiert ist, die nur validierte Eingaben erhalten darf, könnte feststellen, dass der Aufruf dieser Funktion in Zeile sechs keinen Fehler darstellt, da zuvor in Zeile fünf die Validierung erfolgt ist. Dementsprechend könnte das SCAT den Aufruf in Zeile 13 als Fehler erkennen.

```
1  public static void main(String[] args) {
2    @Size(min = 5, max = 20)
3    String name = args[0];
4
5    if(Validator.isValid(name)) {
6      sink(name);
7    }
8    else {
9      // report error
10   }
11
12   // method call reported by SCAT:
13   sink(name);
14 }
15
16 public static void sink(String name) {
17   // accept only valid inputs
18 }
```

Listing 6.4: Quellcode für das Konzept eines annotationsbasierten Ansatzes

Viele Richtlinien befassen sich jedoch auch mit der Validierung von Eingaben für einen spezifischen Verwendungszweck und stellen daher fest definierte Einschränkungen für die Vorgehensweise zur Fehlervermeidung auf, wie beispielsweise *IDS00-J. Prevent SQL injection* oder *IDS16-J. Prevent XML Injection*. Die zuvor beschriebenen bestehenden Konzepte zur Hilfe bei der Eingabevalidierung unter der Nutzung von Annotationen, auf denen dieser Ansatz basiert, bieten keine Möglichkeiten für die Vermeidung von SQL- oder XML-Injektionen. Bei der näheren Betrachtung dieses Ansatzes wird schnell erkenntlich, dass sich die Verwendung von Annotationen zu diesem Zweck nicht eignet, da in diesen Fällen keine bestimmten Anforderungen durch eine Eingabe eingehalten werden müssen, sondern konkrete Vorgehensweisen anzuwenden sind, um Validierungsfehler auszuschließen.

Als maßgebliches Beispiel zur Verdeutlichung dieser Tatsache kann die Richtlinie *IDS00-J. Prevent SQL injection* angeführt werden. Das Listing 6.5 zeigt einen Quellcodeauszug, der für SQL-Injektionen anfällig ist, während Listing 6.6 die durch die Richtlinie empfohlene Vorgehensweise unter der Verwendung der entsprechenden Funktionen der Datenbankschnittstelle zeigt, um SQL-Injektionen zu unterbinden.

```
1 String input = args[0];
2 String sqlString =
3   "SELECT * FROM users WHERE username = '" + input + "'";
4 Statement stmt = connection.createStatement();
5 ResultSet resultSet = stmt.executeQuery(sqlString);
```

Listing 6.5: Für SQL-Injektionen anfälliger Quellcode

```
1 String input = args[0];
2 String sqlString =
3   "SELECT * FROM users WHERE username = ?";
4 PreparedStatement stmt =
5   connection.prepareStatement(sqlString);
6 stmt.setString(1, input);
7 ResultSet resultSet = stmt.executeQuery(sqlString);
```

Listing 6.6: Quellcode für die Vermeidung von SQL-Injektionen unter der Anwendung der Richtlinie IDS00-J

An dem hier veranschaulichten Quellcode wird schnell ersichtlich, dass Annotationen sich für die Unterstützung bei der Anwendung dieser Richtlinie nicht eignen. Dies ist der Tatsache geschuldet, dass Annotationen genutzt werden, um Zusatzinformationen in dem Quellcode zu hinterlegen, die durch ein Programm ausgewertet werden können. Bei der Anwendung der hier beschriebenen Richtlinie bieten Annotationen jedoch keinen Mehrwert, da sämtliche Informationen, die zur Umsetzung der korrekten Vorgehensweise für die Vermeidung von SQL-Injektionen nötig sind, bereits in dem Quellcode vorhanden sind.

Alternativ können Annotationen auch verwendet werden, um Programmcode im Rahmen des Kompilationsprozesses generieren zu lassen. Diese Vorgehensweise wird beispielsweise durch die Bibliothek „Project Lombok“[21] umgesetzt, um etwa standardmäßige Zugriffsfunktionen auf Eigenschaften eines Objektes automatisch generieren zu können. Die Verwendung von Annotationen, um etwa den Code zur Vermeidung einer SQL-Injektion generieren zu lassen, eignet sich ebenfalls nicht, da dieser je nach Richtlinie und Kontext der Eingabe sehr individuell ausfallen kann und ausschließlich im Bytecode existiert, weshalb der Entwickler ihn nicht korrekt bei der Implementierung der restlichen Software miteinbeziehen kann.

Da es sich bei SQL-Injektionen um eine besonders weit verbreitete Angriffsform mit potenziell schwerwiegenden Konsequenzen handelt, die mit diesem Ansatz nicht sinnvoll vermieden werden kann, wird deutlich, dass sich diese Variante nicht eignet, um die effektive Durchsetzung der Eingabevalidierung zu gewährleisten.

Je nach Richtlinie könnten zwar gänzlich unterschiedliche Herangehensweisen umgesetzt werden, beispielsweise indem die Validierung von Strings mit einem allgemeinen Verwendungszweck wie hier beschrieben anhand von Annotationen durchgeführt wird, während die Vermeidung von SQL-Injektionen ohne die Nutzung von Annotationen durchgesetzt wird. Eine solche Vorgehensweise würde jedoch dazu führen, dass der Entwickler unterschiedliche Herangehensweisen je nach Fehlertyp anwenden muss, wodurch die Verwendung dieses Ansatzes deutlich erschwert werden würde. Die Abschnitte 6.6 und 6.7 beschreiben hingegen konsistentere Ansätze.

Da diese Möglichkeit bereits bei einer theoretischen Betrachtung offensichtliche konzeptionelle Mängel aufweist und sich nicht für die Unterstützung bei der Anwendung verschiedener wichtiger Secure Coding Richtlinien eignet, wurde im Rahmen dieser Arbeit kein Proof of Concept implementiert und dieser Ansatz wird im weiteren Verlaufe der Arbeit nicht näher betrachtet.

[21] https://projectlombok.org/, aufgerufen am 13.04.2020

6.6 Ansatz 2: APIs zur Validierung

In diesem Abschnitt wird der zweite Ansatz beschrieben, der im Rahmen dieser Arbeit konzipiert wurde. Dieser befasst sich basierend auf den Erkenntnissen des in Abschnitt 6.3.2 beschriebenen Konzeptes für die Umsetzung von Eingabevalidierung anhand von APIs mit der Idee, eine API zu entwickeln, welche auf die Verwendung in Kombination mit einem SCAT ausgelegt ist, um Validierungsfehler zu erkennen und zu vermeiden.

6.6.1 Konzept

Die zentralen Komponenten dieses Ansatzes und ihre Zusammenhänge werden in Abbildung 6.4 dargestellt. Die Idee des Ansatzes basiert darauf, dass ein SCAT den Programmcode des Entwicklers analysiert, um fehlerhafte Eingabevalidierungen zu erkennen und zu berichten. Um eine korrekte Validierung zu implementieren, kann der Entwickler je nach Verwendungszweck der Eingabe die entsprechenden Funktionen einer API zur Eingabevalidierung verwenden. Die API ist dem SCAT bekannt, damit dieses erkennen kann, dass bei der Verwendung der API eine korrekte Validierung durchgeführt wird und dementsprechend keinen Fehler berichtet. Eine solche Kombination ist sinnvoll, da SCATs Eingaben grundsätzlich gut erkennen können, jedoch häufig nicht in der Lage sind, den vorhandenen Programmcode zur Validierung dieser Eingaben zu identifizieren (Zhu et al., 2013, Seite 2).

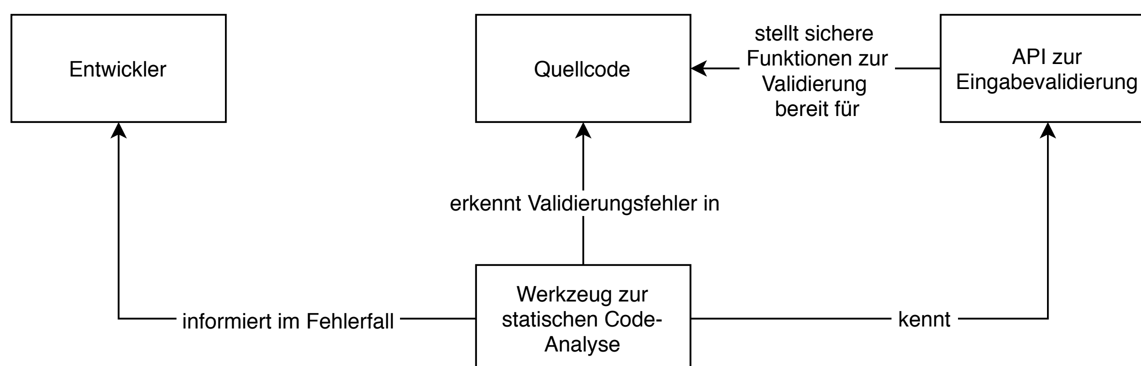

Abbildung 6.4: Komponenten für einen API-basierten Ansatz

Eine entsprechend konzipierte API kann zudem sicherstellen, dass die für den jeweiligen Verwendungszweck der Eingabe notwendigen Secure Coding Richtlinien garantiert eingehalten werden: Indem eine Funktion zur Validierung einer Eingabe vom Datentyp `String` gemäß der Richtlinie *IDS01-J. Normalize strings before validating them* eine Normalisierung der Eingabe vor ihrer Validierung durchführt, kann vermieden werden, dass der Entwickler dies aufgrund der hohen kognitiven Belastung durch die Umsetzung des restlichen Programms vergisst oder eine Normalisierung nicht implementiert, da er die verschiedenen Richtlinien zur Eingabevalidierung und ihre Feinheiten nicht kennt. Eine entsprechende API sollte somit eine möglichst vollständige Funktionalität zur Eingabevalidierung bereitstellen und garantieren können, dass ein Aufruf der Funktionen eine korrekte Validierung für den jeweiligen Verwendungszweck unter der Einhaltung der dafür relevanten Secure Coding Richtlinien durchführt. Die genauere Funktionsweise dieses Ansatzes wird in Abschnitt 6.6.2 anhand praktischer Beispiele veranschaulicht.

6.6.2 Funktionsweise an praktischen Beispielen

Um feststellen zu können, inwiefern ein solcher Ansatz umsetzbar ist, sowie um seine Funktionsweise und Praxistauglichkeit zu veranschaulichen, wurde ein Proof of Concept implementiert. Dieser verwendet das Checker Framework[22], um statische Code-Analysen durchzuführen. Außerdem wurde eine API umgesetzt, welche die Funktionalitäten zur Durchführung der Eingabevalidierungen basierend auf ihrem Verwendungszweck und unter der Einhaltung einiger relevanter Secure Coding Richtlinien bereitstellt. Der hier implementierte Proof of Concept deckt die in Tabelle 6.1 aufgelisteten Verwendungszwecke und Richtlinien ab.

Verwendungszweck der Eingabe	*Eingehaltene Richtlinien*
String (allgemeiner Verwendungszweck)	**IDS01-J.** Normalize strings before validating them
String als Pfadname (spezifischer Verwendungszweck)	**IDS01-J.** Normalize strings before validating them **FIO16-J.** Canonicalize path names before validating them
String, Integer als Teil einer SQL-Abfrage (spezifischer Verwendungszweck)	**IDS00-J.** Prevent SQL injection
String, Integer als Teil eines XML-Dokuments (spezifischer Verwendungszweck)	**IDS16-J.** Prevent XML Injection

Tabelle 6.1: Umfang der Implementierung des Proof of Concepts des API-basierten Ansatzes

Für die Durchführung der statischen Analysen im Rahmen des Proof of Concepts wurde das Checker Framework genutzt, da es vergleichsweise einfach verwendet und erweitert werden kann und typische Funktionalitäten für die Durchführung von Taint Propagation Analysen bietet. Zudem ist es in der Lage, den Programmcode zu erkennen, der eine Validierung durchführt. Das Checker Framework funktioniert dabei ähnlich wie der zuvor beschriebene Ansatz CQUAL, indem das Typsystem von Java anhand von Annotationen ergänzt wird, die durch Compiler Plugins *(„Checkers“)* analysiert werden, um Fehler zu erkennen. Im Hinblick auf die Validierung von Eingaben werden Annotationen hier standardmäßig verwendet, um Quellen, Senken und Reinigungen zu markieren. Diese Annotationen können entweder direkt im Quellcode angegeben oder in sogenannten Stub-Dateien separat bereitgestellt werden. Das Checker Framework liefert standardmäßig verschiedene solcher Stub-Dateien für grundlegende Funktionen von Java mit aus. Bei einer Analyse des in Listing 6.7 dargestellten Quellcodes produziert das Checker Framework beispielsweise die in Abbildung 6.5 dargestellte Warnung. Auf der Basis einer Datenflussanalyse erkennt das Werkzeug hier, dass eine als `@Tainted` gekennzeichnete und somit nicht validierte Eingabe an

[22] https://checkerframework.org/, aufgerufen am 04.04.2020

die Methode `sink()` übergeben wird, welche durch die Markierung ihres Parameters als `@Untainted` jedoch eine validierte Eingabe erfordert[23].

```
14 public static void main(@Tainted String[] args) {
15   String input = args[0];
16   sink(input);
17 }
18
19 public static void sink(@Untainted String input) {
20   // use only validated inputs
21 }
```

Listing 6.7: Beispiel für die Verwendung der Annotationen des Checker Frameworks

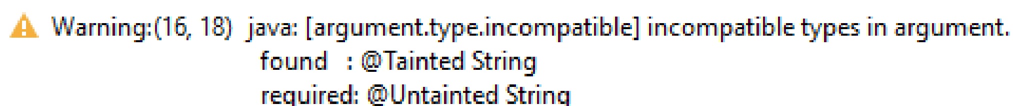

Abbildung 6.5: Warnung des Checker Frameworks für eine fehlende Validierung

Anhand des Proof of Concepts werden im Folgenden Beispiele für die Validierung eines Strings mit allgemeinem Verwendungszweck und die Vermeidung von SQL-Injektionen beschrieben, um die grundlegenden Ideen zu verdeutlichen, die für diesen Ansatz und seine praktische Umsetzung relevant sind. Diese beiden Richtlinien wurden exemplarisch gewählt, da sie besonders die wesentlichen Prinzipien der Eingabevalidierung zur syntaktischen sowie semantischen Validierung von Eingaben mit allgemeinen und spezifischen Verwendungszwecken abdecken. Im Rahmen der Umsetzung des Proof of Concepts war fraglich, ob eine statische API auch komplexere Richtlinien einhalten kann. Um dies zu untersuchen wurde zusätzlich eine Möglichkeit für die Vermeidung von XML-Injektionen implementiert, die anschließend ebenfalls beschrieben wird.

Validierung eines Strings mit allgemeinem Verwendungszweck

Um Funktionen bereitzustellen, welche in der Lage sind, die relevanten Secure Coding Richtlinien für die Validierung von Eingaben des Datentyps `String` ohne einen spezifischen Verwendungszweck garantiert einzuhalten, wurde in dem Proof of Concept die Klasse `UniversalStringValidator` implementiert, welche intern mit Annotationen versehen ist, die dem Checker Framework signalisieren, an welchen Stellen eine Validierung stattfindet. Diese Klasse vereinfacht es für den Entwickler zudem, an unterschiedliche, eventuell relevante Einschränkungen zu denken, indem sie entsprechende Funktionen bereitstellt, um beispielsweise festzulegen, dass eine Eingabe eine bestimmte minimale Länge aufweisen muss. Listing 6.8 zeigt, wie diese API verwendet werden kann, um eine korrekte Validierung eines Strings mit allgemeinem Verwendungszweck umzusetzen.

[23]Der Parameter der Main-Methode muss nicht im Quellcode als `@Tainted` markiert werden, da das Checker Framework dies anhand einer mitgelieferten Stub-Datei erkennt. Die Annotation wurde an dieser Stelle lediglich zur Veranschaulichung der Funktionsweise des Werkzeugs angegeben.

```
14 public static void main(@Tainted String[] args) {
15   String input = args[0];
16
17   // accepted format: <#>Text:
18   @Regex String pattern = "<\\d+>[^\\n]+";
19
20   UniversalStringValidator stringValidator =
21           new UniversalStringValidator.Builder(false)
22           .setCheckMinLength(4)
23           .setCheckMaxLength(12)
24           .setCheckMustMatchPattern(pattern)
25           .build();
26
27   if(stringValidator.isValid(input)) {
28       System.out.println("Input is valid");
29       // Checker Framework will not report this call:
30       sink(input);
31   }
32   else {
33       System.err.println("Input is not valid");
34       ArrayList<StringValidationError> errors =
35         stringValidator.getValidationErrorsFor(input);
36       errors.forEach(e -> System.err.println(e.getMessage()));
37   }
38
39   // Checker Framework would still report this call:
40   // sink(input);
41 }
```

Listing 6.8: Verwendung des API-basierten Ansatzes zur Validierung einer Eingabe

In diesem Beispiel instanziiert der Entwickler anhand der bereitgestellten API in den Zeilen 20 bis 25 ein Objekt der Klasse `UniversalStringValidator`, welches für die Validierung zuständig ist und über die semantischen Anforderungen an eine Eingabe, wie etwa die minimale und maximale erlaubte Länge, informiert wird. Über die Funktion `setCheckMustMatchPattern()` wendet der Entwickler ein Whitelisting-Prinzip an, damit der Validator prüft, ob die Eingabe das übergebene Muster erfüllt. Im Gegensatz zu diesen optionalen Anforderungen werden nicht optionale Anforderungen anhand des Konstruktors festgelegt. In diesem Beispiel gibt der Parameter des Konstruktors an, ob die Eingabe den Wert `NULL` annehmen darf. Die API kann zudem garantieren, dass verschiedene relevante Secure Coding Richtlinien während der Validierung garantiert eingehalten werden. In diesem einfachen Beispiel wird lediglich die Richtlinie *IDS01-J. Normalize strings before validating them* automatisch durch die API eingehalten. Grundsätzlich stellt die API somit eine einfache Möglichkeit für den Entwickler bereit, um die Anforderungen an eine gültige Eingabe zu formulieren und kann ihn auch durch die Existenz der einzelnen Funktionen und Konstruktoren an die Umsetzung potenziell relevanter Anforderungen erinnern.

In Zeile 27 wird der instanziierte Validator verwendet, um die Eingabe zu validieren. Ist die Eingabe gültig, kann sie in dem Then-Block der If-Abfrage verwendet

werden, ist sie ungültig, kann wie in den Zeilen 34 bis 36 dargestellt eine Liste der Fehler während des Validierungsprozesses abgerufen werden. Durch die API-interne Verwendung entsprechender Annotationen des Checker Frameworks ist die in Zeile 27 aufgerufene Funktion `isValid()` zudem dem SCAT bekannt und signalisiert ihm, dass die als Argument übergebene Eingabe als gültig und somit `@Untainted` anzuerkennen ist, falls die Funktion den Wert `true` zurückgibt; siehe Listing 6.9. Aus diesem Grund würde das Checker Framework bei einer Analyse nicht vor dem Aufruf der Methode `sink()` in Zeile 30 warnen, während ein Aufruf in Zeile 40 in einer Warnung resultieren würde. Diese erste Implementierung einer API zur Validierung von Strings ohne spezifischen Verwendungszweck stellt ausschließlich die Möglichkeit bereit, Eingaben die als nicht gültig erkannt werden abzulehnen. Eine Transformation in einen gültigen Zustand ist in der aktuellen Implementierung noch nicht möglich.

```
1 @SuppressWarnings({"Untainted"})
2 @EnsuresQualifierIf(
3     result = true,
4     expression = "#1",
5     qualifier = Untainted.class
6 )
7 public boolean isValid(String input) {
8     return getValidationErrors(input).isEmpty();
9 }
```

Listing 6.9: Markierung des Validierungsprozesses anhand der Annotationen des Checker Frameworks

Vermeidung von SQL-Injektionen

Im Hinblick auf die Ergänzung der API um eine Möglichkeit zur Vermeidung von SQL-Injektionen wird erkenntlich, dass je nach Verwendungszweck der Eingabe andere Herangehensweisen für die Umsetzung adäquater Funktionen nötig sind. Hier wird beschrieben, wie eine API verwendet werden kann, um SQL-Injektionen unter der Anwendung der entsprechenden Richtlinie zu vermeiden.

In dem in Listing 6.10 dargestellten Quellcode weist das Checker Framework darauf hin, dass es bei dem Aufruf der Methode `executeQuery()` in Zeile 5 möglicherweise zu einer SQL-Injektion kommen kann, da die Eingabe `input` aus dem `args`-Parameter der Main-Methode stammt und somit als `@Tainted` erkannt wird.

```
1 String input = args[0];
2 String sqlString = "SELECT * FROM users WHERE username = '"
3   + input + "'";
4 Statement stmt = connection.createStatement();
5 ResultSet resultSet = stmt.executeQuery(sqlString);
```

Listing 6.10: Quellcode in dem eine Verwundbarkeit für SQL-Injektionen durch das Checker Framework erkannt wird

Gemäß der Richtlinie *IDS00-J. Prevent SQL injection* sollen für die Vermeidung von SQL-Injektionen keine eigenen Funktionalitäten zur Validierung der Eingabe ge-

schrieben werden. Stattdessen sind bestehende APIs der jeweiligen Datenbankschnittstelle zu verwenden. Anders als bei der zuvor beschriebenen Validierung von Strings ohne spezifischen Verwendungszweck weisen diese APIs eine ungültige Eingabe üblicherweise nicht zurück, sondern maskieren sie so, dass es nicht zu einer SQL-Injektion kommen kann. Dementsprechend müssen auch die Funktionen einer API zur Validierung von Eingaben für die Integration in SQL-Abfragen anders konzipiert werden. Listing 6.11 zeigt einen Aufruf der in dem Rahmen des Proof of Concepts umgesetzten API, um SQL-Injektionen zu vermeiden.

```
1 String input = args[0];
2 PreparedStatement stmt = new SQLValidator().getSafeQuery(
3   connection,
4   "SELECT * FROM users WHERE username = ?",
5   input);
6 ResultSet resultSet = stmt.executeQuery();
```

Listing 6.11: Verwendung des API-basierten Ansatzes zur Vermeidung von SQL-Injektionen

Um eine richtlinienkonforme API bereitzustellen, wird in dem hier beschriebenen Beispiel eine Funktion definiert, welche die einzubindenden Eingaben und die SQL-Abfrage mit Platzhaltern für diese Eingaben gesondert annimmt. Intern verwendet die Funktion die nativen Funktionalitäten der Datenbankschnittstelle, um eine Abfrage zu erstellen, welche nicht für SQL-Injektionen anfällig ist. Dies ist ein wesentlicher Unterschied zu der zuvor beschriebenen Vorgehensweise von ESAPI, welche die Funktionalität für das Maskieren der Eingaben selbst implementiert. Wird die Eingabe hier nicht gesondert als Argument bei dem Aufruf der Funktion `getSafeQuery()` angegeben, sondern durch eine Konkatenation in die SQL-Abfrage integriert, generiert das Checker Framework eine Warnung. So garantiert dieser Ansatz, dass bei der Verwendung der bereitgestellten API SQL-Injektionen praktisch ausgeschlossen werden können und bietet eine einfache Verwendungsmöglichkeit und Hilfestellung für die Nutzung der jeweils relevanten Funktionen der Datenbankschnittstelle.

Bei der Vermeidung von SQL-Injektionen kommen jedoch weitere Schwierigkeiten für die Umsetzung einer entsprechenden API hinzu, je nachdem welche Datenbankschnittstellen und Frameworks der Entwickler verwendet. Im Hinblick auf das Object-Relational Mapping (ORM) Framework Hibernate sollten beispielsweise eigene Methoden bereitgestellt werden, welche eine Schnittstelle zu den Funktionalitäten von Hibernate bilden, um SQL- und HQL-Injektionen zu vermeiden. Entsprechende Funktionen sind in dem Proof of Concept implementiert und können nahezu identisch zu der zuvor beschriebenen Funktion `getSafeQuery()` aufgerufen werden.

Vermeidung von XML-Injektionen

Um zu prüfen, inwiefern eine API realisierbar ist, die auch komplexere Validierungsprozesse vereinfachen und unter der Hilfe statischer Code-Analyse ermöglichen kann, wurde mit dem Proof of Concept eine Funktionalität für die richtlinienkonforme Vermeidung von XML-Injektionen umgesetzt. Hierfür beschreibt die Richtlinie *IDS16-J. Prevent XML Injection* zwei Vorgehensweisen, die von dem Datentyp der in das XML-Dokument zu integrierenden Eingabe abhängig sind. Falls eine Eingabe eines

primitiven Datentyps, etwa eines Integers, für die Integration in das XML-Dokument erwartet wird, genügt es in diesem Fall sicherzustellen, dass es sich bei der Eingabe tatsächlich um einen Integer handelt und dass je nach Logik des Programmes relevante semantische Eigenschaften durch die Eingabe erfüllt werden, z.B. dass sie keine negativen Werte annehmen darf. Diese Validierung kann durch eine API analog zu dem zuvor beschriebenen `UniversalStringValidator` umgesetzt werden.

Für komplexere Eingaben wie Strings muss eine Validierung jedoch anhand einer Dokumenttypdefinition in der Form eines XML-Schemas umgesetzt werden, um festzustellen, ob das XML-Dokument unter Verwendung der Eingaben das XML-Schema erfüllt. So können Injektionen vermieden werden, bei denen die Eingabe zusätzliche XML-Tags enthält. Der in Listing 6.12 enthaltene Quellcode veranschaulicht, wie die Vermeidung einer XML-Injektion anhand eines Schemas durch die Verwendung der in dem Proof of Concept umgesetzten API erzielt werden kann. Der Inhalt der Datei xmlSchema.xsd wird in Listing 6.13 dargestellt.

```
1  public static void main(String[] args) {
2
3    // args[0] is "<price>1.0</price>"
4    String input = args[0];
5
6    // XML with placeholder for input
7    String xmlString =
8        "<item>" +
9          "<description>Widget</description>" +
10         "<price>500.0</price>" +
11         "<info>?</info>" +
12       "</item>";
13
14   // retrieve the schema for validation
15   File schema = new File("xmlSchema.xsd");
16
17   // 1. validate the semantics of the provided string first
18   UniversalStringValidator stringValidator =
19           new UniversalStringValidator.Builder(false)
20           .setCheckMinLength(1)
21           .setCheckMaxLength(100)
22           .build();
23
24   if(!stringValidator.isValid(input)) {
25       // error handling
26       System.err.println("Provided string was not valid");
27       return;
28   }
29
30   // 2. schema validation
31   // Checker Framework reports constructor call if user
        concatenates input into xmlString:
32   XMLValidator xmlValidator = new XMLValidator.Builder(
        xmlString, schema)
33           .addInputString(input)
```

```
34          .build();
35
36   // call prevented by exception if input is not valid:
37   // String validXML2 = xmlValidator.getValidatedXML();
38
39   if (xmlValidator.isValid()) {
40       System.out.println("XML is valid");
41       String validXML = xmlValidator.getValidatedXML();
42       sink(validXML);
43   } else {
44       System.err.println("XML is not valid");
45   }
46 }
47
48 public static void sink(@ValidatedXML String xml) {
49   // write xml to file
50 }
```

Listing 6.12: Verwendung des API-basierten Ansatzes zur Vermeidung von XML-Injektionen

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2 <xs:element name="item">
3   <xs:complexType>
4     <xs:sequence>
5       <xs:element name="description" type="xs:string"/>
6       <xs:element name="price" type="xs:decimal"/>
7       <xs:element name="info" type="xs:string"/>
8     </xs:sequence>
9   </xs:complexType>
10 </xs:element>
11 </xs:schema>
```

Listing 6.13: Inhalt der Datei xmlSchema.xsd

In diesem Beispiel soll der Wert der Variable `input` in das `<info>`-Tag des in `xmlString` gespeicherten XML-Dokuments eingefügt werden. Das XML-Dokument soll anschließend durch die Funktion `sink()` verwendet werden, welche anhand der Annotation `@ValidatedXML` ausschließlich validierte XML-Dokumente als Parameter akzeptiert. Da `input` in diesem Beispiel den Wert „<price>1.0</price>“ hat, kann es dazu kommen, dass so der ursprünglich in dem Dokument angegebene Preis überschrieben wird. Um solche XML-Injektionen zu vermeiden, wird in Zeile 32 die Klasse `XMLValidator` der API aufgerufen, welche das XML-Dokument mit Platzhaltern für die Eingaben in der Form von Fragezeichen, sowie die Eingaben separat und das Schema zur Validierung erhält. Der Validator erwartet als XML-Dokument zudem eine Variable, die als `@Untainted` gilt. So weist das SCAT den Entwickler auf einen Fehler hin, falls eine Eingabe zuvor in die Variable `xmlString` konkateniert wurde. Um sicherzustellen, dass eine semantische Validierung der Eingabe stattfindet,

akzeptiert die Funktion `XMLValidator.addInputString()` zudem ausschließlich validierte Eingaben, weshalb in den Zeilen 18 bis 28 eine zusätzliche Validierung anhand des `UniversalStringValidator` durchgeführt wird. Um den Entwickler weiterhin zu unterstützen generiert der `XMLValidator` außerdem eine Exception, falls eine ungleiche Anzahl an Platzhaltern und Eingaben vorhanden ist oder falls eine Eingabe des Datentyps String ohne die Angabe eines Schemas spezifiziert wurde.

Ähnlich wie bei dem zuvor beschriebenen `UniversalStringValidator` wird in diesem Beispiel anhand eines Aufrufes der Funktion `isValid()` in Zeile 39 überprüft, ob das XML-Dokument unter Verwendung der angegebenen Eingaben gültig ist oder nicht. Falls es gültig ist, stellt der `XMLValidator` das Dokument mit den eingefügten Eingaben über die Funktion `getValidatedXML()` bereit. Diese Rückgabe ist zudem als `@ValidatedXML` markiert und kann somit der Funktion `sink()` als Argument übergeben werden, ohne dass das Checker Framework vor diesem Aufruf warnt. Bei der Ausführung dieses konkreten Beispieles würde der Validator erkennen, dass das XML-Dokument unter Verwendung der Eingabe „<price>1.0</price>“ nicht gültig ist, da seine Struktur nicht mehr dem angegebenen Schema entspricht.

6.6.3 Erste Beobachtungen

Bei der Entwicklung des Proof of Concepts für die Bereitstellung einer API, die mit einem SCAT zusammenarbeitet, wurde erkenntlich, dass ein entsprechendes Analysewerkzeug verwendet werden kann, um fehlende Eingabevalidierungen oder falsche Verwendungen der API zu erkennen, während die API die Durchführung korrekter Validierungsprozesse stark vereinfacht. In erster Linie wird diese Vereinfachung erzielt, indem die API die Einhaltung je nach Verwendungszweck relevanter Secure Coding Richtlinien garantieren kann, ohne dass der Entwickler deren Details kennen und selbst anwenden muss. Zudem vereinfacht sie die Umsetzung korrekter Validierungen, da sie Funktionen bereitstellt, welche den Entwickler an die Validierung semantischer Eigenschaften erinnern, wie etwa die maximale Länge einer Eingabe oder ihre Nullwertfähigkeit, und garantieren, dass ausschließlich gültige Eingaben durch die API zurückgegeben werden, wie in dem Beispiel zur Vermeidung von SQL-Injektionen.

Verbesserungen zu dem hier beschriebenen Proof of Concept können etwa erzielt werden, indem weitere Validatoren für andere Verwendungszwecke von Eingaben implementiert oder die bestehenden Klassen um weitere Funktionalitäten ergänzt werden. Zudem sollten neue Annotationen, welche für den jeweiligen Verwendungszweck spezifisch sind, nach dem Vorbild der hier definierten Annotation `@ValidatedXML` implementiert und weitere Stub-Dateien bereitgestellt werden, um jeweils relevante Funktionen mit diesen Annotationen standardmäßig zu versehen. Für diese Annotationen könnten zudem eigene Checker implementiert werden, welche spezifischere Aussagen zu den jeweiligen Validierungsfehlern treffen können und den Entwickler anhand individualisierter Warnungen auf die Verwendung relevanter Funktionen der bereitgestellten API hinweisen. Die API, die neuen Annotationen und Stub-Dateien sowie das benötigte Werkzeug zur Durchführung der statischen Analysen sollten schlussendlich als vollwertiges SCAT zusammen ausgeliefert werden.

Im Hinblick auf unvermeidbare Faktoren des hier vorgestellten Ansatzes für die Verwendung einer API zur Eingabevalidierung, die sich potenziell negativ auf diesen Ansatz auswirken können, wurden bei der Entwicklung des Proof of Concepts weiterhin die folgenden Punkte erkenntlich.

1. Eine API zu konzipieren, die Funktionalitäten für die Umsetzung aller möglichen und relevanten Vorgehensweisen zur Eingabevalidierung umsetzt, ist anspruchsvoll und aufwändig. Falls der Entwickler beispielsweise im Rahmen einer allgemeinen Validierung eines Strings bestimmte Werte ausschließen möchte, die API aber keine Methoden für Blacklisting-Verfahren bereitstellt, müsste der validierte Wert durch die API zurückgegeben werden und anschließend noch einmal durch den Entwickler manuell im Hinblick auf die auszuschließenden Werte geprüft werden. In diesem Fall dürfte der zurückgegebene Wert zudem nicht als validiert gelten, da er diese letzte Anforderung noch nicht erfüllt. Weiterhin müsste eine API auch Funktionen für verschiedenste Anwendungsfälle wie beispielsweise die Nutzung von Frameworks wie Hibernate und ihre jeweiligen Funktionalitäten bereitstellen.

2. Der Entwickler erhält üblicherweise keinen Einblick in die Details des Validierungsprozesses, da der hierfür relevante Quellcode durch die API bereitgestellt wird. Somit ist der Lerneffekt bei der Verwendung dieses Ansatzes eher gering, da der Entwickler die konkreten Vorgehensweisen der Validierung beispielsweise nicht in Zukunft selbst anwenden oder für andere Programmiersprachen abstrahieren kann, für die ein solcher Ansatz nicht umgesetzt wurde.

3. In bestimmten Fällen kann eine API keine ausreichende Hilfestellung leisten. Die einzige korrekte Möglichkeit zur Validierung von E-Mail-Adressen ist beispielsweise, dem Anwender eine E-Mail zur Bestätigung zu senden. Da für die Umsetzung einer solchen Validierungsfunktionalität auch die Architektur der Software relevant ist, kann ein API-basierter Ansatz hier keine adäquate Hilfe leisten.

6.7 Ansatz 3: Generieren von Quellcode

In diesem Abschnitt wird der dritte näher betrachtete Ansatz beschrieben. Dieser ist mit den zuvor beschriebenen nicht-traditionellen Ansätzen von FixBugs und CogniCrypt vergleichbar, da hier der notwendige Quellcode für die Implementierung einer korrekten Eingabevalidierung teilautomatisiert generiert wird.

6.7.1 Konzept

Die relevanten Komponenten für das grundlegende Konzept dieses Ansatzes sind in Abbildung 6.6 dargestellt. Hierbei soll ein SCAT verwendet werden, um den Programmcode des Entwicklers auf Fehler bei der Eingabevalidierung zu überprüfen und die gefundenen Probleme anschließend in der verwendeten IDE anmerken. Anhand eines berichteten Problems kann der Entwickler anschließend einen Assistenten aufrufen, um den Quellcode zur Validierung der Eingabe unter der Einhaltung relevanter Secure Coding Richtlinien zu generieren. Je nach Verwendungszweck der Eingabe könnte der Quellcode dabei entweder vollautomatisch oder anhand von Benutzereingaben über den Assistenten erstellt werden. Gewissermaßen ähnelt dieser Ansatz damit einem komplexeren Quick-Fix. Ebenso wie bei der zuvor beschriebenen API-basierten Variante muss das hier verwendete SCAT ebenfalls in der Lage sein, den validierenden Programmcode erkennen zu können.

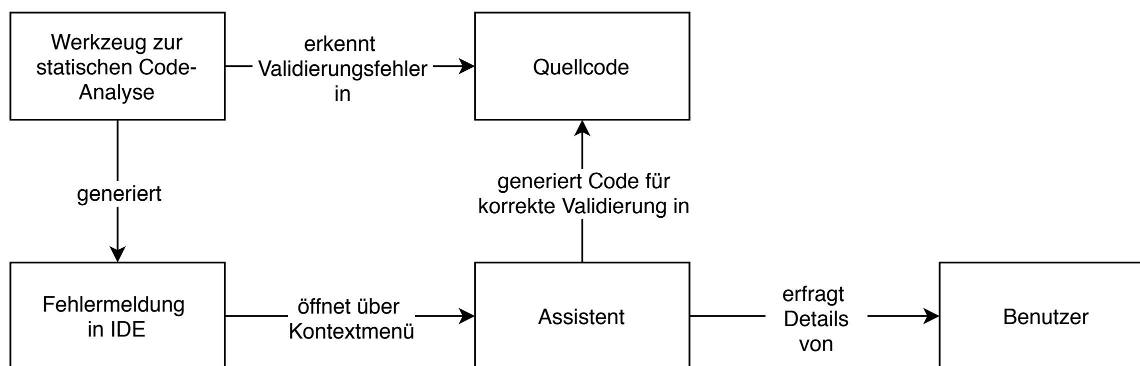

Abbildung 6.6: Komponenten für einen Ansatz zur interaktiven Generierung von Quellcode

Ein solcher Ansatz, bei dem der Quellcode generiert werden kann, hat mehrere Vorteile im Vergleich zu einer API-basierten Variante. Da der Code des Validierungsprozesses hier offenliegt, kann er, falls nötig, durch den Benutzer noch problemlos manuell um Funktionalitäten ergänzt werden, welche durch die Implementierung des Ansatzes selbst nicht abgedeckt werden. Anders als bei einer API erhält der Entwickler hier zudem Einblick in die Implementierungsdetails der Validierung, weshalb es zu einem deutlich besseren Lerneffekt kommen kann. Dieser kann auch verstärkt werden, indem der generierte Quellcode Kommentare enthält, welche die einzelnen Schritte des Validierungsprozesses erläutern. Ein interaktiver Assistent kann den Entwickler zudem sehr effektiv unterstützen, indem er beispielsweise eine Hilfestellung bei der Definition eines XML-Schemas für die Vermeidung von XML-Injektionen leisten oder bei dem Verfassen regulärer Ausdrücke helfen kann. Auch für Fälle, in denen ein Assistent den nötigen Quellcode nicht generieren kann, könnte er Unterstützung bieten, etwa indem er im Hinblick auf die Validierung von E-Mail-Adressen den Entwickler darauf hinweist, dass in diesem Fall eine E-Mail zur Bestätigung an die jeweilige Adresse gesendet werden sollte. Dies entspricht auch der Erkenntnisse nach Omar et al. (2012) sowie Barik et al. (2016), die besagen, dass ein interaktiver Ansatz für das Generieren des Quellcodes auf die individuellen Eigenheiten und Anforderungen des jeweiligen Verwendungszweckes anzupassen ist.

Für die Umsetzung dieses Konzepts sind gemäß der Erkenntnisse von Krüger et al. (2017) sowie B. Johnson et al. (2013) zwei verschiedene Vorgehensweisen möglich. Bei dem zu generierenden Quellcode kann es sich entweder um eine anzupassende Quellcodevorlage oder um eine möglichst vollständige Lösung handeln. Eine Vorlage kann den wesentlichen Ablauf zur korrekten Validierung der jeweiligen Eingabe veranschaulichen, muss aber durch den Entwickler an den eigenen Quellcode angepasst werden. Eine vollständige Lösung würde im Idealfall den Quellcode zur Validierung ergänzen, ohne dass weitere Anpassungen durch den Entwickler nötig sind. In beiden Fällen kann der generierte Quellcode sehr hilfreich sein, da er dem Entwickler effektiv dabei hilft, eine korrekte Eingabevalidierung umzusetzen. Die Entwicklung eines Ansatzes, der eine möglichst vollständige und somit dynamische Lösung generiert, ist deutlich komplexer als das Bereitstellen von statischen Vorlagen, da hierbei der Quellcode des Entwicklers automatisch analysiert werden muss, um relevante Informationen zu finden. Eine Lösung hätte jedoch die Vorteile, dass weitere Fehler bei der Anpassung des generierten Quellcodes durch den Entwickler besser vermieden werden

können und dass dem Entwickler zusätzlicher Aufwand erspart bleibt. Das Bereitstellen von Vorlagen hat jedoch einen besseren Lerneffekt, da der Entwickler sich hierbei konkreter mit dem Quellcode zur Validierung der Eingabe auseinandersetzen muss.

Dieser Ansatz und seine weiteren Details werden in Abschnitt 6.7.2 anhand eines implementierten Proof of Concepts am Beispiel der Validierung eines Strings mit allgemeinem Verwendungszweck und der Vermeidung von SQL-Injektionen veranschaulicht, da diese beiden Fälle wie zuvor beschrieben besonders exemplarisch sind und sich hier zudem gut eignen, um die Prinzipien für das Generieren einer Quellcodevorlage und einer vollständigen Lösung zu veranschaulichen. Eine Möglichkeit zur Vermeidung von XML-Injektionen wird anschließend nur kurz umrissen, da diese keine neuen Erkenntnisse liefert, sondern lediglich die vorherigen Beobachtungen bestätigt.

6.7.2 Funktionsweise an praktischen Beispielen

Um die Funktionsweise und Realisierbarkeit dieses Ansatzes besser veranschaulichen zu können wurde im Rahmen dieser Arbeit ein Proof of Concept implementiert. Diese Implementierung verwendet ebenso wie die zuvor beschriebene API-basierte Variante das Checker Framework, um Validierungsfehler in dem Quellcode des Entwicklers zu erkennen und zu melden. Der darauf aufbauende Assistent zur Generierung des Quellcodes wurde als Plugin für die IDE Intellij IDEA umgesetzt, um eine möglichst einfache Integration in den Arbeitsablauf des Entwicklers zu ermöglichen. Der Proof of Concept stellt Möglichkeiten bereit, um den Quellcode zur korrekten Validierung der in Tabelle 6.2 aufgelisteten Verwendungszwecke unter der Einhaltung der angegebenen Richtlinien zu generieren.

Verwendungszweck der Eingabe	*Eingehaltene Richtlinien*
String (allgemeiner Verwendungszweck)	**IDS01-J.** Normalize strings before validating them
Eingabe als Teil einer SQL-Abfrage (spezifischer Verwendungszweck)	**IDS00-J.** Prevent SQL injection
String als Teil eines XML-Dokuments (spezifischer Verwendungszweck)	**IDS16-J.** Prevent XML Injection

Tabelle 6.2: Umfang der Implementierung des Proof of Concepts zur Generierung von Quellcode

Im Folgenden wird anhand des implementierten Proof of Concepts veranschaulicht, wie der hier beschriebene Ansatz in der Praxis verwendet werden kann, um dem Entwickler bei der Umsetzung einer korrekten Validierung für Eingaben vom Datentyp `String` ohne spezifischen Verwendungszweck sowie bei der Vermeidung von SQL-Injektionen und XML-Injektionen zu helfen.

Validierung eines Strings mit allgemeinem Verwendungszweck
Identisch zu dem zuvor beschriebenen Beispiel im Rahmen der API-basierten Lösung erstellt das Checker Framework für den in Listing 6.14 dargestellten Quellcode eine

Warnung, da das Argument für den Aufruf der Methode `sink()` nicht validiert wurde. Durch das im Rahmen des Proof of Concepts umgesetzte Plugin bietet die IDE IntelliJ IDEA einen neuen Eintrag im Kontextmenü der Warnung, über den der Entwickler den Assistenten zur Generierung des Quellcodes aufrufen kann, siehe Abbildung 6.7.

```
50 public static void main(String[] args) {
51   String input = args[0];
52   sink(input);
53 }
54
55 public static void sink(@Untainted String input) {
56   // use only validated inputs
57 }
```

Listing 6.14: Quellcode ohne Eingabevalidierung eines Strings

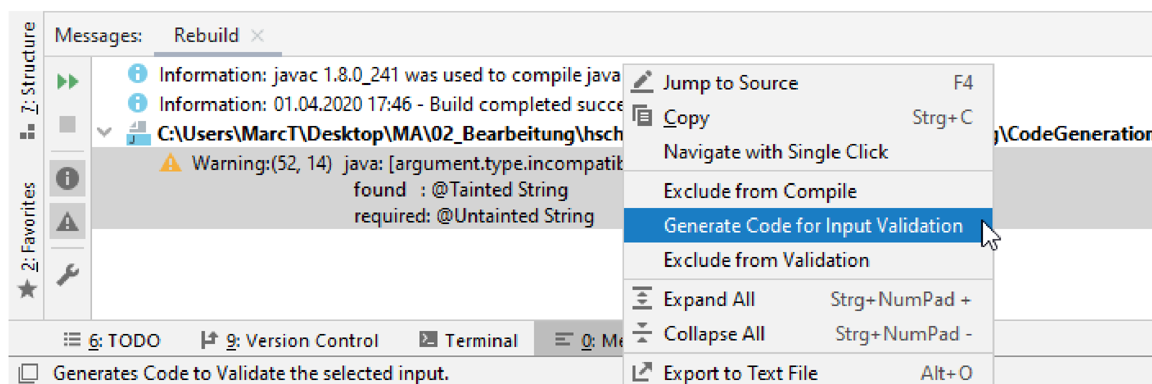

Abbildung 6.7: Eintrag in dem Kontextmenü einer Warnung zum Aufruf eines Assistenten für das Generieren von Quellcode zur Eingabevalidierung

Bei der Auswahl dieser Option wird der in dem Plugin enthaltene Assistent gestartet. Anhand der Warnung des Checker Frameworks kann das Plugin dabei erkennen, welches Element im Quellcode die Warnung ausgelöst hat, in diesem Fall das Argument `input` in Zeile 52, sowie welche Typen vorgefunden und welche Typen an dieser Stelle erfordert werden. Anhand der erforderlichen Typen kann das Plugin den Verwendungszweck der Eingabe ableiten. Da in diesem Beispiel der Typ `@Untainted String` erforderlich ist, stellt das Plugin dementsprechend fest, dass hier ein validierter String ohne spezifischen Verwendungszweck erwartet wird und öffnet den in Abbildung 6.8 dargestellten Assistenten.

Aufgrund der erforderlichen Typen wird der Verwendungszweck der Eingabe hier automatisch auf „Other (universal String)“ gesetzt und gemäß der Ergebnisse der Studie von B. Johnson et al. (2013, Seite 7) wird eine Vorschau des generierten Quellcodes auf der rechten Seite des Assistenten dargestellt. Der durch das Plugin abgeänderte Quellcode wird hierbei gelb markiert, während hinzugefügter Code grün hinterlegt wird. Für den hier erkannten Verwendungszweck der Eingabe generiert das Plugin die neue Methode `isValid()`, welche feststellt, ob die Eingabe gültig ist und mit Annotationen versehen wird, um dem Checker Framework zu signalisieren, dass

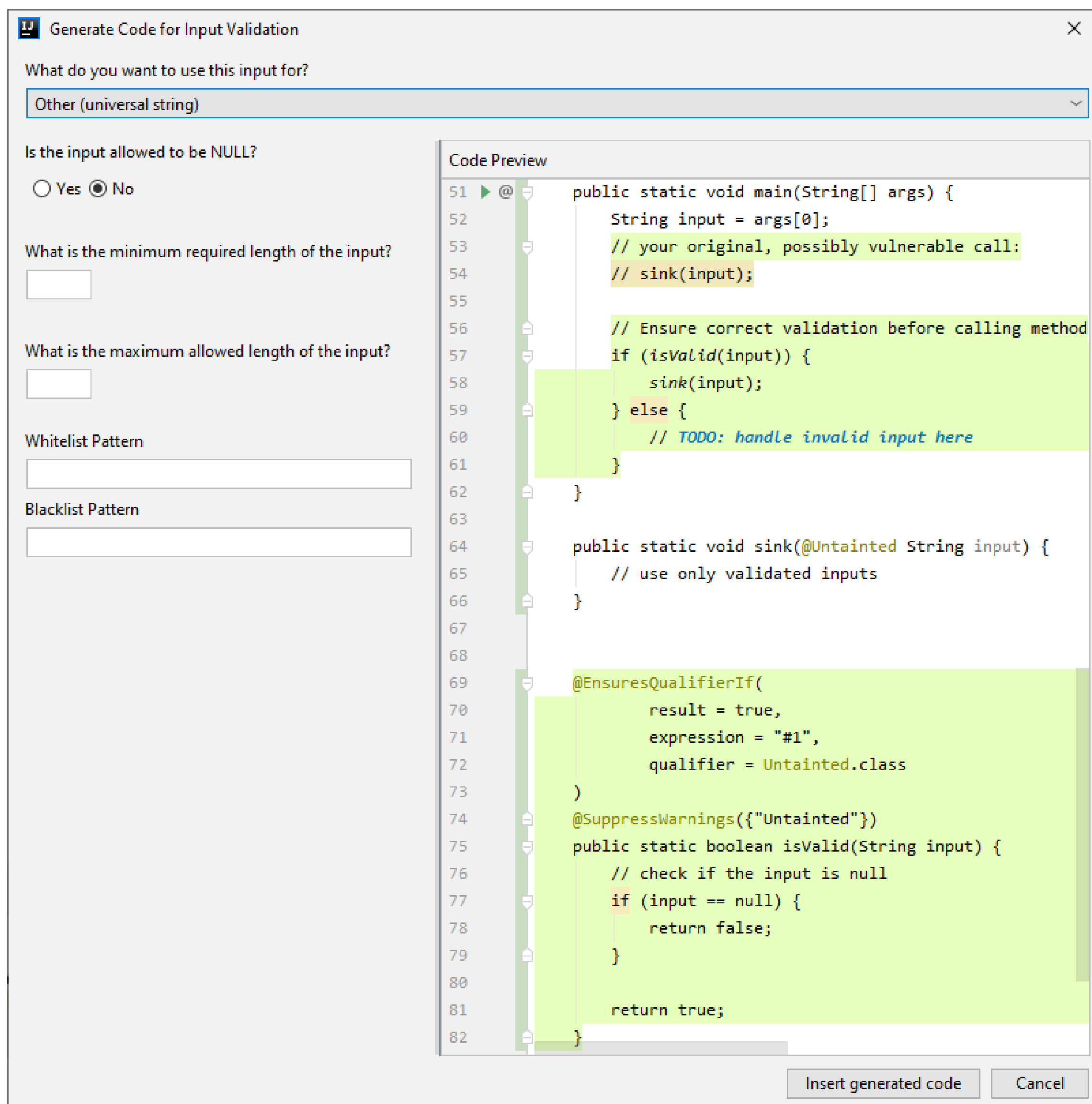

Abbildung 6.8: Assistent für das Generieren von Quellcode zur Validierung eines Strings ohne spezifischen Verwendungszweck

das Argument, welches dem ersten Parameter übergeben wird als „Untainted“ anzusehen ist, falls diese Methode den Wert `true` zurückgibt. Zusätzlich wird ein Aufruf der Methode mit der fraglichen Eingabe generiert und der ursprüngliche, potenziell verwundbare Aufruf der `sink()`-Methode wird auskommentiert und mit einem Kommentar zur Erklärung versehen. Auf der linken Seite des Assistenten kann der Entwickler für den jeweiligen Verwendungszweck der Eingabe individuelle Informationen bereitstellen, welche das Plugin verwendet, um den generierten Quellcode präziser auf die jeweiligen Anforderungen zuzuschneiden. In dem implementierten Proof of Concept kann an dieser Stelle für Strings ohne spezifischen Verwendungszweck festgelegt werden, ob die Eingabe den Wert `NULL` annehmen darf, ob eine minimale und maximale Länge existiert und es können reguläre Ausdrücke angegeben werden, die gemäß des Prinzips von White- und Blacklisting-Verfahren entweder erfüllt werden müssen oder nicht erfüllt werden dürfen. Bei jeder Änderung dieser Angaben wird

der generierte Quellcode automatisch angepasst und kann anschließend durch einen Klick auf „Insert generated code“ übernommen werden.

Vermeidung von SQL-Injektionen

Um einige Besonderheiten dieses Ansatzes zu veranschaulichen, vor allem die Unterscheidung zwischen dem Generieren einer Vorlage und dem Generieren einer vollständigen Lösung für ein Problem, wurde eine Möglichkeit implementiert, um SQL-Injektionen anhand des Plugins zu vermeiden. Damit der Assistent den Verwendungszweck der Eingabe automatisch erkennen kann, wurde zu diesem Zweck die neue Annotation `@SQL` definiert und die in Listing 6.15 dargestellte Stub-Datei für das Checker Framework erstellt.

```
1 import annotations.*;
2 import org.checkerframework.checker.tainting.qual.*;
3
4 package java.sql;
5
6 class Statement {
7     ResultSet executeQuery(@Tainted @SQL String sql);
8 }
```

Listing 6.15: Stub-Datei zur Erkennung einer Verwundbarkeit für SQL-Injektionen

Diese Stub-Datei wird hier genutzt, um die Methode `executeQuery()` der Klasse `java.sql.Statement` so für das Checker Framework zu markieren, dass sie ein Argument erwartet, welches mit der Annotation `@SQL` versehen ist. Standardmäßig ist der Parameter dieser Methode durch das Checker Framework mit der Annotation `@Untainted` versehen, die hier zusätzlich überschrieben wird. Für den in Listing 6.16 dargestellten Quellcode erstellt das Checker Framework dementsprechend eine Warnung, da dieser anfällig für SQL-Injektionen ist, siehe Abbildung 6.9.

```
12 public static void main(String[] args) {
13   try (Connection conn = DriverManager.getConnection(
14           dbUrl, "root", "root"
15   )) {
16       String username = args[0];
17       int id = Integer.parseInt(args[1]);
18       String sqlString = "SELECT * FROM users WHERE id = '"
19               + id + "' AND username = '" + username + "'";
20       Statement stmt = conn.createStatement();
21       ResultSet resultSet = stmt.executeQuery(sqlString);
22       print(resultSet);
23   } catch (SQLException e) {
24       e.printStackTrace();
25   }
26 }
```

Listing 6.16: Quellcode mit einer Verwundbarkeit für SQL-Injektionen

```
⚠ Warning:(21, 53)  java: [argument.type.incompatible] incompatible types in argument.
                     found   : @Unvalidated String
                     required: @SQL String
```

Abbildung 6.9: Warnung des Checker Frameworks für eine Verwundbarkeit für SQL-Injektionen

Wird der Assistent anhand dieser Warnung aufgerufen, so erkennt das Plugin basierend auf den erforderlichen Typen, dass die Eingabe für eine SQL-Abfrage gültig sein muss und wählt daher den Verwendungszweck „Use input as part of SQL Query“ automatisch in dem Assistenten aus. Der entwickelte Proof of Concept kann für dieses Problem entweder eine vollständige Lösung oder eine Quellcodevorlage, die durch den Entwickler an den eigenen Quellcode angepasst werden muss, generieren. Abbildung 6.10 zeigt die Verwendung des Assistenten zum Generieren einer Vorlage.

In der generierten Vorlage werden neben dem neuen Quellcode auch einige Kommentare zu seiner Erläuterung hinzugefügt, um die Anpassung der Vorlage durch den Entwickler zu vereinfachen, und der verwundbare Aufruf von `executeQuery()` wird auskommentiert. In diesem Beispiel muss der Entwickler anschließend den Namen der Variable für die Verbindung zu der Datenbank in Zeile 71 und den String der SQL-Abfrage in den Zeilen 71 und 72 sowie die einzusetzenden Benutzereingaben in Zeile 75 an seinen eigenen Quellcode anpassen.

Das Bereitstellen des Quellcodes als statische Vorlage ist vergleichsweise leicht umsetzbar, da hierbei der Kontext des Quellcodes des Entwicklers nur minimal analysiert werden muss. In der aktuellen Implementierung wird so lediglich identifiziert, welcher Funktion die durch die Warnung des SCAT markierte Variable als Argument übergeben wurde. In diesem Beispiel wird daher der Aufruf der Funktion `executeQuery()` der Klasse `java.sql.Statement` als zentraler Faktor für die Konkretisierung des durch die erforderte Annotation `@SQL` markierten Anwendungsfalles erkannt. So kann eine für diesen konkreten Fall passende Vorlage ausgewählt und ihr Quellcode in dem Programm des Entwicklers eingefügt werden.

Das Generieren einer dynamischen und vollständigen Lösung statt einer statischen Vorlage kann hingegen realisiert werden, indem der AST traversiert wird, um die für den zu generierenden Code relevanten Informationen im Quellcode des Entwicklers zu finden. In dem Proof of Concept wurde zur Vermeidung von SQL-Injektionen die Möglichkeit umgesetzt, eine vollständige Lösung generieren zu lassen. Zu diesem Zweck wird das Program Structure Interface (PSI) verwendet, welches durch IntelliJ IDEA bereitgestellt wird und eine Schnittstelle zur Traversierung des AST bietet. So kann ausgehend von dem Element im Quellcode, welches die Warnung durch das Checker Framework ausgelöst hat, der AST durchlaufen werden, um beispielsweise festzustellen, welche Variable die Verbindung zur Datenbank speichert oder aus welchen Komponenten sich der String für die Definition der SQL-Abfrage zusammensetzt. Basierend auf diesen Informationen kann daher wie in Abbildung 6.11 veranschaulicht der nötige Quellcode für eine korrekte Lösung vollautomatisch generiert werden.

In diesem Beispiel werden die Namen der Variablen für die Verbindung zur Datenbank und das Speichern der Ergebnisse der SQL-Abfrage automatisch erkannt und bei dem Generieren des Quellcodes verwendet. Zusätzlich wird die Deklaration des Strings, der die Abfrage definiert, untersucht, um eine parametrisierte Abfrage unter der Verwendung von Platzhaltern für die Eingaben zu definieren und die Eingaben

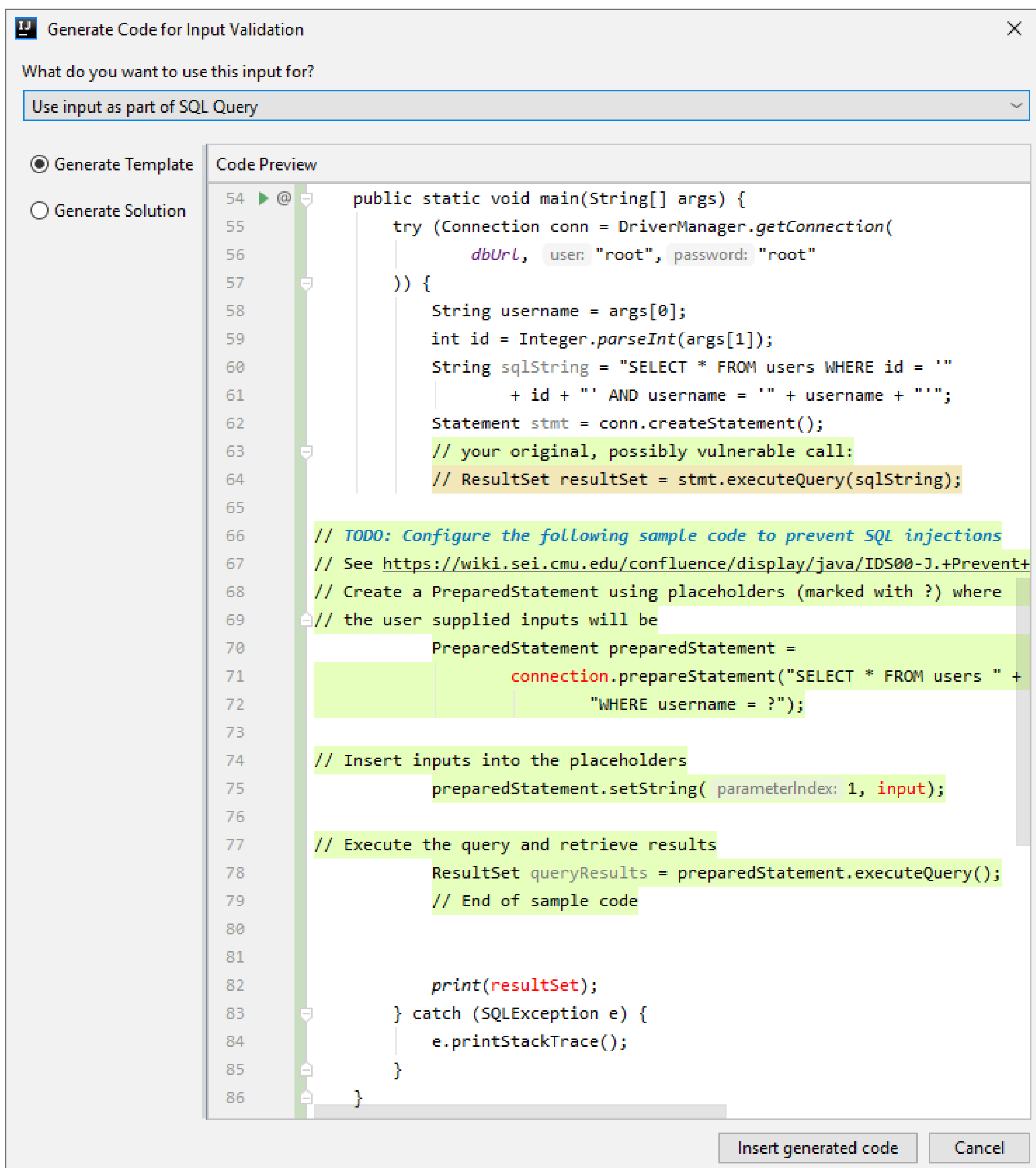

Abbildung 6.10: Verwendung des Assistenten für das Generieren einer Vorlage zur Vermeidung von SQL-Injektionen

anschließend gemäß der Richtlinie *IDS00-J. Prevent SQL injection* in den Zeilen 75 und 76 separat in die Abfrage einzufügen.

Das dynamische Generieren einer solchen, möglichst vollständigen Lösung ist insofern praktischer für den Entwickler, als dass er keine weiteren Anpassungen an dem neuen Quellcode vornehmen muss und Zeit einsparen kann. Allerdings ist die Implementierung dieser Vorgehensweise deutlich komplexer als das Bereitstellen von Vorlagen, da der AST je nach Aufbau des Quellcodes des Entwicklers anders traversiert werden muss, um die nötigen Informationen finden zu können. In der beigefügten Implementierung des Proof of Concepts wird der String, der die SQL-Abfrage definiert, beispielsweise nur innerhalb der Funktion gesucht, in der die Warnung des Checker

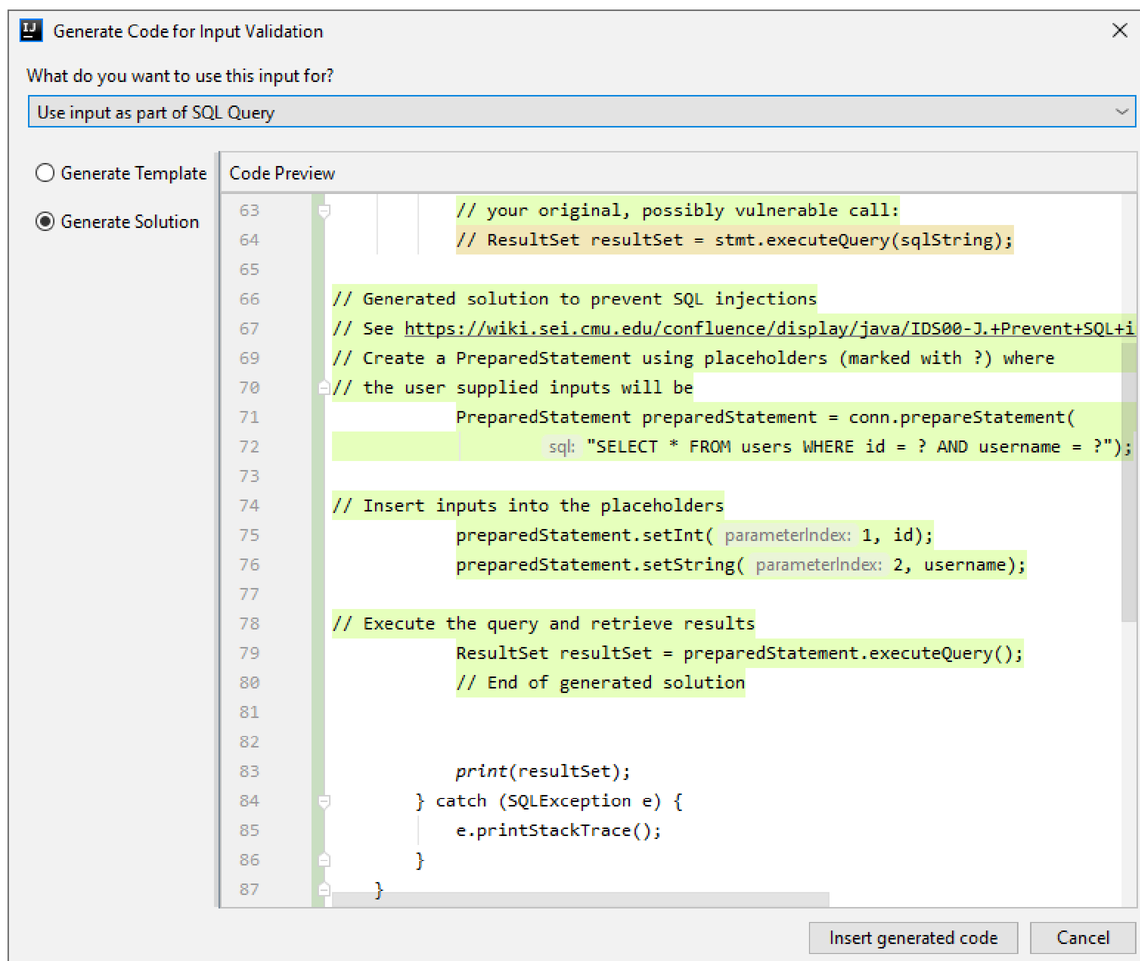

Abbildung 6.11: Verwendung des Assistenten für das Generieren einer Lösung zur Vermeidung von SQL-Injektionen

Frameworks aufgetreten ist. In einer vollständigen Implementierung dieses Ansatzes müsste das Plugin zusätzlich in der Lage sein, diese Definition auch außerhalb der Funktion zu finden, falls der String beispielsweise als Argument übergeben wurde. Weiterhin ist der Grad, in dem der Quellcode automatisiert generiert werden kann, abhängig von dem jeweiligen Verwendungszweck. Während für den hier angegebenen Fall der Vermeidung von SQL-Injektionen der Quellcode vollautomatisch generiert werden kann, müssen bei dem zuvor beschriebenen, allgemeinen Verwendungszweck von Strings zwingend Informationen für die semantische Validierung durch den Entwickler angegeben werden, da diese nicht automatisch inferiert werden können.

Vermeidung von XML-Injektionen

Eine Funktionalität, um den Quellcode für die Vermeidung von XML-Injektionen gemäß der Richtlinie *IDS16-J. Prevent XML Injection* zu generieren wurde wie in Abbildung 6.12 dargestellt ebenfalls im Rahmen des Proof of Concept umgesetzt. Da bei der Implementierung dieser Funktionalität keine zusätzlichen Erkenntnisse sichtbar wurden, wird diese hier nicht näher beschrieben. Allerdings bestätigt diese Ergänzung den ersten Eindruck, dass ein solcher Ansatz sinnvoll und realisierbar ist.

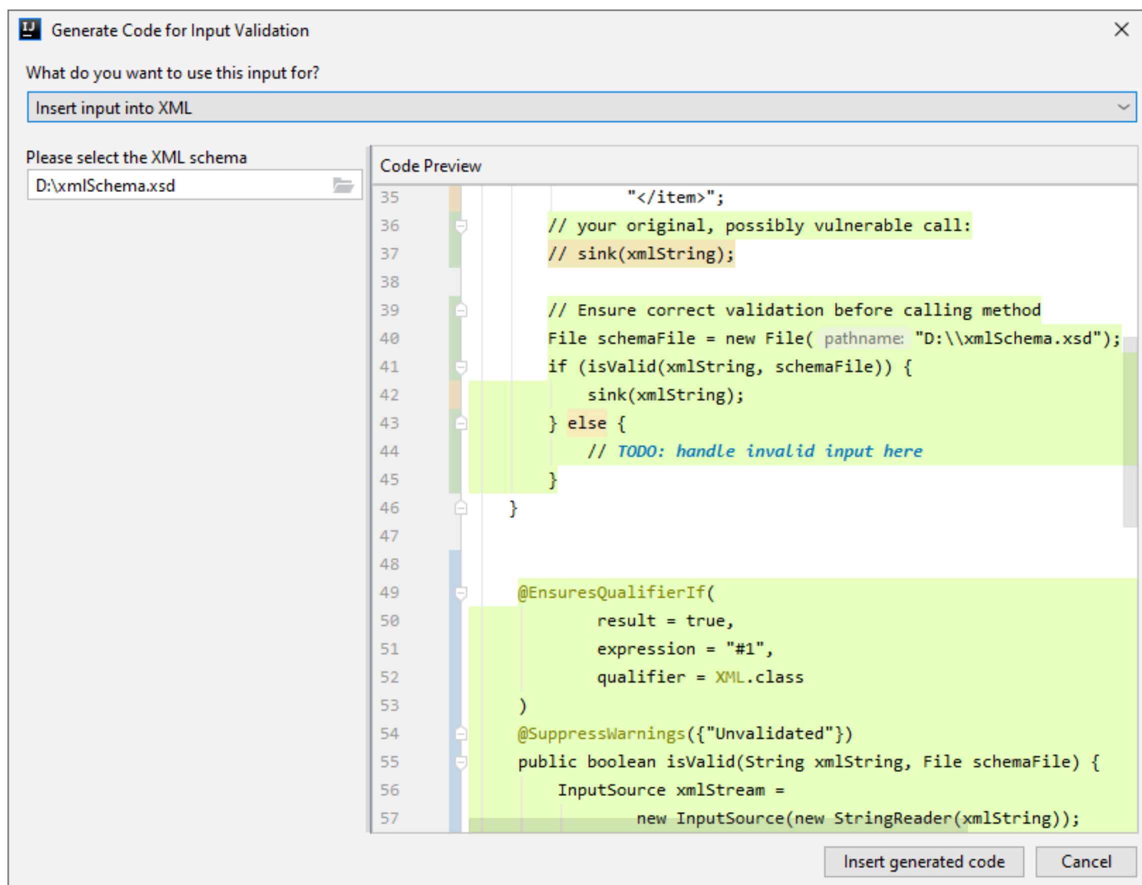

Abbildung 6.12: Verwendung des Assistenten für das Generieren einer Lösung zur Vermeidung von XML-Injektionen

6.7.3 Erste Beobachtungen

Anhand des implementierten Proof of Concepts wird ersichtlich, dass ein Ansatz, der ein SCAT ergänzt und als Plugin für eine IDE umgesetzt wird, um den Quellcode für eine Vorlage oder eine vollständige Lösung zur Behebung eines Validierungsfehlers zu generieren, eine sinnvolle Hilfestellung leisten kann. Im Gegensatz zu einem API-basierten Ansatz ist die Umsetzung eines solchen Plugins allerdings deutlich komplexer. Dieses Konzept ist jedoch dynamischer, da hierbei die Details der Implementierung des Validierungsprozesses offenliegen und somit einfacher durch den Entwickler angepasst werden können. Zudem entsteht durch den einsehbaren Quellcode ein besserer Lerneffekt. Dies gilt insbesondere für die Variante, in der eine Vorlage erstellt wird, da der Entwickler sich zwangsweise näher mit dem hierbei generierten Quellcode auseinandersetzen muss, um ihn in das eigene Programm zu integrieren. Das Bereitstellen einer vollständigen Lösung erspart dem Entwickler hingegen vor allem Aufwand und vermeidet weitere Fehler, weil der generierte Quellcode nicht zusätzlich angepasst werden muss.

Anders als bei dem API-basierten Konzept kann ein solches Plugin auch anhand des interaktiven Assistenten eine deutlich effektivere und dynamischere Hilfestellung bei der Umsetzung korrekter Validierungsprozesse leisten. Die Implementierung könnte beispielsweise um einen neuen Verwendungszweck für die Validierung von E-Mail-Adressen erweitert werden, den der Entwickler in dem Assistenten auswählen kann.

Dieser könnte den Entwickler anschließend darauf hinweisen, dass für diesen Fall eine E-Mail zur Bestätigung an die Adresse gesendet werden sollte, anstatt neuen Quellcode zu generieren. Falls das Plugin beispielsweise noch nicht um die Möglichkeit zur Ergänzung des Quellcodes für einen bestimmten Verwendungszweck erweitert wurde, könnte es zudem anhand des Assistenten dennoch eine grundlegende Beschreibung der anzuwendenden Vorgehensweise liefern. Im Hinblick auf die Vermeidung von SQL-Injektionen unter der Verwendung von Frameworks wie Hibernate könnte das Plugin den Entwickler etwa anhand des Assistenten textuell auf die wesentlichen Schritte, wie etwa die Verwendung der nativen Funktionalitäten des Frameworks zur Parametrisierung der SQL-Abfrage, hinweisen und ähnlich wie herkömmliche SCATs auf relevante Secure Coding Richtlinien verweisen, falls eine Möglichkeit für das Generieren des Quellcodes noch nicht implementiert wurde.

Der im Rahmen dieser Arbeit implementierte Proof of Concept bietet viel Verbesserungspotenzial. Zunächst muss das Checker Framework in der aktuellen Implementierung in das Projekt des Entwicklers eingebunden werden, um das Plugin nutzen zu können. Stattdessen sollte das Checker Framework als eigenes Plugin für die IDE oder als Teil des hier entwickelten Plugins umgesetzt werden, damit es nicht manuell eingebunden werden muss[24]. Da dieser Ansatz dem des SCAT CogniCrypt gewissermaßen ähnelt, sollte zudem untersucht werden, inwiefern der eigentliche Prozess der Codegenerierung hier ebenfalls durch die Verwendung eines Variabilitätsmodells umgesetzt werden sollte (Krüger et al., 2017, Seite 2). Zudem muss die Gebrauchstauglichkeit des aktuell implementierten Assistenten näher betrachtet werden. Beispielsweise sollte es in Betracht gezogen werden, die Vorschau des generierten Quellcodes nicht in einem eigenen Fenster, sondern direkt in dem Quellcode des Entwicklers darzustellen. Details in der Implementierung sollten zudem verbessert werden, indem die generierte Methode `isValid()` für die Validierung von Strings mit allgemeinem Verwendungszweck beispielsweise auch eine Liste an Validierungsfehlern produzieren sollte, damit der Quellcode des Entwicklers präziser auf die verschiedenen auftretenden Fehler reagieren kann. Weiterhin müssen bei einer umfassenden Implementierung dieses Ansatzes zusätzliche, individuelle Hilfestellungen für weitere Verwendungszwecke von Eingaben und die Einhaltung der jeweils relevanten Richtlinien anhand des Assistenten sowie Möglichkeiten für das Generieren entsprechenden Quellcodes umgesetzt werden. Hierfür bietet es sich an, das Plugin zunächst auf das Bereitstellen passender Quellcode-Vorlagen auszulegen und sinnvolle Vorlagen je nach Verwendungszweck zu entwickeln, da diese Vorgehensweise deutlich einfacher umzusetzen ist als das Generieren vollständiger Lösungen. Anschließend könnte das Plugin um die Möglichkeit ergänzt werden, den korrekten Quellcode für den jeweiligen Anwendungsfall möglichst vollautomatisch generieren zu lassen. Als wesentliche Schwierigkeiten dieses Ansatzes können anhand der Beobachtungen bei der Entwicklung des Proof of Concepts daher die folgenden Punkte festgehalten werden.

1. Die umfassende Implementierung dieses Ansatzes ist vergleichsweise komplex. Das gilt insbesondere für die Umsetzung einer Variante, in welcher der Quellcode für eine vollständige Lösung des jeweiligen Problems möglichst vollautomatisch bereitgestellt werden soll.

[24] https://github.com/dovchinnikov/intellij-checker-framework/issues/3#issuecomment-154524659, aufgerufen am 06.04.2020

2. Anders als die API-basierte Variante ist die konkrete Implementierung des hier beschriebenen Ansatzes für das Generieren von Quellcode zwangsweise von der jeweiligen IDE, für die das Plugin entwickelt wird, abhängig.

Kapitel 7

Evaluierung

In diesem Kapitel werden die zuvor entwickelten Vorgehensweisen für die Anwendung von Secure Coding Richtlinien zur Umsetzung korrekter Eingabevalidierungen evaluiert. Hierfür wird zunächst ein Kriterienkatalog definiert, der festhält, wodurch sich besonders sinnvolle Konzepte zu diesem Zweck auszeichnen und anhand dessen die entwickelten Ansätze anschließend bewertet werden. So wird festgestellt, inwiefern sich diese Ansätze für die praktische Verwendung eignen und in der folgenden Forschung wieder aufgegriffen werden sollten. Die als Proof of Concept umgesetzten Implementierungen werden hier nicht evaluiert, da diese lediglich zur Veranschaulichung der Funktionsweise und der Realisierbarkeit des jeweiligen Ansatzes dienen und nicht im Mittelpunkt dieser Arbeit stehen.

7.1 Definition des Kriterienkatalogs

Der in Tabelle 7.1 dargestellte Kriterienkatalog beschreibt die verschiedenen Kriterien, durch welche sich insbesondere die Effektivität und Effizienz eines Ansatzes für die Unterstützung des Entwicklers bei der Anwendung von Secure Coding Richtlinien definieren. Diese beiden Faktoren sind maßgeblich dafür, ob der fragliche Ansatz in der Praxis durch den Entwickler akzeptiert wird. Die einzelnen Kriterien ergeben sich dabei aus der zuvor durchgeführten Recherche zu den nicht-traditionellen Ansätzen für die Verwendung statischer Code-Analyse sowie der Entwicklung der neuen Ansätze im Rahmen dieser Arbeit.

Kennung	***Kriterium***	***Kurzbeschreibung***
		Effektivität
K1	Einhaltung von Richtlinien	Inwiefern kann der Ansatz sicherstellen, dass die in dem jeweiligen Kontext relevanten Richtlinien beachtet werden?
K2	Fehlervermeidung	Inwiefern können auch bei der Verwendung des Ansatzes dennoch Fehler auftreten?
K3	Vollständigkeit	Inwiefern kann der Ansatz eine möglichst vollständige Lösung für das Problem der Eingabevalidierung umsetzen?

K4	False Positives / False Negatives	Inwiefern ist der Ansatz für False Positives und False Negatives anfällig?
		Effizienz
K5	Erforderte Vorkenntnisse	Benötigt der Entwickler Vorkenntnisse auf dem Gebiet der sicheren Programmierung für die Verwendung des Ansatzes?
K6	Anpassung des Arbeitsablaufs	Muss der Arbeitsablauf des Entwicklers für die Verwendung des Ansatzes angepasst werden?
K7	Praktischer Aufwand	Wie viel Aufwand muss der Entwickler einmalig betreiben, um den Ansatz nutzen zu können?
K8	Kognitiver Aufwand	Inwiefern entsteht durch den Ansatz zusätzlicher kognitiver Aufwand bei dem Entwickler während der Arbeit?
K9	Zeitersparnis	Inwiefern reduziert der Ansatz den zeitlichen Aufwand für die Einhaltung relevanter Richtlinien?
		Sonstige Kriterien
K10	Allgemeine Restriktionen	Welche allgemeinen Restriktionen für die Verwendung des Ansatzes existieren?
K11	Motivation	Inwiefern motiviert der Ansatz den Entwickler dazu, die nötigen Secure Coding Richtlinien einzuhalten?
K12	Lerneffekt	Inwiefern bewirkt die Verwendung des Ansatzes einen Lerneffekt bei dem Entwickler?
K13	Flexibilität	Inwiefern kann die durch den Ansatz bereitgestellte Lösung angepasst werden?

Tabelle 7.1: Kriterienkatalog zur Evaluierung der entwickelten Ansätze

7.2 Evaluierung der entwickelten Ansätze

Um die entwickelten Ansätze zu bewerten werden sie in diesem Abschnitt im Hinblick auf die einzelnen Kriterien des zuvor aufgestellten Kriterienkatalogs betrachtet. So soll festgestellt werden, welche der Ansätze in der weiteren Forschung weiterentwickelt werden sollten und inwiefern sie in der Praxis voraussichtlich sinnvoll für die Anwendung der jeweils relevanten Secure Coding Richtlinien verwendet werden können. Der zuerst beschriebene Ansatz für die Verwendung von Annotationen, um bei der Anwendung der Richtlinien zu helfen, wird an dieser Stelle nicht evaluiert, da bereits bei seiner theoretischen Betrachtung konzeptionelle Faktoren erkenntlich wurden, aufgrund derer sich diese Herangehensweise zu diesem Zweck nicht eignet.

7.2.1 Effektivität

Die Kriterien der Kategorie „Effektivität“ befassen sich damit, inwiefern der jeweilige Ansatz es für den Entwickler vereinfacht, die relevanten Richtlinien einzuhalten und

ist somit maßgeblich dafür, ob der Ansatz bei der Entwicklung sicherer Software helfen kann.

K1 Einhaltung von Richtlinien
Das erste Kriterium befasst sich damit, inwiefern der Ansatz den Entwickler bei der Anwendung der in dem jeweiligen Kontext relevanten Richtlinien unterstützen kann. Der API-basierte Ansatz kann hierbei vor allem implizit helfen, indem die einzelnen Funktionen der API so konzipiert werden, dass sie die jeweiligen Richtlinien garantiert einhalten. Im Hinblick auf die Validierung von Strings ohne spezifischen Verwendungszweck kann beispielsweise garantiert werden, dass eine Normalisierung vor der Validierung durchgeführt wird, während für die Vermeidung von SQL-Injektionen ausschließlich Funktionen bereitgestellt werden, die eine SQL-Injektion unterbinden. Im Hinblick auf den Ansatz für das Generieren des relevanten Quellcodes werden die jeweiligen Richtlinien expliziter eingehalten, indem der Code für den korrekten Validierungsprozess offen einsehbar in das Programm des Entwicklers integriert wird.

Beide Ansätze können den Entwickler ausschließlich bei der Anwendung von Secure Coding Richtlinien unterstützen, welche sich mit der Implementierung eines korrekten Validierungsprozesses befassen. Der Umfang und die Qualität, in der sie hierbei helfen, hängt zwar von der Implementierung des jeweiligen Ansatzes ab, prinzipiell können beide Varianten aber dieselben Richtlinien durchsetzen. Die Ansätze können allerdings nicht bei der Anwendung von Richtlinien helfen, die sich nicht direkt auf die Implementierung des Validierungsprozesses, sondern auf die Fehlerfindung beziehen, wie beispielsweise *IDS11-J. Perform any string modifications before validation* oder *IDS14-J. Do not trust the contents of hidden form fields*. Diese Aufgabe fällt weiterhin dem als Grundlage für die Umsetzung des jeweiligen Ansatzes verwendeten SCAT zu.

K2 Fehlervermeidung
Dieses Kriterium befasst sich mit der Frage, inwiefern Validierungsfehler trotz der Verwendung eines Ansatzes dennoch auftreten können. Die wesentliche Vermeidung von Fehlern bei der Umsetzung des jeweiligen Validierungsprozesses wird sowohl durch den API-basierten Ansatz, als auch durch den Ansatz für das Generieren von Quellcode gleichermaßen erreicht, da in beiden Fällen eine Hilfestellung bei der Anwendung derselben Richtlinien geleistet werden kann. Der wesentliche Unterschied beider Ansätze im Hinblick auf dieses Kriterium ergibt sich daraus, dass der Programmcode zur Validierung bei dem Generieren des Quellcodes offenliegt, während dies bei der API-basierten Variante nicht der Fall ist. Dies führt dazu, dass der Validierungsprozess hier durch den Entwickler abgeändert werden kann, wodurch Verwundbarkeiten entstehen können. Im Hinblick auf die Validierung von Strings ohne spezifischen Verwendungszweck könnte der Entwickler beispielsweise die Funktionalität zur Normalisierung der Eingabe aus dem generierten Quellcode entfernen und der somit entstandene Fehler könnte durch den hier beschriebenen Ansatz nicht garantiert vermieden werden.

K3 Vollständigkeit
Entsprechend der Beobachtungen aus Abschnitt 6.2.3 ist anhand des Kriteriums *K3 Vollständigkeit* zu betrachten, inwiefern der Ansatz nicht nur bei der Anwendung isolierter Richtlinien, sondern auch im größeren Kontext der Thematik der Eingabevalidierung hilfreich sein kann, beispielsweise indem er für allgemeine sowie spezifische

Verwendungszwecke einer Eingabe gleichermaßen hilft oder dem Entwickler bei der Einhaltung von Details und Abwandlungen hilft, die nicht durch Richtlinien explizit festgehalten werden.

Die beiden entwickelten Ansätze sind dazu in der Lage, sowohl bei der Umsetzung korrekter Validierungsprozesse für Eingaben mit allgemeinen als auch mit spezifischen Verwendungszwecken zu helfen. Dies wurde im Rahmen der Implementierung der Proof of Concepts für die Validierung von Strings mit allgemeinem Verwendungszweck und der Funktionalität für die Vermeidung von SQL-Injektionen erkenntlich.

Der Ansatz für das Generieren des Quellcodes ist durch die Verwendung des interaktiven Assistenten allerdings besser dazu in der Lage, den Entwickler bei der Umsetzung von komplexeren Validierungsprozessen zu unterstützen, da dieser Assistent beispielsweise auf Feinheiten hinweisen oder besser an die Prüfung bestimmter Eigenschaften bei der Validierung von Strings erinnern kann. Der API-basierte Ansatz hilft dem Entwickler hingegen lediglich über die bereitgestellten Funktionen, ihre Dokumentation und eventuelle Exceptions. Ein allgemeines Problem beider Ansätze ist allerdings, dass sie nur dazu in der Lage sind, eine vollständige Lösung für das jeweilige Problem bereitzustellen. Falls der Entwickler einen Validierungsprozess für einen String bereits implementiert hat, der jedoch keine Normalisierung der Eingabe durchführt, können die Ansätze in der hier beschriebenen Form keine effektive Unterstützung zur Korrektur bieten, sondern lediglich durch das vollständige Ersetzen der bereits existierenden Validierung helfen.

K4 False Positives / False Negatives
Dass False Positives und False Negatives bei der Verwendung statischer Code-Analysen nicht zu vermeiden sind, wurde zuvor in Abschnitt 4.3 herausgestellt. Entsprechend der hier festgehaltenen Beobachtungen ist die Vermeidung von False Negatives an dieser Stelle dabei wichtiger als die Vermeidung von False Positives. Das Kriterium *K4 False Positives / False Negatives* dient dazu festzuhalten, ob und wie diese Fehlertypen im Rahmen der entwickelten Ansätze vermehrt oder verringert auftreten.

Beide Fehlertypen können dementsprechend bei den beiden Ansätzen gleichermaßen auftreten, da diese von dem jeweils verwendeten SCAT für die Erkennung der Validierungsfehler produziert werden. False Negatives treten bei der hier veranschaulichten Funktionsweise auf, da das jeweilige SCAT nicht selbst erkennen kann, an welchen Stellen im Quellcode beispielsweise ein validiertes XML-Dokument oder ein validierter String erwartet wird. Dies muss durch den Entwickler anhand entsprechender Annotationen markiert werden. Falls diese Annotationen nicht an den jeweiligen Stellen im Quellcode verwendet werden, können False Negatives nicht vermieden werden.

7.2.2 Effizienz

Die Kategorie „Effizienz“ beschreibt Kriterien, an denen festgestellt werden kann, inwiefern der Aufwand für die Anwendung der relevanten Richtlinien durch die Nutzung des Ansatzes verringert wird oder inwiefern zusätzlicher Aufwand anfällt. Entsprechend der Ergebnisse von Xie et al. (2012, Seite 2715) sind die hier aufgelisteten Kriterien somit maßgeblich dafür, ob der Ansatz durch den Entwickler akzeptiert und verwendet wird.

K5 Erforderte Vorkenntnisse

Dieses Kriterium befasst sich damit, in welchem Maße für die Verwendung des jeweiligen Ansatzes Vorkenntnisse hinsichtlich der sicheren Programmierung nötig sind. Dies ist insbesondere relevant, da viele Entwickler üblicherweise nicht über wesentliche Kenntnisse zu dieser Thematik verfügen (Chess & West, 2007, Seite 59).

Die Verwendung beider Ansätze erfordert keine besonderen Vorkenntnisse hinsichtlich der sicheren Programmierung in Bezug auf die Eingabevalidierung, da das verwendete SCAT den Entwickler in beiden Fällen auf verschiedene Fehler hinweist und der jeweilige Ansatz die Vorgehensweise zur Vermeidung des Fehlers vollständig bereitstellen kann. Lediglich für die Vermeidung von False Negatives können entsprechende Vorkenntnisse hilfreich sein, da ein für diese Problematik sensibilisierter Entwickler so möglicherweise eher darauf achtet, die nötigen Annotationen für die Markierung von Senken einzufügen.

K6 Anpassung des Arbeitsablaufs

Gemäß dieses Kriteriums muss betrachtet werden, inwiefern sich die Verwendung des Ansatzes auf den üblichen Arbeitsablauf des Entwicklers auswirkt. Bei beiden Ansätzen wird keine wesentliche Anpassung benötigt. Die Durchführung der statischen Analyse kann, wie durch das jeweilige Proof of Concept veranschaulicht, in den Kompilationsprozess eingebunden werden, während der entsprechende Ansatz direkt während Implementierung der Software verwendet werden kann. Im Falle der API-basierten Variante müssen lediglich die Funktionen der API aufgerufen werden, während der Assistent bei dem Ansatz zur Generierung des Quellcodes direkt in der jeweiligen IDE aufgerufen werden kann und sich effektiv wie ein komplexerer Quick-Fix verhält. Externe Programme müssen bei keinem der beiden Ansätze verwendet werden, außer falls das Konzept zum Generieren des Quellcodes unabhängig von der IDE als eigenständiges Programm umgesetzt wird.

K7 Praktischer Aufwand

K7 Praktischer Aufwand hält fest, wie aufwändig das Aufsetzen und Erlernen der Verwendung des jeweiligen Ansatzes ist. Der einmalige praktische Aufwand für das Aufsetzen des Konzeptes zur Quellcodegenerierung fällt besonders gering aus, da der Entwickler bei einer vollständigen Implementierung dieser Variante lediglich das Plugin installieren muss, welches sowohl das SCAT als auch den Assistenten beinhaltet. Der Aufwand für die Verwendung der API-basierten Variante fällt etwas höher aus, da hier sowohl das SCAT aufgesetzt werden muss, als auch die API in das Softwareprojekt eingebunden werden muss.

Der Aufwand für das Erlernen der Verwendung des jeweiligen Ansatzes fällt in beiden Fällen gering aus. Im Hinblick auf die Verwendung des API-basierten Ansatzes muss der Entwickler sich lediglich mit der korrekten Nutzung der bereitgestellten Funktionen auseinandersetzen, wohingegen der Assistent des Ansatzes für das Generieren des Quellcodes den Entwickler zusätzlich anleiten kann. Um eine möglichst vollständige Vermeidung von False Negatives zu erreichen, muss der Entwickler sich in beiden Fällen zudem über die Verwendung von Annotationen zur Markierung von Senken informieren, da diese Funktionalität für Fälle, die nicht fest definiert sind, nicht automatisch durch ein SCAT bereitgestellt werden kann.

K8 Kognitiver Aufwand
Dieses Kriterium befasst sich mit dem zusätzlichen kognitiven Aufwand, der durch die Verwendung des Ansatzes entsteht. Eine Auseinandersetzung mit diesem Faktor ist notwendig, da der Entwickler durch die eigentliche Programmierung der Software bereits kognitiv ausgelastet ist und der Ansatz den Entwickler nicht von seiner eigentlichen Aufgabe abhalten oder diese erschweren darf, sondern ihn bei ihrer Bearbeitung unterstützen muss (Zhu et al., 2013, Seite 2).

Durch beide Ansätze fällt ein sehr geringer zusätzlicher kognitiver Aufwand an, da die wesentlichen Details der Implementierung des nötigen Validierungsprozesses durch den API-basierten Ansatz abstrahiert werden und durch die Variante zum Generieren des Quellcodes automatisch bereitgestellt werden. Ein zusätzlicher geringer Aufwand entsteht allerdings auch dadurch, dass der Entwickler Senken aktiv in seinem Programm mit Annotationen versehen muss.

K9 Zeitersparnis
Die Zeitersparnis durch die Anwendung der einzelnen Ansätze ist relevant, da Zeitdruck durch Fristen ein zentraler Grund dafür ist, warum bestimmte Funktionalitäten, wie etwa die korrekte Validierung von Eingaben, nicht implementiert werden (Chess & West, 2007, Seite 59). Beide Ansätze können den nötigen Zeitaufwand für die Umsetzung der Validierungsmechanismen grundsätzlich verringern, da der Entwickler sich bei ihrer Verwendung nicht mehr selbst um die Implementierungsdetails kümmern muss. Zusätzlich wird auch der zeitliche Lernaufwand für ein Verständnis der korrekten Vorgehensweise reduziert.

7.2.3 Sonstige Kriterien

Diese Kategorie enthält Kriterien, die sich nicht direkt darauf auswirken, wie effektiv und effizient der jeweilige Ansatz bei der Einhaltung der nötigen Secure Coding Richtlinien helfen kann.

K10 Allgemeine Restriktionen
Die Verwendung der API-basierten Variante verfügt über keine wesentlichen Einschränkungen, da die API und das verwendete SCAT lediglich in das Softwareprojekt eingebunden werden müssen. Für den Ansatz zur Generierung des Quellcodes besteht hingegen eine deutlich striktere Restriktion, da das Plugin von der jeweiligen IDE abhängig ist, für die es implementiert wurde. Falls kein Plugin für die verwendete IDE existiert, kann der Ansatz in der hier beschriebenen Form nicht verwendet werden. Alternativ kann statt einem Plugin ein eigenständiges Programm umgesetzt werden, bei dessen Verwendung der Arbeitsablauf des Entwicklers jedoch stärker angepasst werden müsste.

K11 Motivation
In Hinblick auf das Kriterium *K11 Motivation* muss betrachtet werden, inwiefern der Ansatz eher dazu führen kann, dass eine korrekte Eingabevalidierung umgesetzt wird. Dies ist insbesondere relevant, da der Programmierer, der den fehlerhaften Quellcode verfasst hat, über die detailliertesten Kenntnisse zu diesem Code verfügt und somit am besten dazu in der Lage ist, den Fehler zu beheben. In der Praxis verlassen sich Entwickler jedoch wie zuvor beschrieben meist auf andere Personen oder Prozesse, um

die Softwaresicherheit durchzusetzen. Durch beide Ansätze wird dies erreicht, indem der Aufwand für die Umsetzung des korrekten Validierungsprozesses auch für schwerwiegende Verwundbarkeiten, die etwa SQL-Injektionen ermöglichen, stark reduziert wird, insbesondere weil der Entwickler sich nicht mit den Implementierungsdetails auseinandersetzen muss.

K12 Lerneffekt
Nach diesem Kriterium soll festgehalten werden, inwiefern durch die Verwendung des jeweiligen Ansatzes ein Lerneffekt entsteht, damit der Entwickler die Vorgehensweisen für eine korrekte Eingabevalidierung beispielsweise abstrahieren und auch in anderen Programmiersprachen anwenden kann. Dies ist zudem notwendig, da viele Entwickler nicht über Kenntnisse hinsichtlich spezifischer Verwundbarkeiten und Vorgehensweisen zu ihrer Vermeidung verfügen (Xie, 2012, Seite 78).

Bei beiden Ansätzen wird der Entwickler durch das verwendete SCAT darauf aufmerksam gemacht, Eingaben zu validieren und wird somit für die Bedeutsamkeit dieser Fehlerform sensibilisiert. Da die Details des korrekten Validierungsprozesses bei dem API-basierten Ansatz für den Entwickler nicht einsehbar sind kommt es hier zu einem geringen Lerneffekt hinsichtlich der konkreten Vorgehensweisen zur Fehlervermeidung. Der Ansatz für das Generieren des Quellcodes erzielt hingegen einen deutlich stärkeren Lerneffekt, da der Code für den Entwickler offengelegt wird. Dies gilt insbesondere für die Variante, in der Quellcodevorlagen für den korrekten Validierungsprozess bereitgestellt werden, die durch den Entwickler manuell an die eigene Software angepasst werden müssen, da er sich hierbei näher mit den Details des korrekten Validierungsprozesses auseinandersetzen muss.

K13 Flexibilität
Das Kriterium *K13 Flexibilität* befasst sich damit, inwiefern die durch den Ansatz bereitgestellte Lösung durch den Entwickler angepasst werden kann. Eine Herangehensweise unter der Verwendung einer API ist darauf angewiesen, dass diese API möglichst vollständig umgesetzt ist, um alle nötigen Validierungsprozesse für alle möglichen Verwendungszwecke einer Eingabe umsetzen zu können. Der Ansatz für das Generieren des Quellcodes ist im Vergleich dazu deutlich flexibler, da der offengelegte Quellcode durch den Entwickler auch im Nachhinein noch angepasst werden kann.

7.2.4 Einschätzung der Ansätze

Durch die Betrachtung der entwickelten Ansätze anhand der einzelnen Kriterien wird ersichtlich, dass sowohl eine API-basierte Variante als auch ein Ansatz für das Generieren des Quellcodes unter der Verwendung statischer Code-Analysen den Entwickler sinnvoll bei der Anwendung relevanter Secure Coding Richtlinien für die korrekte Validierung von Eingaben unterstützen können.

Im Detail wird jedoch erkenntlich, dass der Ansatz für das Generieren des nötigen Quellcodes insbesondere durch den offengelegten Code verschiedene Vorteile gegenüber einer API-basierten Herangehensweise hat. Der Ansatz zum Generieren des Quellcodes kann vor allem anhand eines interaktiven Assistenten eine bessere Hilfestellung leisten und liefert flexiblere Lösungen, die problemlos durch den Entwickler direkt angepasst werden können, falls der standardmäßig bereitgestellte Quellcode

nicht alle Anforderungen des Entwicklers erfüllt. Zudem kann bei einer Variante, in der Vorlagen für den korrekten Validierungsprozess in den Quellcode des Entwicklers eingefügt werden, ein deutlich besserer Lerneffekt erreicht werden.

Die wesentlichen Nachteile eines Ansatzes für das Generieren des nötigen Quellcodes sind hingegen, dass durch manuelle Anpassungen des eingefügten Codes möglicherweise neue Fehler auftreten können sowie dass dieser Ansatz maßgeblich von der verwendeten IDE abhängig ist. Diese Variante kann daher nur in IDEs verwendet werden, für die ein entsprechendes Plugin umgesetzt wurde. Eine Alternative wäre die Umsetzung dieses Ansatzes als eigenständiges Programm, für dessen Verwendung der typische Arbeitsablauf des Entwicklers jedoch stärker angepasst werden müsste.

Letztendlich wurde durch die Entwicklung der Ansätze insbesondere ersichtlich, dass beide Varianten sinnvoll verwendet werden können, um traditionelle Werkzeuge zur statischen Code-Analyse sinnvoll zu ergänzen, da sie Möglichkeiten bereitstellen, um erkannte Probleme im Bezug auf fehlerhafte Eingabevalidierungen zu beheben. Dies wird insbesondere erzielt, indem in beiden Fällen ein vergleichsweise geringer Aufwand für den Entwickler anfällt und Fehler bei der Implementierung des jeweiligen Validierungsprozesses effektiv vermieden werden können. Insofern kann als zentrales Ergebnis dieser Arbeit festgehalten werden, dass zwei Ansätze gefunden werden konnten, welche die typische Funktionalität von traditionellen SCATs für das Auffinden potenzieller Fehler um eine Möglichkeit für ihre Behebung ergänzen, die sinnvoll und realisierbar sind: Entweder indem eine API umgesetzt wird, welche die korrekten Validierungsprozesse bereitstellt, oder indem ein IDE-Plugin mit einem interaktiven Assistenten für das Generieren des nötigen Quellcodes angeboten wird. Schlussendlich sollte in der weiteren Forschung auf der Basis dieser Erkenntnisse vor allem der Ansatz zum Generieren des Quellcodes weiterverfolgt werden, konkret indem ein Plugin für eine IDE entwickelt wird, um den Quellcode für verschiedenste Validierungsfehler generieren zu können[1]. Die Entwicklung eines möglichst vollständigen API-basierten Ansatzes kann ergänzend dazu näher betrachtet werden, da dieser trotz seiner Nachteile im direkten Vergleich mit der Variante für das Generieren des Quellcodes auch den maßgeblichen Vorteil hat, dass er unabhängig von der verwendeten Entwicklungsumgebung genutzt werden kann.

Für die erste Forschungsfrage kann somit konkretisierend festgehalten werden, dass zur Unterstützung des Entwicklers bei der Anwendung von Secure Coding Richtlinien im Hinblick auf die Thematik der Eingabevalidierung die Verwendung von Annotationen zur Definition des Validierungsprozesses, die Verwendung einer API, welche den Validierungsprozess übernimmt und das Generieren des Quellcodes für die Validierung denkbar wären. Auf der Basis der näheren Betrachtung und Evaluierung dieser Ansätze wird im Hinblick auf die zweite Forschungsfrage erkenntlich, dass der annotationsbasierte Ansatz sich zu diesem Zweck nicht eignet, da bereits die Einhaltung grundlegender Richtlinien mit ihm nicht sinnvoll garantiert werden kann. Die Ansätze für die Verwendung einer API oder das Generieren des nötigen Quellcodes stellen hingegen sinnvolle Ergänzungen traditioneller SCATs dar, da der Entwickler so nicht nur bei dem Auffinden verschiedener Fehler, sondern auch bei ihrer korrekten Vermeidung effektiv und effizient unterstützt werden kann.

[1]Weitere Hinweise für die Verbesserungsmöglichkeiten beider Ansätze können in den Abschnitten 6.6.3 und 6.7.3 gefunden werden.

Kapitel 8

Fazit

In diesem Kapitel werden die zentralen Erkenntnisse dieser Arbeit zusammengefasst und es wird ein Ausblick darüber gegeben, welche dieser Erkenntnisse in der nachfolgenden Forschung aufgegriffen werden sollten, um die wissenschaftliche Situation auf dem Themengebiet der Softwaresicherheit weiter zu verbessern.

8.1 Zusammenfassung

Um individuelle Ansätze zur Anwendung von Secure Coding Richtlinien für Java basierend auf statischer Code-Analyse entwickeln zu können, wurden in dieser Arbeit zunächst die wesentlichen Grundlagen der einzelnen relevanten Themen aufgearbeitet. Im Hinblick auf die Entwicklung sicherer Software wurde hierbei erkenntlich, dass das Ziel des Secure Coding die Durchsetzung der Softwaresicherheit von Beginn des Entwicklungsprozesses an ist. Dabei befasst sich das Secure Coding nicht ausschließlich mit Aspekten der Softwareentwicklung, die direkt mit der Sicherheit des Programms verbunden sind, sondern auch mit Faktoren, die sich indirekt auf die Sicherheit auswirken können, wie beispielsweise der Wartbarkeit einer Software.

Mit Secure Coding Richtlinien existiert ein Hilfsmittel, durch dessen Anwendung die Softwaresicherheit in großen Teilen gewährleistet werden kann. Bei der näheren Betrachtung dieser Thematik wurde erkenntlich, dass der Begriff der Secure Coding Richtlinie nicht eindeutig definiert ist und dass verschiedene Quellen dementsprechend Richtlinien mit einer unterschiedlichen Zielsetzung, Qualität und Vollständigkeit beschreiben. Im Hinblick auf die sichere Programmierung von Software in der Programmiersprache Java bietet der Kodierungsstandard des CERT die extensivste und qualitativ hochwertigste Liste an implementierungsbezogenen Richtlinien. Obwohl die Anwendung von Secure Coding Richtlinien in einer deutlichen Verbesserung der Softwaresicherheit resultieren kann, werden sie oft nicht angewendet. Die zentralen Gründe dafür sind, dass Softwareentwickler die Durchsetzung der Sicherheit oft nicht als ihre Aufgabe ansehen, kognitiv bereits ausgelastet sind oder die Secure Coding Richtlinien nicht kennen.

Werkzeuge zur statischen Code-Analyse, sogenannte SCATs, können hierbei aushelfen, da sie verschiedenste sicherheitsbezogene Fehler in der Software des Entwicklers automatisiert erkennen und den Entwickler über sie informieren können. Es wurde festgestellt, dass SCATs zwar verschiedene, teils nicht vermeidbare Probleme aufweisen, aber dennoch eine sehr effektive und effiziente Hilfestellung bei der Vermeidung

schwerwiegender Fehler leisten können, insbesondere da durch die Analyse des Codes Fehlerquellen statt Symptomen aufgedeckt werden. An dieser Stelle wurde zudem festgehalten, dass Werkzeuge zur statischen Code-Analyse üblicherweise eine vollautomatische Analyse der Software durchführen und dem Entwickler anschließend eine Liste potenzieller Fehler präsentieren. Werkzeuge, die diese Vorgehensweise umsetzen, wurden im Rahmen der Arbeit als „traditionelle SCATs“ bezeichnet.

Im Hinblick auf das Verbesserungspotenzial dieser Werkzeuge für eine bessere Unterstützung des Entwicklers bei der Anwendung von Secure Coding Richtlinien wurden zunächst nicht-traditionelle Ansätze für die Verwendung statischer Code-Analyse betrachtet. Hierbei wurde festgestellt, dass diese Ansätze sich von den traditionellen SCATs unterscheiden, indem sie interaktivere Konzepte umsetzen und zudem in zwei Kategorien eingeteilt werden können: Sie helfen entweder bei dem Auffinden von Fehlern oder bei dem Beheben gefundener Fehler. Um die Entwicklung der neuen Ansätze auf eine bestimmte Problemstellung fokussiert durchführen zu können, wurde untersucht, welche Formen von Schwachstellen und Verwundbarkeiten besonders häufig auftreten und schwerwiegend sind. Anhand der hierbei gewonnenen Erkenntnisse wurde beschlossen, neue Ansätze zur Hilfe bei der Anwendung von Secure Coding Richtlinien für das Problem der Eingabevalidierung zu entwickeln, die sich konkret mit der Unterstützung bei der Behebung gefundener Fehler befassen, da so eine möglichst effektive Hilfestellung gewährleistet werden kann.

Bei einer ersten Betrachtung der Thematik der Eingabevalidierung wurde erkenntlich, dass Ansätze für die Hilfe bei der Anwendung isolierter Richtlinien nicht sinnvoll sind. Stattdessen sollten die einzelnen Richtlinien im größeren Kontext der Eingabevalidierung betrachtet werden. Anschließend wurden die zwei zentralen, bereits existierenden Konzepte für die Unterstützung des Entwicklers bei der Umsetzung korrekter Eingabevalidierungen vorgestellt. Diese basieren auf der Verwendung von Annotationen, um Einschränkungen an Eingaben zu definieren, sowie der Verwendung von APIs, um die Validierung durchzuführen, und verwenden keine statischen Code-Analysen. Auf der Grundlage dieser Konzepte und der bisherigen Beobachtungen wurden daraufhin drei neue Ansätze zur Unterstützung bei der Anwendung von Secure Coding Richtlinien für die korrekte Umsetzung von Eingabevalidierungen unter der Verwendung von statischer Code-Analyse näher betrachtet. Im Hinblick auf einen annotationsbasierten Ansatz wurde schnell erkenntlich, dass dieser die Einhaltung elementarer Richtlinien nicht sinnvoll garantieren kann und dementsprechend keine ausreichend vollständige Lösung für das Problem der Eingabevalidierung erzielen kann, weshalb dieser Ansatz als untauglich für diesen Zweck herausgestellt wurde. Mit den Ansätzen unter der Verwendung einer API und für das Generieren des nötigen Quellcodes wurden zwei Möglichkeiten an dem Beispiel praktisch umgesetzter Proof of Concepts beschrieben, die dem Entwickler bei der Anwendung relevanter Richtlinien helfen können. Durch die Evaluierung dieser Ansätze wurde ersichtlich, dass beide Varianten traditionelle SCATs sinnvoll ergänzen und den Entwickler effektiv und effizient unterstützen können. Insbesondere der Ansatz für das Generieren des Quellcodes eignet sich für diesen Zweck, da der offengelegte Code diverse Vorteile hat. Allerdings unterliegt dieser Ansatz einer festen Restriktion, da er entweder als IDE-abhängiges Plugin oder als zusätzliches Programm, dessen Verwendung sich auf den typischen Arbeitsablauf des Entwicklers auswirkt, umgesetzt werden muss.

Im Hinblick auf die erste Forschungsfrage dieser Arbeit konnte somit festgestellt werden, dass sich nicht-traditionelle Ansätze für die Verwendung statischer Code-

Analysen entweder dazu eignen, eine interaktivere und somit präzisere Fehlerfindung zu ermöglichen oder eine vollautomatische oder interaktive Hilfestellung bei der Fehlerkorrektur zu leisten. Bezogen auf das Problem der Eingabevalidierung wurde erkenntlich, dass vorhandene SCATs entsprechende Fehler bereits gut erkennen können. Um die gefundenen Fehler zu beheben konnten jedoch keine Ansätze gefunden werden und es wurde festgehalten, dass die Problematik der Eingabevalidierung so komplex ist, dass eine vollautomatische Fehlerkorrektur nicht sinnvoll ist. Aus diesem Grund wurden mit der Verwendung von Annotationen für die Umsetzung des Validierungsprozesses, der Nutzung einer API und dem Generieren des Quellcodes zur Eingabevalidierung drei interaktive Varianten für die Fehlerkorrektur näher betrachtet. Für die zweite Forschungsfrage wurde erkenntlich, dass sich ein annotationsbasierter Ansatz hier nicht eignet, während die anderen beiden Ansätze eine sinnvolle Ergänzung bestehender SCATs darstellen können, um den Entwickler nicht nur bei dem Finden, sondern auch bei dem Beheben verschiedener Fehler zu unterstützen. Der Ansatz für das Generieren des Quellcodes eignet sich für diesen Zweck zudem besser als der API-basierten Ansatz.

Abschließend kann festgehalten werden, dass die zu Beginn der Arbeit definierten Ziele erfüllt wurden, indem sie zunächst einen Überblick über die grundlegenden Thematiken der sicheren Programmierung, auch für Java und später mit dem Schwerpunkt der Validierung von Eingaben liefert. Weiterhin wurden verschiedene Möglichkeiten zur Umsetzung sicheren Programmcodes beschrieben, von denen die Secure Coding Richtlinien als Fokus gewählt wurden. Die verschiedenen Aspekte, Methoden und Vorgehensweisen von Werkzeugen zur statischen Code-Analyse wurden herausgestellt und es wurde ein Überblick über ihr Verbesserungspotenzial und alternative Herangehensweisen geliefert. Gemäß der aufgestellten Ziele wurden drei Ansätze für die Unterstützung des Entwicklers bei der Anwendung von Secure Coding Richtlinien im Hinblick auf die Eingabevalidierung näher betrachtet, von denen sich zwei als sinnvoll herausstellten. Diese beiden Ansätze wurden durch einen praktisch umgesetzten Proof of Concept veranschaulicht und evaluiert. Das zentrale Ergebnis der Arbeit ist dabei, dass die Ergänzung traditioneller SCATs durch Möglichkeiten für die Behebung der gefundenen Probleme insbesondere im Hinblick auf Fehler in der Eingabevalidierung sowohl sinnvoll als auch praktisch realisierbar ist und dass zwei konkrete Ansätze zu diesem Zweck gefunden werden konnten, die in der weiteren Forschung weiter ausgearbeitet werden sollten.

8.2 Ausblick

Da die Durchsetzung der Softwaresicherheit ein besonders wichtiger Faktor für die Qualität und den Erfolg eines Softwareprojekts ist und im Rahmen dieser Arbeit ein Überblick über Ansätze geschaffen werden konnte, die eine effiziente und effektive Hilfestellung hierbei liefern, sollten die hier erlangten Erkenntnisse in der weiteren Forschung aufgegriffen werden. Dabei sollte zunächst die für die Softwaresicherheit besonders wichtige Thematik der Eingabevalidierung näher betrachtet werden. Konkret müssen zusätzlich zu den existierenden Secure Coding Richtlinien auch weitere sinnvolle Vorgehensweisen für die syntaktische und semantische Validierung von Eingaben basierend auf ihrem Verwendungszweck definiert werden. Spezialfälle wie die Verwendung relevanter Frameworks sind ebenfalls zu betrachten, wie etwa Hi-

bernate und die Vermeidung von HQL-Injektionen, welche durch die Secure Coding Richtlinien des CERT nicht konkret beschrieben werden. Die so festgehaltenen Vorgehensweisen können anschließend durch die Ansätze umgesetzt werden, um neben der Einhaltung der konkret durch die Secure Coding Richtlinien angegebenen Beispiele auch eine möglichst vollständige Lösung für das übergeordnete Problem der Eingabevalidierung bieten zu können.

Da im Hinblick auf die drei näher untersuchten Ansätze festgestellt wurde, dass die Verwendung von Annotationen für die Definition des Validierungsprozesses sich nicht für die Durchsetzung elementarer Secure Coding Richtlinien eignet, ist von einer weiteren Betrachtung dieses Ansatzes abzuraten. Der API-basierte Ansatz sowie die Variante für das Generieren des Quellcodes zur Umsetzung des Validierungsprozesses stellten sich hingegen im Rahmen der Evaluierung als vielversprechend heraus und sollten dementsprechend weiterverfolgt werden. Aufgrund der Ergebnisse der Evaluierung ist der Ansatz für das Generieren des Quellcodes vorzuziehen. In beiden Fällen ist festzulegen, ob der jeweilige Ansatz als neues Werkzeug oder als Erweiterung eines bestehenden SCATs umgesetzt werden sollten. Das Checker Framework könnte jeweils wie in dieser Arbeit veranschaulicht als Basis für die Durchführung der statischen Code-Analysen verwendet werden, da es die grundlegend nötigen Funktionalitäten bereitstellt. Alternativ kann untersucht werden, inwiefern die beschriebenen Ansätze für andere SCATs wie beispielsweise SpotBugs umgesetzt werden könnten.

Für den Ansatz unter der Verwendung einer API sollten bei einer vollständigeren Umsetzung die Erkenntnisse aus der zuvor erwähnten näheren Untersuchung der Thematik der Eingabevalidierung verwendet werden, um Funktionen bereitstellen zu können, welche die Vermeidung von Validierungsfehlern unter der Anwendung relevanter Secure Coding Richtlinien bei möglichst vielen verschiedenen Verwendungszwecken der Eingabe erleichtern. Zusätzlich sollte das als Grundlage verwendete SCAT so erweitert werden, dass es konkrete Hinweise dazu liefert, welche Funktionen der API zur Behebung des jeweils berichteten Fehlers verwendet werden sollten[1].

Im Hinblick auf den Ansatz zum Generieren des Quellcodes sollte das verwendete SCAT zunächst idealerweise als Plugin für die ausgewählte IDE umgesetzt werden. So ist zu vermeiden, dass der Arbeitsablauf des Entwicklers zur Verwendung des Analysewerkzeugs unnötig abgeändert werden muss oder dass das Softwareprojekt das SCAT selbst einbinden muss, wie etwa im Falle des Checker Frameworks. Dies wirkt sich maßgeblich auf die Gebrauchstauglichkeit des Ansatzes aus, die auch im Hinblick auf die konkrete Umsetzung des für diesen Ansatz nötigen Assistenten näher betrachtet werden sollte. Bei der Umsetzung dieses Ansatzes sollten ebenfalls die zuvor definierten konkreten Vorgehensweisen für die Durchführung von Validierungsprozessen genutzt werden, um den Quellcode für Eingaben mit verschiedensten Verwendungszwecken und unter der Einhaltung der relevanten Secure Coding Richtlinien generieren zu können. Der Assistent sollte den Entwickler hierbei möglichst effektiv und effizient unterstützen, etwa indem der er bei dem Umsetzen von XML-Schemata für die Vermeidung von XML-Injektionen oder bei dem Verfassen regulärer Ausdrücke für die Validierung eines Strings hilft. Weiterhin kann auf der Grundlage des hier entwickelten Proof of Concepts und der durchgeführten Evaluierung empfohlen werden, dass im Rahmen einer praxistauglichen Implementierung zunächst die

[1] Weitere Informationen zu dem Verbesserungspotenzial des Ansatzes können in Abschnitt 6.6.3 gefunden werden.

Funktionalität für das Generieren von Quellcodevorlagen umgesetzt werden sollte. Anschließend kann eine Möglichkeit ergänzt werden, um den Code weitestgehend vollautomatisch zu generieren und an den Kontext der Eingabe anzupassen[2].

Im Rahmen der Implementierung beider Ansätze sollten diese zudem an existierenden Softwareprojekten mit relevanten Verwundbarkeiten getestet werden, um so konkretere Erkenntnisse im Hinblick auf eine möglichst praxistaugliche Umsetzung gewinnen zu können. Anschließend ist zu betrachten, inwiefern sich diese Ansätze auch für die Unterstützung bei der Anwendung von Richtlinien eignen, die sich nicht mit der Thematik der Eingabevalidierung befassen. Hierbei könnte sich insbesondere der Ansatz für das Generieren des Quellcodes als hilfreich erweisen, da dieser durch den interaktiven Assistenten sowie das dynamische Generieren des Codes deutlich besser an verschiedene Problemfälle angepasst werden kann als eine statische API. Zusätzlich kann über eine Adaptierung der Ansätze für andere Programmiersprachen nachgedacht werden.

Abseits der hier untersuchten nicht-traditionellen Ansätze könnte außerdem betrachtet werden, inwiefern traditionelle SCATs verbessert werden können. Beispielsweise indem nach den Erkenntnissen von Goseva-Popstojanova und Perhinschi (2015) neue Regeln entwickelt werden oder indem die Gebrauchstauglichkeit vorhandener SCATs näher betrachtet wird.

[2] Weitere Informationen für mögliche Verbesserungen dieses Ansatzes können in Abschnitt 6.7.3 gefunden werden.

Anhang A

Inhalt der Begleit-CD

```
.
+-- Expose/
|   +-- Expose.pdf
+-- Masterarbeit/
|   +-- Masterarbeit.pdf
+-- Implementierung/
    +-- CheckerFrameworkAPI/
    +-- CodeGenerationExample/
    +-- CodeGenerationPlugin/
    +-- Datenbank_Backup/
        +-- db_backup.sql
```

Listing A.1: Inhalt der Begleit-CD

Anhang B

Hinweise zur Ausführung der entwickelten Proof of Concepts

B.1 Ansatz 2: SCAT-basierte APIs

Der Quellcode für die Implementierung des Proof of Concepts zu diesem Ansatz ist auf der Begleit-CD unter `Implementierung/CheckerFrameworkAPI/` beigelegt. Die folgenden Hinweise, um den Quellcode auszuführen werden anhand der IDE IntelliJ IDEA[1] beschrieben. Für die Verwendung dieses Ansatzes muss zudem Java 8 installiert sein[2] und um die beigelegten Beispiele für die Vermeidung von SQL-Injektionen ausführen zu können, muss eine MySQL Datenbank wie in Anhang B.3 beschrieben angelegt werden. Anschließend kann das Projekt CheckerFrameworkAPI in IntelliJ IDEA geöffnet werden und die Dependencies müssen durch den Aufruf von `View` → `Tool Windows` → `Maven` → `Reimport all Maven Projects` importiert werden. Zusätzlich müssen die neuen Annotationen für das Checker Framework über `File` → `Project Structure` → `Modules` → `Add` → `JARs or directories...` und die Auswahl des Ordners `CheckerFrameworkAPI/_compiledAnnotations/` als Dependencies hinzugefügt werden, siehe Abbildung B.1. Falls andere Zugangsdaten oder ein anderer Datenbankname für MySQL verwendet werden sollen, müssen diese Daten zusätzlich in den Klassenvariablen der Klasse `SQLValidationExample` sowie in der Datei `src/main/resources/hibernate.cfg.xml` abgeändert werden.

Die standardmäßig bereitgestellten Argumente, welche als Eingaben für die beigelegten Beispiele verwendet werden, werden durch das Programm automatisch aus der Datei `src/main/resources/default_args_utf8.txt` ausgelesen. Je nachdem welches Beispiel ausgeführt werden soll, kann der entsprechende Quellcode in der Main-Methode in der Datei `Program.java` einkommentiert werden. Die Beispiele für die Funktionsweise des Ansatzes sind in dem Package `testclasses.validationExamples` zu finden, während die als Proof of Concept umgesetzte API sich in dem Package `validatorApi` befindet. Sobald das Programm kompiliert wird, beispielsweise per `Build` → `Recompile...`, werden vorhandene Fehler unter dem Reiter „Messages“ durch das Checker Framework angemerkt. Wenn das Programm gestartet wird, werden die jeweils einkommentierten Beispiele ausgeführt.

[1] https://www.jetbrains.com/idea/, aufgerufen am 15.03.2020
[2] https://www.java.com/de/download/, aufgerufen am 15.03.2020

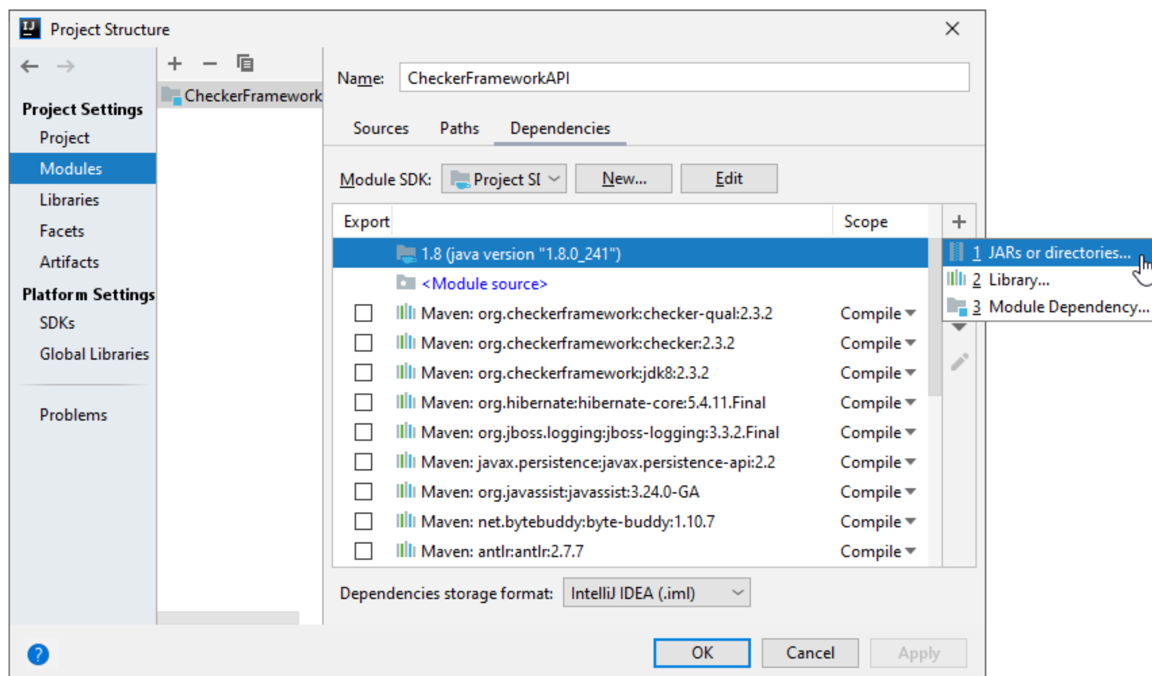

Abbildung B.1: Einbinden neuer Annotationen als Dependencies für das Checker Framework in IntelliJ

B.2 Ansatz 3: Generierung von Quellcode

Der Quellcode für das implementierte Plugin für die IDE IntelliJ IDEA[3] ist auf der Begleit-CD in dem Ordner `Implementierung/CodeGenerationPlugin/` zu finden. Um diesen Ansatz auszuführen muss Java 8 installiert sein[4]. Um das Beispiel zur Vermeidung von SQL-Injektionen ausführen zu können, muss zudem eine MySQL-Datenbank angelegt werden, siehe Anhang B.3. Anschließend kann das Projekt CodeGenerationPlugin in IntelliJ IDEA geöffnet werden und die Dependencies müssen per `View` → `Tool Windows` → `Gradle` → `Reimport All Gradle Projects` importiert werden. Danach kann das Projekt ausgeführt werden, um eine neue Instanz der IDE zu öffnen. Hier ist das Projekt `Implementierung/CodeGenerationExample/` zu öffnen und die Dependencies für das Checker Framework sind per `View` → `Tool Windows` → `Maven` → `Reimport All Maven Projects` zu importieren. Anschließend müssen die neuen Annotationen für das Checker Framework anhand von `File` → `Project Structure` → `Modules` → `Add` → `JARs or directories...` und der Auswahl des Ordners `CodeGenerationExample/_compiledAnnotations/` als Dependencies hinzugefügt werden, siehe Abbildung B.1. Falls andere Zugangsdaten oder ein anderer Datenbankname für MySQL verwendet werden, müssen diese Daten zusätzlich in den Klassenvariablen der Klasse `SQLExample` abgeändert werden.

Anschließend kann das Beispielprojekt kompiliert werden, woraufhin IntelliJ die durch das Checker Framework produzierten Warnungen anzeigt. Per Rechtsklick auf eine der Warnungen kann der Assistent zum Generieren des Quellcodes über den Eintrag „Generate Code for Input Validation“ aufgerufen werden. Die standardmäßig bereitgestellten Argumente, welche als Eingaben in den beigelegten Bei-

[3]https://www.jetbrains.com/idea/, aufgerufen am 15.03.2020
[4]https://www.java.com/de/download/, aufgerufen am 15.03.2020

spielen verwendet werden, liest das Programm hierbei automatisch aus der Datei `CodeGenerationExample/src/main/resources/default_args_utf8.txt` aus. Die zur Verwendung des Ansatzes im Assistenten anzugebende Datei für die Vermeidung der potenziellen XML-Injektion „xmlSchema.xsd" befindet sich in dem Ordner `CodeGenerationExample/_files/`.

B.3 Aufsetzen der MySQL-Datenbank

Zunächst muss die Community-Version von MySQL installiert werden[5]. Im Folgenden wird angenommen, dass der Benutzer der Datenbank während der Installation von MySQL das Passwort „root" für den root-Account gewählt hat und sich anhand dieses Accounts mit MySQL verbindet. Um die Datenbank zur Veranschaulichung der hier entwickelten Ansätze aufzusetzen, kann ein beigelegtes Backup geladen werden. Zu diesem Zweck muss sich der Anwender zunächst über die Kommandozeile mit seinem Namen und Passwort mit MySQL verbinden, eine neue Datenbank erstellen und anschließend die Verbindung trennen, siehe Listing B.1.

```
> mysql -u root -p
> CREATE DATABASE mydb;
> quit
```

Listing B.1: Erstellen einer neuen MySQL Datenbank

Danach kann das unter `Implementierung/Datenbank_Backup/db_backup.sql` auf der Begleit-CD hinterlegte Backup in die erstellte Datenbank geladen werden, siehe Listing B.2.

```
> mysql -u root -p mydb < db_backup.sql
```

Listing B.2: Importieren des beigelegten Datenbank-Backups

Das Backup erstellt die Tabelle `users` mit zwei Datensätzen, die über MySQL eingesehen werden können, siehe Listing B.3.

```
    > mysql -u root -p
    > use mydb
    > SELECT * FROM users;
+----+------------+-----------+
| id | username   | password  |
+----+------------+-----------+
|  1 | validuser1 | password1 |
|  2 | validuser2 | password2 |
+----+------------+-----------+
```

Listing B.3: Inhalt des beigelegten Datenbank-Backups

[5]https://dev.mysql.com/downloads/, aufgerufen am 02.04.2020

Glossar

Annotation Ein Begriff, um Metadaten im Quellcode eines Programms zu hinterlegen, *„Anmerkungen“*.

Best Practice Eine bewährte Methode, um ein konkretes Problem bestmöglich zu lösen.

Code-Smell Schlecht konstruierter Code, der selbst keinen Programmfehler darstellt, aber meist schwer verständlich ist.

Fuzzing Auch „Fuzzy Testing“, eine Technik für Softwaretests, bei der zufällige Eingaben an ein Programm übergeben werden, um mögliche Fehler festzustellen.

Internet of Things Sammelbegriff für Technologien zur Vernetzung physischer und virtueller Objekte.

Principle of Least Privilege Prinzip der geringsten Privilegien, ein Konzept das besagt, dass ein Benutzer immer nur über so viele Zugriffsrechte wie absolut nötig verfügen sollte, um unautorisierte Zugriffe auf Ressourcen auszuschließen.

Proof of Concept Konzeptioneller Beweis, eine Form einer Machbarkeitsstudie, oft auch im Zusammenhang mit der Umsetzung eines Prototypen.

Quick-Fix Funktion einer IDE, um kleinere Korrekturen des Quellcodes vollautomatisch durchzuführen.

Sandbox Das Sandbox-Prinzip bezeichnet einen isolierten Bereich, in dem jegliche Änderungen keine Auswirkungen auf das umliegende System haben.

Syntaktischer Zucker Alternative und vereinfachte, programmiersprachenspezifische Schreibweisen für bestimmte Grundkonstrukte in Quellcode.

Literatur

Alexander, R. T., Bieman, J. M. & Viega, J. (2000). Coping with Java Programming Stress. *Computer*, *33*. (Siehe S. 27).

Alkhalaf, M. A. (2014). *Automatic Detection and Repair of Input Validation and Sanitization Bugs* (Diss., University of California). (Siehe S. 77).

Andress, J. (2011). *The Basics Of Information Security: Understanding the Fundamentals of InfoSec in Theory and Practice* (R. Rogers, Hrsg.). Syngress. (Siehe S. 9).

Antunes, N. & Vieira, M. (2009). Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services. *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. (Siehe S. 33).

AUTOSAR. (2017). Guidelines for the use of the C++14 language in critical and safety-related systems. (Siehe S. 26).

Avizienis, A., Laprie, J.-C., Randell, B. & Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on dependable and secure computing*. (Siehe S. 12).

Ayewah, N. & Pugh, W. (2010). The Google FindBugs fixit. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis*. (Siehe S. 33).

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D. & Penix, J. (2008). Using Static Analysis to Find Bugs. *IEEE Software*. (Siehe S. 38).

Baca, D., Carlsson, B. & Lundberg, L. (2008). Evaluating the Cost Reduction of Static Code Analysis for Software Security. *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*. (Siehe S. 33, 38).

Baca, D., Petersen, K., Carlsson, B. & Lundberg, L. (2009). Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter? *2009 International Conference on Availability, Reliability and Security*. (Siehe S. 38, 39).

Bardas, A. G. (2010). Static Code Analysis. (Siehe S. 37).

Barik, T., Song, Y., Johnson, B. & Murphy-Hill, E. (2016). From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. (Siehe S. 60, 61, 95).

Bergeron, J., Debbabi, M., Erhioui, M. M., Lavoie, Y. & Tawbi, N. (2009). Static Detection of Malicious Code in Executable Programs. (Siehe S. 32).

Bishop, M. (2012). Some Thoughts on Teaching Secure Programming. (Siehe S. 11).

Bloch, J. (2008). *Effective Java* (G. Doench, Hrsg.). Addison-Wesley. (Siehe S. 27).

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M. & Klemmer, S. R. (2009). Two Studies of Opportunistic Programminc: Interleaving Web Foraging, Learning, and Writing Code. *CHI '09: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. (Siehe S. 23).

BSA. (2019). *The BSA Framework for Secure Software: A new approach to securing the software lifecycle*. Business Software Alliance. (Siehe S. 15).

BSI. (2019). *Cyber-Sicherheits-Umfrage - Cyber-Risiken & Schutzmaßnahmen in Unternehmen*. BSI. (Siehe S. 1).

Chelf, B. & Ebert, C. (2009). Ensuring the Integrity of Embedded Software with Static Code Analysis. *IEEE Software*. (Siehe S. 36, 46).

Chess, B. & West, J. (2007). *Secure Programming with Static Analysis* (J. Goldstein, Hrsg.). Addison-Wesley Professional. (Siehe S. 10, 11, 27, 29, 33–36, 38–47, 110, 111).

Cova, M., Kruegel, C. & Vigna, G. (2010). Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. *Proceedings of the 19th international conference on World wide web*. (Siehe S. 38).

de Bruhin, H. & Janssen, M. (2017). Building Cybersecurity Awareness: The need for evidence-based framing strategies. *Government Information Quarterly Volume 34, Issue 1, January 2017, Pages 1-7*. (Siehe S. 1).

DHS Cyber Security. (2013). Software Assurance. (Siehe S. 11).

Dijkstra, E. W. (1970). *Notes on structured programming*. Technische Hogeschool Eindhoven. (Siehe S. 32).

Dougherty, C., Sayre, K., Seacord, R. C., Svoboda, D. & Togashi, K. (2009). *Secure Design Patterns*. Software Engineering Institute. (Siehe S. 15, 16, 26).

Eckert, C. (2014). *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. De Gruyter Studium. (Siehe S. 7–10).

Emanuelsson, P. & Nilsson, U. (2008). A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science*. (Siehe S. 34).

Ersoy, E. & Sözer, H. (2016). Extending Static Code Analysis with Application-Specific Rules By Analyzing Runtime Execution Traces. *ISCIS 2016: Computer and Information Sciences*. (Siehe S. 46).

Fabian, B., Gürses, S., Heisel, M., Santen, T. & Schmidt, H. (2009). A comparison of security requirements engineering methods. (Siehe S. 15).

Fähndrich, M. & Leino, K. R. M. (2003). Declaring and Checking Non-null Types in an Object-Oriented Language. *ACM SIGPLAN Notices*. (Siehe S. 57).

Firesmith, D. (2004). A Taxonomy of Safety-Related Requirements. (Siehe S. 8).

Firesmith, D. G. (2005). A Taxonomy of Security-Related Requirements. (Siehe S. 12).

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B. & Stata, R. (2002). Extended Static Checking for Java. *ACM SIGPLAN Notices*. (Siehe S. 57).

Foster, J. S. (2007). *Cqual User's Guide*. University of Maryland. (Siehe S. 54).

Gleirscher, M., Golubitskiy, D., Irlbeck, M. & Wagner, S. (2016). On the Benefit of Automated Static Analysis for Small and Medium-Sized Software Enterprises. (Siehe S. 33).

Gong, L., Ellison, G. & Dageforde, M. (2003). *Inside Java 2 Platform Security: Architecture, API Design and Implementation* (M. Dageforde, Hrsg.). Addison-Wesley. (Siehe S. 27).

Goseva-Popstojanova, K. & Perhinschi, A. (2015). On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology Volume 68, December 2015, Pages 18-33.* (Siehe S. 51, 118).

Graff, M. G. & van Wyk, K. R. (2003). *Secure Coding: Principles & Practices: Principles and Practices* (D. Russell, Hrsg.). O'Reilly & Associates. (Siehe S. 31).

Greenfieldboyce, D. & Foster, J. S. (2004). Visualizing Type Qualifier Inference with Eclipse. *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange.* (Siehe S. 54, 55).

Greenfieldboyce, D. & Foster, J. S. (2007). Type Qualifier Inference for Java. *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications.* (Siehe S. 56).

Habib, A. & Pradel, M. (2018). How Many of All Bugs Do We Find? A Study of Static Bug Detectors. (Siehe S. 71).

Heffley, J. & Meunier, P. (2004). Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security? In *Proceedings of the 37th Hawaii International Conference on System Sciences.* (Siehe S. 11, 32, 36).

Hovemeyer, D., Spacco, J. & Pugh, W. (2005). Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. *ACM SIGSOFT Software Engineering Notes.* (Siehe S. 65).

Howard, M. & LeBlanc, D. (2003). *Writing Secure Code, Second Edition* (D. Bird, D. Musgrave & B. Johnson, Hrsg.). Microsoft Press Books. (Siehe S. 14, 22).

Howard, M. & Lipner, S. (2003). Inside the Windows security push. *IEEE Security & Privacy.* (Siehe S. 11, 36).

ISO/IEC 27000. (2018). *Information technology — Security techniques — Information security management systems — Overview and vocabulary.* ISO/IEC. International Organization for Standardization. (Siehe S. 7–9).

ISO/IEC TR 24772. (2013). *Information technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages through language selection and use.* International Organization for Standardization. (Siehe S. 16).

Jaspan, C. C., Chen, I.-C. & Sharma, A. (2007). Understanding the Value of Program Analysis Tools. (Siehe S. 33).

Johnson, B., Song, Y., Murphy-Hill, E. & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering.* (Siehe S. 72, 80, 95, 97).

Johnson, S. C. (1978). Lint, a C Program Checker. (Siehe S. 41).

Karim, M. (2015). *Source Code based Buffer Overflow Detection Technology* (Magisterarb., Northwestern Polytechnical University). (Siehe S. 45).

Knuth, D. E. (1989). The Errors of TeX. *Software—Practice & Experience.* (Siehe S. 31).

Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., … Kamath, R. (2017). *CogniCrypt: Supporting Developers in using Cryptography.* Technische Universität Darmstadt. (Siehe S. 80, 95, 104).

Lai, C. (2008). Java Insecurity: Accounting for Subtleties That Can Compromise Code. *IEEE Software.* (Siehe S. 28).

Leuer, S. (2019). *Sharper Crypto-API Analysis: Entwicklung eines integrierten Plugins zur statischen Code-Analyse der Nutzung von Krypto-APIs in C#* (Magisterarb., Hochschule Düsseldorf). (Siehe S. 33).

Li, P. & Cui, B. (2010). A Comparative Study on Software Vulnerability Static Analysis Techniques and Tools. *2010 IEEE International Conference on Information Theory and Information Security*. (Siehe S. 33).

Liggesmeyer, P. (2009). *Software-Qualität: Testen, Analysieren und Verifizieren von Software* (D. A. Rüdinger & D. M. Barth, Hrsg.). Spektrum Akademischer Verlag. (Siehe S. 32).

Lipford, H., Thomas, T., Chu, B. & Murphy-Hill, E. (2014). Interactive Code Annotation for Security Vulnerability Detection. In *Proceedings of the 2014 ACM Workshop on Security Information Workers* (17–22). SIW '14. New York, NY: USA: ACM. (Siehe S. 2, 52).

Long, F. (2005). *Software Vulnerabilities in Java*. Software Engineering Institute. (Siehe S. 23).

Long, F., Mohindra, D., Seacord, R. C., Sutherland, D. F. & Svoboda, D. (2011). *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional. (Siehe S. 12, 22, 38, 41).

Long, F., Mohindra, D., Seacord, R. C., Sutherland, D. F. & Svoboda, D. (2013). *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs*. Addison-Wesley. (Siehe S. 4, 18, 19, 23, 25, 26).

Marcilio, D., Bonifacio, R., Monteiro, E., Canedo, E., Luz, W. & Pinto, G. (2019). Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. (Siehe S. 71).

Marcilio, D., Furia, C. A., Bonifácio, R. & Pinto, G. (2019). Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings. *19th IEEE International Working Conference on Source Code Analysis and Manipulation*. (Siehe S. 59).

McGraw, G. (2006). *Software Security: Building Security In* (K. Gettman, Hrsg.). Addison-Wesley Software Security. (Siehe S. 10, 11, 15, 28, 35, 65, 73).

McGraw, G. & Chess, B. (2004). Static Analysis for Security. *IEEE Security & Privacy*. (Siehe S. 40, 42, 43).

Meng, N., Nagy, S., Yao, D., Zhuang, W. & Argoty, G. A. (2017). Secure Coding Practices in Java: Challenges and Vulnerabilities. (Siehe S. 23, 72, 81).

Mettler, A. & Wagner, D. (2009). The Joe-E Language Specification, Version 1.1. 18. (Siehe S. 15).

Michels, B. (2014). *Heartbleed - Wie sicher sind wir wirklich?* (E. Schindler, Hrsg.). CreateSpace Independent Publishing Platform. (Siehe S. 13).

MISRA. (2016). MISRA Compliance: 2016: Achieving compliance with MISRA Coding Guidelines. (Siehe S. 26).

Møller, A. & Schwartzbach, M. I. (2019). *Static Program Analysis*. Aarhus University, Denmark. (Siehe S. 32, 38, 43, 46).

Nadi, S., Krüger, S., Mezini, M. & Bodden, E. (2016). "Jumping Through Hoops": Why do Java Developers Struggle With Cryptographic APIs? *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. (Siehe S. 72, 80).

Nagappan, N. & Ball, T. (2005). Static Analysis Tools as Early Indicators of Pre-Release Defect Density. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* (Siehe S. 36).

Novak, J., Krajnc, A. & Žontar, R. (2000). Taxonomy of Static Code Analysis Tools. *The 33rd International Convention MIPRO.* (Siehe S. 41, 48).

Oaks, S. (2001). *Java Security: 2nd Edition* (D. Cameron, Hrsg.). O'Reilly Media. (Siehe S. 17, 19, 22).

Omar, C., Yoon, Y., LaToza, T. D. & Myers, B. A. (2012). Active Code Completion. *2012 34th International Conference on Software Engineering (ICSE).* (Siehe S. 61, 76, 95).

OWASP. (2010). OWASP Secure Coding Practices Quick Reference Guide. (Siehe S. 16, 26).

OWASP. (2017). *Software Assurance Maturity Model: A guide to building security into software development.* OWASP. (Siehe S. 15).

OWASP. (2019). *Application Security Verification Standard 4.0.* OWASP. (Siehe S. 15).

Peltier, T. R. (2013). *Information Security Fundamentals.* CRC Press. (Siehe S. 7, 9).

Pfleeger, C. P., Pfleeger, S. L. & Margulies, J. (2015). *Security in Computing.* Prentice Hall. (Siehe S. 9, 10, 13, 14).

Plakosh, D., Seacord, R., Stoddard, R., Svoboda, D. & Zubrow, D. (2014). *Improving the Automated Detection and Analysis of Secure Coding Violations.* Software Engineering Institute. (Siehe S. 33).

Ponemon Institute. (2018). *2018 Cost of a Data Breach Study: Global Overview.* IBM. (Siehe S. 1).

Reason, J. (1991). *Human Error.* Cambridge University Press. (Siehe S. 31).

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. (Siehe S. 38).

RogueWave. (2017). *Static Code Analysis: Myths, continuous integration, and success stories.* RogueWave. (Siehe S. 33, 36).

Saltzer, J. H. & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE, Vol. 63.* (Siehe S. 15, 20).

Scholte, T., Robertson, W., Balzarotti, D. & Kirda, E. (2012). Preventing Input Validation Vulnerabilities in Web Applications through Automatied Type Analysis. *2012 IEEE 36th Annual Computer Software and Applications Conference.* (Siehe S. 77).

Schönefeld, M. (2011). *Java Security: Sicherheitslücken identifizieren und vermeiden* (E. H. Pröfener, Hrsg.). mitp. (Siehe S. 9–11, 17, 19, 23, 41).

Seacord, R. C. (2006). Secure Coding Standards. *Proceedings of the Static Analysis Summit.* (Siehe S. 25, 28, 29, 33).

Secure Software, I. (2005). The CLASP Application Security Process. (Siehe S. 28).

Shen, H., Fang, J. & Zhao, J. (2011). EFindBugs: Effective Error Ranking for FindBugs. *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation.* (Siehe S. 38).

Sun, B., Shu, G., Podgurski, A. & Robinson, B. (2012). Extending static analysis by mining project-specific rules. *2012 34th International Conference on Software Engineering (ICSE).* (Siehe S. 37).

Sutherland, D. F. & Scherlis, W. L. (2010). Composable Thread Coloring. *ACM SIGPLAN Notices*. (Siehe S. 58).

Symantec. (2019). *Internet Security Threat Report Volume 24*. Symantec. (Siehe S. 1).

Tassey, G. (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology*. (Siehe S. 14).

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*. (Siehe S. 38).

Veracode. (2019). *State of Software Security Volume 10*. Veracode. (Siehe S. 1, 4, 65).

Viega, J., Bloch, J., Kohno, Y. & McGraw, G. (2000). ITS4: A Static Vulnerability Scanner for C and C++ Code. *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. (Siehe S. 40, 41, 45).

Wagner, D., Foster, J. S., Brewer, E. A. & Aiken, A. (2000). A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. *Proceedings of Network and Distributed Systems Security (NDSS 2000)*. (Siehe S. 33).

Ware, M. S. & Fox, C. J. (2008). Securing Java code: heuristics and an evaluation of static analysis tools. *SAW '08 Proceedings of the 2008 workshop on Static analysis*. (Siehe S. 28, 38).

Willis, C. & Britton, K. (2011). On Analyzing Static Analysis Tools. (Siehe S. 33).

Xie, J. (2012). *Interactive Programming Support For Secure Software Development* (Diss., University of North Carolina at Charlotte). (Siehe S. 112).

Xie, J., Lipford, H. R. & Chu, B. (2011). Why do programmers make security errors? *2011 IEEE Symposium on Visual Languages and Human-Centric Computing*. (Siehe S. 31, 39, 72, 76, 80).

Xie, J., Lipford, H. R. & Chu, B. (2012). Evaluating Interactive Support for Secure Programming. *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. (Siehe S. 80, 109).

Zave, P. & Jackson, M. (1997). Four Dark Corners Of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*. (Siehe S. 9).

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P. & Vouk, M. A. (2006). On the Value of Static Analysis for Fault Detection in Software. *IEEE Transactions on Software Engineering*. (Siehe S. 33).

Zhu, J., Chu, B., Lipford, H. & Thomas, T. (2015). Mitigating Access Control Vulnerabilities through Interactive Statice Analysis. *SACMAT '15: Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*. (Siehe S. 53, 54).

Zhu, J., Xie, J., Lipford, H. R. & Chu, B. (2013). Supporting secure programming in web applications through interactive static analysis. *Journal of Advanced Research 5(4)*. (Siehe S. 51, 82, 85, 111).

Zitser, M., Lippmann, R. & Leek, T. (2004). Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code. *ACM SIGSOFT Software Engineering Tools*. (Siehe S. 38).