

Masterarbeit

Erstellung eines Frameworks zur statischen Analyse von Quelltexten

vorgelegt von

Jonas A. Wendorf (Matrikelnummer: 51754)

Studiengang: Informatik

3. Mastersemester



Hochschule Aalen

Hochschule für Technik und Wirtschaft

Betreut durch Prof. Dipl.-Ing. Roland Hellmann

Zweitgutachter: Prof. Dr. Rainer Werthebach

08. April 2019

Erklärung

Ich versichere, dass ich die Ausarbeitung mit dem Thema »Erstellung eines Frameworks zur statischen Analyse von Quelltexten« selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung (Zitat) kenntlich gemacht. Das Gleiche gilt für beigelegte Skizzen und Darstellungen.

Aalen, den 08. April 2019

Ort, Datum

Jonas A. Wendorf

Autor

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Problemstellung und -abgrenzung	2
2	Theoretische Grundlagen	4
2.1	Übersicht über die Vorgehensweise	4
2.2	Anforderungen	6
2.3	Anwendungsgebiete	6
2.4	Marktanalyse	7
2.4.1	Vergleichbare Produkte	8
2.5	Auffinden verwundbarer Funktionen	9
2.5.1	Allgemeine Einschränkungen	10
2.5.2	Vergleich verschiedener Ansätze	10
2.6	Aufbau eines Parsers	14
2.6.1	Der Lexer	15
2.6.2	Der eigentliche Parser	17
2.6.3	Der AST	20
2.7	Beispiele verwundbarer Anwendungen	20
2.7.1	Beispiel eines verwundbaren PHP-Skripts	21
2.7.2	Beispiel eines verwundbaren C-Programms	23
2.8	Grundprinzipien der statischen Analyse	26
2.9	Satz von Rice	27
2.9.1	Beweis	28
2.9.2	Implikationen für die statische Analyse	28
2.10	Graphentheorie	29
2.10.1	Halbordnung	30
2.10.2	Verbandsordnung	30
2.10.3	Kontrollflussgraphen	31
2.10.4	Vereinfachungen und deren Auswirkungen auf die Qualität der Analyse	32
2.11	Erkennung des Dateiformats	34
2.11.1	Dateiendung	34
2.11.2	Magische Zahl	36
2.11.3	Vergleich der beiden Methoden	38
2.12	Erstellung von Regelsätzen	39
2.12.1	Inhalt eines Regelsatzes	40

2.12.2	Speicherung einzelner Regeln	46
2.12.3	Mögliche Formate für den Aufbau von Regeln	49
2.12.4	Erstellung universeller Regeln	60
2.13	Allgemeine Analyse der Codekomplexität	61
2.13.1	Berechnung der McCabe-Metrik	61
3	Praktische Umsetzung	68
3.1	UML-Klassendiagramm	68
3.2	Entwicklung in einer isolierten Umgebung	68
3.2.1	venv	70
3.2.2	Pipenv	70
3.2.3	Erstellung eines Installationsskripts	71
3.3	Codestil	71
3.4	Unittests	72
3.5	Einlesen der Kommandozeilenoptionen	72
3.5.1	Erstellung einer Hilfe	73
3.5.2	Übergabe von Kommandozeilenoptionen mittels docopt	73
3.6	Einlesen der zu untersuchenden Quelltexte	73
3.7	Auswahl der Module für verschiedene Programmiersprachen	74
3.7.1	Erkennung des Dateiformats anhand der Endung	74
3.7.2	Erkennung des Dateiformats mittels der magischen Zahl	75
3.7.3	Erkennung des Dateiformats anhand einer Heuristik	75
3.7.4	Erzwungene Erkennung unbekannter Dateiformate	76
3.7.5	Darstellung als Programmablaufplan	76
3.8	Parsen der Quelltexte	78
3.8.1	Auswahl einer Bibliothek zum Parsen	78
3.8.2	Erstellung der Grammatiken	81
3.8.3	Repräsentation im Programm	85
3.8.4	Analyse des AST	87
3.9	Parsen der Regelsätze	87
3.9.1	Definition von Schwachstellen	88
3.9.2	Definition von Sicherungen	88
3.9.3	Definition benutzerkontrollierter Eingaben	89
3.10	Suche nach Schwachstellen	89
3.10.1	Das Observer-Pattern	90
3.10.2	Dependency Injection	91
3.10.3	Ablauf eines Suchdurchlaufs	91
3.10.4	Speicherung der Suchergebnisse	94
3.11	Analyse der Codekomplexität	95
3.11.1	Einschränkungen der Komplexitätsprüfung	95
3.12	Erstellung eines Reports	95
3.12.1	Erstellung eines Reports als Textdatei	96
3.12.2	Erstellung eines Reports als Markdown-Datei	96

3.12.3	Erstellung eines Reports als HTML-Datei	97
4	Test des Frameworks	98
4.1	Test auf verschiedenen Systemen mittels Docker	98
4.2	Beispielreport und Auswertung	100
5	Evaluierung	104
5.1	Erreichte Ergebnisse	104
5.2	Empfehlungen	105
5.3	Ausblick	105

Abkürzungsverzeichnis

API	Application Programming Interface
AST	Abstract Syntax Tree
BASIC	Beginner's All-purpose Symbolic Instruction Code
CFG	Control Flow Graph
CGroups	Control Groups
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
DLL	Dynamic Link Library
DTD	Document Type Description
EBNF	Erweiterte Backus-Naur-Form
GPL	GNU General Public License
grep	global/regular expression/print
HTML	Hypertext Markup Language
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
JSONP	JavaScript Object Notation with Padding
LALR(1)	Look Ahead Left-to-Right
MIT	Massachusetts Institute of Technology
NP	nichtdeterministisch polynomielle Zeit, Komplexitätsklasse
OWASP	Open Web Application Security Project
PEP	Python Enhancement Proposal
PHP	PHP Hypertext Preprocessor
PIP	PIP Installs Packages
PLY	Python Lex-Yacc
PNG	Portable Network Graphics
POSIX	Portable Operating System Interface for Unix
SLY	Sly Lex-Yacc
SQL	Structured Query Language
SSRF	Server Side Request Forgery
TOCTOU	Time Of Check Time Of Use
UnionFS	Unification File System
XEE	XML External Entity
XML	Extensible Markup Language
XSS	Cross-Site Scripting

Yacc	Yet Another Compiler-Compiler
YAML	YAML Ain't Markup Language

1. Einleitung

1.1 Motivation

Das Aufspüren und Beheben von Sicherheitslücken ist eine wichtige Aufgabe bei der Entwicklung von Software.

Zwar existieren auf den meisten heutigen Systemen Schutzmechanismen, die ein Ausnutzen vorhandener Sicherheitslücken erschweren sollen, doch können diese natürlich nicht als Allheilmittel eingesetzt werden, da sie zum einen nicht auf allen Systemen vorhanden und aktiv sind, zum anderen aber auch, da sie nicht das darunter liegende Problem selbst korrigieren, sondern nur das Ausnutzen erschweren sollen und sich somit meistens umgehen lassen.

Dies kann dazu führen, dass häufig nicht mehr einzelne Lücken von Angreifern ausgenutzt werden, um Systeme zu übernehmen, sondern es werden mehrere Lücken miteinander kombiniert, um Stück für Stück die Abwehrmaßnahmen zu umgehen und Angriffsmöglichkeiten auszuweiten.

Eine manuelle Analyse von Quelltexten auf Sicherheitslücken dagegen ist sowohl mit hohen Kosten verbunden, da extra geschulte Mitarbeiter zu diesem Zweck in mühevoller Arbeit Quelltexte analysieren müssen, als auch selbst wiederum fehleranfällig, da gerade bei komplexen Programmabschnitten der Programmfluss unter Umständen sehr kompliziert wird.

Aus diesem Grund kann eine statische Analyse von Quelltexten im Hinblick auf Schwachstellen hilfreich sein, um verschiedene Fehler schnell und einfach zu finden.

Natürlich kann eine statische Analyse nicht sämtliche Fehler finden, allerdings kann die Zeit, die Mitarbeiter mit der Analyse von Quelltexten verbringen, sinnvoller dafür eingesetzt werden, komplexe und fehleranfällige Codestellen intensiver zu prüfen, als den kompletten Quelltext minutiös nach trivialen Schwachstellen abzusuchen.

Aus diesem Grund sollte eine manuelle Analyse gerade auch komplexer Codeabschnitte erfolgen, wobei auf diese besonders hingewiesen werden sollte.

Da immer wieder neue Muster entdeckt werden, welche einen Angriff ermöglichen, ist es dabei besonders wichtig, dass die Regeln für die statische Analyse möglichst einfach erweiterbar sind.

Viele existierende Schwachstellenscanner haben dieses Problem, da sie zum einen auf eine kleine Anzahl von Programmiersprachen beschränkt sind, zum anderen aber auch häufig entweder einen fest einprogrammierten Regelsatz haben oder keine einfache Anpassung und Erweiterung der Regelsätze durch Benutzer erlauben.

1.2 Problemstellung und -abgrenzung

Die Aufgabe besteht darin, ein Framework zu erstellen, welches der statischen Analyse von Quelltexten dient.

Das Framework soll dabei möglichst flexibel sein, um eine große Anzahl von Quelltexten untersuchen zu können und eine stetige Anpassung und Erweiterung im Unternehmensumfeld zu ermöglichen.

Zu diesem Zweck wird das Framework modular aufgebaut, wobei durch jedes Modul eine andere Programmiersprache analysiert werden kann. Somit soll es möglich sein, mit nur einem einheitlichen Framework sämtliche Quelltexte automatisch auf Schwachstellen untersuchen zu können.

Dabei könnte ein Modul für die Erkennung und Analyse von Quelltexten in C, ein anderes für die Erkennung und Analyse von Quelltexten in PHP¹ oder Python verantwortlich sein. Die Modularisierung hilft dabei, die Komplexität gering zu halten und gewährleistet eine einfachere Erweiterbarkeit, da Benutzer auch selbst neue Module schreiben und zum Scannen verwenden können.

Innerhalb der Module wiederum werden Regelsätze verwendet, anhand derer Schwachstellen beschrieben werden.

Hierbei werden in den Regelsätzen verwundbare Funktionen und auch Benutzereingaben aufgelistet, sodass Anwender später nachvollziehen können, an welcher Stelle ihr Quelltext verwundbar sein könnte und ob ein direkter Angriff durch eine Benutzereingabe erkannt wurde.

Anschließend wird das Framework versuchen automatisch zu erkennen, welches Modul es für welche Datei aufrufen soll, und anhand der Regelsätze die Quelltexte untersuchen.

Es wird dabei nicht darauf eingegangen, ob ein Ausnutzen der gefundenen Sicherheitslücken durch moderne Schutzmechanismen verhindert werden könnte, da diese Schutzmechanismen aus den oben beschriebenen Gründen nur als letzte Verteidigungslinie dienen sollten.

Auch eine Prüfung auf syntaktische Korrektheit der zu untersuchenden Quelltexte wird nicht vorgenommen.

Stattdessen wird davon ausgegangen, dass es sich bei den zu prüfenden Quelltexten um syntaktisch korrekten Programmcode handelt, um den Arbeitsaufwand zu reduzieren und das Programm auf das Wesentliche – die Erkennung von Sicherheitslücken – zu beschränken.

Weiterhin ist zu beachten, dass in einer statischen Analyse, welche beliebige Quelltexte untersuchen kann, eine Analyse der Programmlogik nicht möglich ist. Schwachstellen in

¹PHP Hypertext Preprocessor

der Programmlogik können aus diesem Grund nicht durch das Framework erkannt werden und es werden auch keine Versuche unternommen, solche zu erkennen. Eine derartige Analyse sollte stattdessen aus Sicht des Autors durch Unittests durchgeführt werden.

So wird zum Beispiel nicht untersucht, ob ein Benutzer mit der Rolle »Benutzer« in der Lage ist, sich selbst die Rolle »Administrator« zu geben, wodurch seine unprivilegierte Rolle umgangen werden kann.

Weiterhin ist es dem Autor nicht möglich, für sämtliche Programmiersprachen und Angriffsmuster Regeln zu schreiben.

Aus diesem Grund wird versucht, anhand von Modulen für einzelne Programmiersprachen und einfach zu konstruierenden Regelsätzen eine Erweiterbarkeit zu gewährleisten.

Eine Erstellung vollständiger Grammatiken und Regelsätze dagegen ist nicht Ziel der Ausarbeitung. Stattdessen werden vereinfachte Versionen erstellt, die zeigen sollen, wie eine Umsetzung im späteren Betrieb möglich sein wird.

Da das Ziel der Ausarbeitung nicht die Erstellung eines vollständigen Parsers welcher auch die syntaktische Korrektheit der Eingabedateien prüfen kann, sondern die statische Analyse von Schwachstellen ist, scheint dieser Ansatz sinnvoll.

Um die in der Motivation angesprochene manuelle Analyse zu vereinfachen, wird zusätzlich versucht, anhand verschiedener Metriken die Komplexität eines Codeabschnitts zu ermitteln.

Da komplexerer Code naturgemäß auch ein höheres Fehlerpotenzial birgt, können hiermit entsprechende Abschnitte hervorgehoben werden, sodass die manuelle Analyse zielgerichteter und schneller durchgeführt werden kann.

2. Theoretische Grundlagen

2.1 Übersicht über die Vorgehensweise

Der grobe Programmablauf kann dem Programmablaufplan aus Abbildung 2.1 auf der nächsten Seite entnommen werden.

Zuerst werden die Kommandozeilenoptionen des Programms eingelesen. Hier kann der Anwender auswählen, welche Dateien untersucht werden sollen, ob er die Auswahl des Moduls für die Untersuchung unerkannter Dateiformate erzwingen will (vergleiche Abschnitt 2.11.3 auf Seite 38) sowie welches Ausgabeformat genutzt werden soll.

Hiermit soll es einem Anwender ermöglicht werden, beliebige Dateien oder auch komplette Verzeichnisse mit Quelltexten auf Schwachstellen zu untersuchen.

Nach der Auswahl der zu untersuchenden Dateien muss anschließend das Framework herausfinden, welche Module für die Untersuchung infrage kommen. Zu diesem Zweck wird entweder die Dateiendung oder die Magische Zahl verwendet. Ein Vergleich dieser beiden Methoden findet sich in Abschnitt 2.11.3 auf Seite 38.

Da allerdings beide Erkennungsmethoden ihre Schwächen haben, kann die Erkennung auch durch eine Heuristik oder den Benutzer selber festgelegt werden.

Anschließend müssen die Quelltexte anhand von Grammatiken, welche in den Modulen beschrieben sind, geparkt werden, um eine für das Framework nutzbare Repräsentation des Quelltextes zu erhalten.

Bei der Erstellung dieser Repräsentation ist es wichtig, dass sämtliche Funktionsnamen erhalten bleiben, da im späteren Verlauf anhand derer erkannt wird, ob verwundbare Funktionen, Benutzereingaben und Absicherungen vorhanden sind und wie sie miteinander in Beziehung stehen.

Um diese Repräsentation auf Schwachstellen untersuchen zu können, müssen vordefinierte Regelsätze genutzt werden, wie sie in Abschnitt 2.12 auf Seite 39 beschrieben werden.

Anschließend kann die Suche nach Schwachstellen beginnen, wie in Abschnitt 2.5 auf Seite 9 beschrieben wird.

Hierbei ist es notwendig, die Quelltexte mindestens zweimal zu untersuchen, da eine Untersuchung zum Beispiel im ersten Durchlauf eine Schwachstelle finden kann, diese allerdings durch eine andere Funktion indirekt abgesichert wird, sodass diese zweite Funktion als zusätzliche absichernde Funktion hinzugefügt werden muss, bevor die Funktion neu untersucht und die Schwachstelle als abgesichert markiert werden kann.

Da diese indirekte Absicherung allerdings nicht den Best Practices der sicheren Programmierung entspricht, senkt sie das erkannte Risiko weniger als eine direkte Absicherung (vergleiche Abschnitt 2.12.1.3 auf Seite 44).

Da, wie in Abschnitt 2.9.2 auf Seite 28 bewiesen, keine 100%ige Erkennung aller Schwachstellen mittels statischer Analyse möglich ist, wird zusätzlich durch eine Analyse der

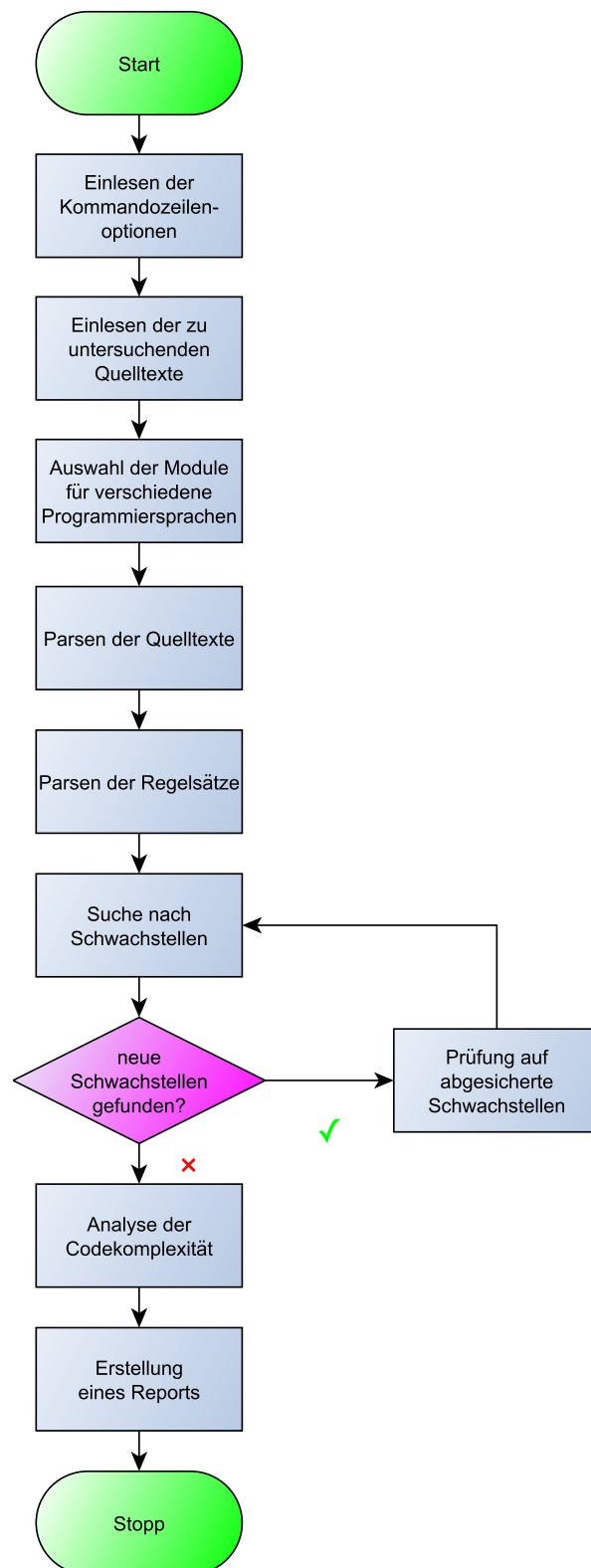


Abbildung 2.1: Programmablaufplan (grob)

Codekomplexität auf Funktionen hingewiesen, welche sehr komplex sind und daher mit höherer Wahrscheinlichkeit Fehler enthalten als andere Funktionen.

Zum Abschluss wird aus den gesammelten Daten ein Bericht in einem vom Benutzer gewählten Format erstellt und das Programm endet.

2.2 Anforderungen

Ziel des Frameworks ist es, bei der Suche nach Sicherheitslücken in Quelltexten zu helfen.

Hierzu müssen zum einen die Dateiformate korrekt erkannt werden, zum anderen müssen aber auch Module vorhanden sein, welche zur Analyse der jeweiligen Dateien dienen und anhand von Regelsätzen Sicherheitslücken erkennen.

Hierbei ist es wichtig, dass die Erstellung von Regelsätzen und – nach Möglichkeit – auch die Erstellung neuer Module für den Endanwender möglichst einfach ist, da im Rahmen dieser Masterarbeit aufgrund des zeitlichen Rahmens natürlich nur ein begrenzter Umfang an Modulen und Regelsätzen erstellt werden kann.

Da im Rahmen dieser Masterarbeit ein komplett neues Framework erstellt wird, wird die Optimierung der Geschwindigkeit oder das Abdecken sämtlicher Kategorien von Schwachstellen als nachrangig betrachtet.

Vorrangig wird versucht, den Programmiercode möglichst gut leserlich und einfach erweiterbar zu gestalten sowie die Module und Regelsätze in einer einfachen und gleichzeitig ausreichend mächtigen Form zu erstellen.

Hierdurch sollte es möglich sein, das Framework und die enthaltenen Regeln im eigenen Betrieb laufend zu erweitern und eigene Regelsätze anhand der individuellen Erfahrungen und Anforderungen anzulegen.

Hiermit sollte es möglich sein, ein Framework zu erstellen, welches als Universallösung etabliert und ständig weiterentwickelt werden kann, ohne dass eine aufwendige Einarbeitung in immer neue Produkte erforderlich wird, weil die bestehenden Produkte veraltet sind und keine Updates mehr angeboten werden.

2.3 Anwendungsgebiete

Mit einer statischen Codeanalyse wird der komplette Code eines Programms gescannt, ohne dass er dabei ausgeführt werden muss.

Somit ist es möglich, sogar Schwachstellen in kaum genutzten Codestellen zu finden, die aufgrund schwerer Erreichbarkeit in einem üblichen Penetration Test unter Umständen übersehen würden.

Hierdurch können Sicherheitslücken nicht nur schnell gefunden werden, sondern ein entsprechender Scan kann auch automatisiert eingebunden werden, wodurch Zeit und Entwicklungskosten gegenüber einer manuellen Prüfung gespart werden.

Auf der anderen Seite kann eine statische Codeanalyse nicht sämtliche Fehler in einem Quelltext finden (vergleiche Abschnitt 2.9.2 auf Seite 28), sodass die statische Analyse als Ergänzung zur manuellen Analyse von Quelltexten und einem Penetration Test eingesetzt werden sollte.

Weiterhin findet das entwickelte Framework Sicherheitslücken unabhängig von Schutzmechanismen des Betriebssystems, was bedeutet, dass nicht jede Lücke, die gefunden wird, automatisch auch auf allen betroffenen Systemen ausnutzbar ist, sofern das Betriebssystem einen entsprechenden Schutzmechanismus einsetzt.

Da allerdings zum einen diese Schutzmechanismen nicht auf allen Systemen installiert und aktiviert sind und zum anderen Sicherheitslücken teilweise auch auf ein tiefer liegendes Problem, wie ein mangelndes Verständnis des Entwicklers für sichere Programmierung oder Unachtsamkeit, hinweisen können, werden vom Framework lieber mehr nicht-ausnutzbare Lücken entdeckt, als ausnutzbare zu übersehen.

Durch die entstehende erhöhte Anzahl an False Positives bei gleichzeitig sinkender Anzahl False Negatives, wird ein Einsatz in Kombination mit dem OWASP DefectDojo[1] oder ähnlichen Projekten empfohlen.

2.4 Marktanalyse

Bei jeder Neuentwicklung ist es wichtig zu betrachten, welche Produkte derzeit erhältlich sind und wie sie sich von den eigenen Ideen unterscheiden, um festzustellen, ob sich die Neuentwicklung lohnt oder ob es sinnvoller ist, in eines der bestehenden Produkte zu investieren.

Aus diesem Grund wurde auch vor Beginn der Entwicklung dieses Frameworks eine Marktanalyse durchgeführt.

Das Ergebnis dieser Analyse brachte allerdings hervor, dass es nur wenige direkt vergleichbare Produkte gibt und diese teilweise entweder nicht mehr weiter entwickelt werden oder einen zu engen Fokus haben und somit eine Einbindung von mehreren Produkten gleichzeitig notwendig wäre, was aufgrund des Aufwands, allein einen statischen Code-scanner in der Entwicklungswerkzeugkette zu etablieren, schnell zu Ablehnung seitens der Anwender führen kann.

2.4.1 Vergleichbare Produkte

Zwei Produkte, die dem Ziel dieses Frameworks vergleichsweise nah kommen, werden im Folgenden näher betrachtet und dabei wird aufgezeigt, aus welchem Grund eine Neuentwicklung notwendig ist.

Hiermit soll keineswegs die Qualität dieser Produkte angezweifelt werden, welche mitunter weit längere Entwicklungszeiten hatten und sich in der Praxis gut bewährt haben.

Stattdessen soll gezeigt werden, welche Schwachstellen eine größere Verbreitung verhindern und wie diese behoben werden können.

2.4.1.1 PyT

Das PyT-Projekt (kurz für »Python Taint«) ist ein aus einer Masterarbeit[2] entstandener, statischer Codescanner für Webanwendungen, welcher Command Injections, SSRF², SQL³ Injections, XSS⁴, Directory Traversal und weitere Lücken erkennen kann.

Die Arbeit ist zwar eine große Inspiration für die Erstellung der vorliegenden Masterarbeit, da sie sich jedoch auf die Schwachstellenerkennung in Webframeworks beschränkt, unterliegt sie einigen dramatischen Einschränkungen, die einer hohen Verbreitung entgegenstehen.

Zum Ersten werden derzeit nur zwei Webframeworks, Flask[3] und Django[4], von dem Projekt untersucht.

Weiterhin implementiert das Projekt keinen eigenen Parser (vergleiche Abschnitt 2.6 auf Seite 14), sondern nutzt das Python-Modul `ast`[5], welches Teil der Python-Standardbibliothek ist und lediglich die Untersuchung von Python-Code ermöglicht.

Da hierdurch die Untersuchung anderer Programmiersprachen – wie PHP – unmöglich wird und auch keine Untersuchung von beliebigen weiteren Python-Projekten geplant ist, ist der Rahmen des Projekts zu klein, um eine weite Verbreitung zu ermöglichen.

Auf der anderen Seite erlaubt dieser enge Rahmen natürlich eine Spezialisierung und sehr gute Qualität der Analyse. Diese könnte allerdings auch mit entsprechend guten Regeln in einem allgemeiner gehaltenen Projekt erreicht werden.

Durch die gute Dokumentation des Projekts und die vielen allgemein verwendbaren guten Ansätze dient es allerdings im weiteren Verlauf dieser Ausarbeitung häufig als Inspiration und es wird auf den Ansätzen und Ergebnissen aufgebaut.

²Server Side Request Forgery

³Structured Query Language

⁴Cross-Site Scripting

2.4.1.2 RATS

RATS (kurz für »Rough Auditing Tool for Security«)[6] ist ein statischer Schwachstellen-scanner für C, C++, Perl, PHP, Python und Ruby.

Er unterstützt die Prüfung auf häufige Schwachstellen wie Buffer Overflows, TOCTOU⁵ und Race Conditions.

RATS benutzt hierfür einen eigenen Parser (vergleiche Abschnitt 2.6 auf Seite 14), um eine gute Erkennung zu gewährleisten.

Leider wird RATS bereits seit dem 31. Dezember 2013 nicht mehr weiterentwickelt und die Regelsätze zur Erkennung von Schwachstellen sind direkt in den C-Code integriert, sodass eine Erweiterung der Regeln für Anwender nur mit viel Einarbeitungszeit in den Quelltext von RATS verbunden ist.

RATS kommt daher den Zielen dieser Masterarbeit bereits sehr nahe, durch die Integration der Schwachstellendefinitionen in den Quelltext wird allerdings auch hier die weitere Verbreitung erschwert.

Da außerdem das Projekt nicht mehr aktiv weiterentwickelt wird, ist es höchstens eine Frage der Zeit, bis der Parser nicht mehr in der Lage ist, neue Versionen der oben aufgelisteten Programmiersprachen korrekt zu untersuchen.

2.5 Auffinden verwundbarer Funktionen

Um verwundbare Funktionen in Quelltexten zu finden, gibt es verschiedene Ansätze, die nachfolgend vorgestellt und miteinander verglichen werden.

Hierdurch ist es möglich, die Vor- und Nachteile der einzelnen Ansätze darzustellen und einen Ansatz zu wählen, der sowohl mit vertretbarem Implementierungsaufwand, als auch mit ausreichender Qualität Schwachstellen im Code aufdecken kann.

Im Folgenden werden daher drei Ansätze vorgestellt und miteinander verglichen.

Leider unterliegen alle vorgestellten Ansätze einigen Einschränkungen, die zu Anfang besprochen werden, um die Grenzen der statischen Analyse aufzuzeigen.

⁵Time Of Check Time Of Use

2.5.1 Allgemeine Einschränkungen

Bei der statischen Analyse von Code ist es niemals möglich, mit 100 %iger Sicherheit sämtliche Schwachstellen im Code zu finden.

Dieses Ergebnis liegt zum einen am Satz von Rice, wie in Abschnitt 2.9 auf Seite 27 nachvollzogen werden kann.

Zum anderen müssen allerdings auch die Regelsätze zur Erkennung von Schwachstellen von Menschen erstellt werden, sodass es immer möglich ist, dass Schwachstellen zwar theoretisch erkannt werden könnten, der Regelsatz zur Erkennung aber entweder nicht existiert oder fehlerhaft bzw. zu spezialisiert ist, sodass im konkreten Fall die Erkennung trotzdem fehlschlägt.

Weiterhin darf die Analyse nicht zu lange dauern, um die Entwicklung nicht zu sehr aufzuhalten, da ansonsten davon auszugehen ist, dass die Anwender den Scan nicht regelmäßig durchführen würden.

Die statische Analyse ist daher auch zeitlich eingeschränkt, wobei diese Grenze nicht fest definierbar ist, da es für das eine Unternehmen möglich ist, den Scan über Nacht unbeaufsichtigt laufen zu lassen, wohingegen ein anderes einen Scan nach jedem Commit auf den Server anstrebt.

Es ist allerdings möglich, nur einzelne Dateien zu scannen, welche durch einen Commit geändert wurden, weshalb, wie bereits angesprochen, das Framework für die erste Entwicklung nicht auf schnelle Ausführungszeit optimiert wird.

Da es bei der Suche nach Sicherheitslücken in der Regel wichtiger ist, dass mehr tatsächliche Sicherheitslücken gefunden werden, als dass weniger falsche Sicherheitslücken entdeckt werden, wird dabei versucht, eine möglichst gute regelbasierte Erkennung zu ermöglichen, die allerdings eine geringere Komplexität bei der Erstellung von Regelsätzen – und damit eine niedrigere Schwelle zur Erstellung und Anpassung eigener Regeln durch den Benutzer – auf Kosten der Vermeidung von False Positives vorzieht.

Diese Abwägung erscheint sinnvoll, da die Bewertung der Schwachstellen ohnehin später von Experten vorgenommen werden muss und diese, wie in Abschnitt 2.3 auf Seite 6 beschrieben, das Framework mit weiteren Projekten kombinieren können, um Reports zu speichern und damit False Positives in nachfolgenden Reports nicht mehr aufzuführen.

2.5.2 Vergleich verschiedener Ansätze

Im Folgenden werden verschiedene Ansätze zur statischen Analyse von Quelltexten verglichen.

Dabei wird aufgezeigt, welche Ansätze grundsätzlich existieren, wie effektiv sie sind, und wie aufwendig der spätere Einsatz wäre.

Im Anschluss werden die Ansätze miteinander verglichen und eine Entscheidung über den für diese Masterarbeit zu wählenden Ansatz begründet.

Es ist hierbei wichtig festzustellen, dass es keinen perfekten Ansatz für die Suche nach Schwachstellen in Quelltexten gibt, sodass immer eine Abwägung zwischen guter Erkennung von Schwachstellen, Geschwindigkeit und Komplexität des Ansatzes getroffen werden muss.

Auch kann die folgende Auflistung keine vollständige Liste darstellen, da einige Ansätze, wie zum Beispiel das Übersetzen der Quelltexte in ausführbaren Code und ein anschließendes Testen mit zufälligen Eingaben, sogenanntes »Fuzzing«, nicht als für diesen Zweck geeignete Methode angesehen wird, da hiermit die Erkennung zufallsbasiert ist und nicht nur Schwachstellen unerkannt bleiben können, sondern erkannte Fehler auch keine Rückschlüsse darüber zulassen, ob es sich um eine Schwachstelle handelt und wie diese behoben werden könnte.

Weitere Analysemethoden, die auf die Erkennung von Schwachstellen nicht mehr in den Quelltexten, sondern in den Binärdateien zielen, bleiben beim folgenden Vergleich aus diesem Grund unberücksichtigt.

2.5.2.1 Einfache Stringsuche

Eine einfache Stringsuche ist die wohl am leichtesten implementierbare Methode, um verwundbare Funktionen und Angriffsmuster in Quelltexten zu erkennen.

Es reicht hierfür ein regulärer Ausdruck, durch den zum Beispiel ungefilterte SQL-Abfragen wie in Abschnitt 2.7.1 auf Seite 21 erkannt werden können. Für diesen Fall könnte ein Ausdruck der folgenden Form zur Erkennung in PHP-Skripten dienen:

```
mysql_query\(("')SELECT .+ FROM .+ WHERE .+(?:\s+I?LIKE\s+|<>|>=?|<=?)\s+\.+\\1\\)
```

Mit diesem Muster wird der fixe String »mysql_query(« gefolgt von entweder einem doppelten oder einfachen Anführungszeichen gesucht, wiederum gefolgt von dem fixen String »SELECT«, einem oder mehreren beliebigen Zeichen, dem fixen String »FROM«, wieder einem oder mehreren beliebigen Zeichen, »WHERE« und anschließend beliebigen Zeichen gefolgt von einem Vergleichsoperator, einem oder mehreren weiteren beliebigen Zeichen und einem zweiten doppelten oder einfachen Anführungszeichen, je nachdem, welches Zeichen am Anfang gefunden wurde.

Wie man allerdings sehen kann, werden derartige reguläre Ausdrücke schnell unübersichtlich, sodass selbst in solchen vergleichsweise einfachen Fällen eine Visualisierung, wie in diesem Fall mittels Regexper⁶ erstellt und in Abbildung 2.2 auf der nächsten Seite zu sehen, notwendig ist, um den Ausdruck zu verstehen.

⁶<https://regexper.com>

Weiterhin können hiermit zusammenhängende Zeilen aufgelöst und als einzelne Anweisung erkannt und verarbeitet werden.

Auch bei der Erstellung eines Parsers ist es erforderlich, verwundbare Funktionen zu erkennen, sodass auch hier reguläre Ausdrücke notwendig sind.

Im Vergleich zur einfachen Stringsuche können die Ausdrücke allerdings weitaus einfacher sein, da durch den Parser bereits viele Ausdrücke normalisiert und in ihre Bestandteile aufgelöst wurden.

Leider können durch den Parser wieder nur Muster erkannt werden, es kann nicht geprüft werden, ob eine Ausführung mit verschiedenen Daten zu korrekten Ergebnissen führt oder ob es vielleicht sogar zu einem Überschreiben von Daten im Speicher kommt.

2.5.2.3 Analyse durch symbolische Ausführung

Bei der symbolischen Ausführung wird das Programm von einem Interpreter simuliert, wobei allerdings keine echten Werte als Eingaben verwendet werden, sondern lediglich Symbole.[7, S. 5]

Nachdem der Interpreter das Programm komplett durchlaufen hat, wobei für sämtliche möglichen Verzweigungen neue Symbole erzeugt wurden, wird ein Constraint Solver – also ein Programm, welches versucht, das Erfüllbarkeitsproblem (SAT) für sämtliche Verzweigungen zu lösen – eingesetzt, um sämtliche möglichen Pfade durch das Programm zu erkennen.

Eines der Hauptprobleme hierbei ist, dass es aufgrund der Vielzahl an möglichen Pfaden durch ein Programm zu einer sogenannten Pfadexplosion kommen kann.[7, S. 9]

Weiterhin wäre es notwendig, nicht nur einen Parser, sondern auch einen Interpreter für jede Programmiersprache zu erstellen, welcher die symbolische Ausführung ermöglicht.

Zwar ist die Rechengeschwindigkeit heute weit besser, als in den Anfangszeiten der symbolischen Ausführung, da es sich bei dem Erfüllbarkeitsproblem allerdings um ein NP^7 -vollständiges Problem handelt – was nach aktuellem Stand der Forschung bedeutet, dass kein effizienter Algorithmus zum Lösen des Problems existiert – kann sie auch heute noch nicht uneingeschränkt für die Suche nach Schwachstellen empfohlen werden.

2.5.2.4 Vergleich der Ansätze und Begründung der Entscheidung

Von den vorgestellten Ansätzen erscheint die Erstellung eines Parsers am vielversprechendsten.

⁷nichtdeterministisch polynomielle Zeit, Komplexitätsklasse

Hierbei kommt es zu weniger False Positives und – was noch viel wichtiger ist – zu weniger False Negatives durch nicht korrekt erkannte Funktionen als bei der Stringsuche, gleichzeitig kommt es aber nicht zu einer Pfadexplosion und der Implementierungsaufwand hält sich im Rahmen.

Durch die Erstellung eines Parsers ist es möglich, verwundbare Funktionen gezielt zu suchen und dabei gleichzeitig die Schwächen einer zu groben Suche zu vermeiden.

Da der Ansatz der symbolischen Ausführung allerdings auch seine Vorzüge hat, scheint es sinnvoll, diesen in modifizierter Form ebenfalls in das zu erstellende Framework aufzunehmen.

Konkret sollen bei Verzweigungen grundsätzlich sämtliche Möglichkeiten getestet werden, um zu erkennen, ob eine Schwachstelle im Quelltext existiert und woher die Eingaben für diese Schwachstelle kommen. Vergleiche hierzu auch Abschnitt 2.10.4 auf Seite 32.

2.6 Aufbau eines Parsers

Bei einem Parser handelt es sich um ein Programm, welches eine Eingabe in Form eines Quelltextes zerlegt und für die Weiterverarbeitung aufbereitet.

Mit anderen Worten ist der Parser dafür verantwortlich, aus dem für Menschen lesbaren Quelltext eine Repräsentation zu schaffen, welche vom Computer algorithmisch verarbeitet werden kann.

Somit kommt dem Parser gerade in Bezug auf die statische Analyse eine sehr wichtige Rolle zu, da die Qualität des Parsers maßgeblich bestimmt, welche Daten im Anschluss überhaupt überprüft werden können und wie diese miteinander in Zusammenhang stehen.

Wenn der Parser zum Beispiel eine Struktur im Code nicht richtig erkennt, so gibt es keine Möglichkeiten, diese Erkennung nachträglich noch hinzuzufügen.

Auf der anderen Seite muss der Parser allerdings eine Vereinfachung des Codes repräsentieren, da für den Programmablauf unnötige Elemente aus dem Quelltext entfernt und Konstrukte vereinheitlicht werden.

Da außerdem nicht in jedem Fall garantiert werden kann, dass die vom Framework ermittelten Repräsentationen dem entsprechen, was ein Compiler entwerfen würde und es auch schon zu Sicherheitslücken durch übereifrige Optimierungen von Compilern gekommen ist,^[8] kann nicht garantiert werden, dass durch diesen Ansatz sämtliche Sicherheitslücken gefunden werden, welche später im lauffähigen Programm auftreten könnten.

Eine Annäherung durch möglichst enge Orientierung des Parsers am Standard ist dabei die beste Voraussetzung, um eine möglichst gute Erkennung zu garantieren.

Ein Parser besteht dabei aus mehreren Teilen, die nachfolgend kurz erläutert werden.

Der Großteil der hier vorgestellten Definitionen basiert auf dem Vortrag »So you want to write an interpreter?« von Alex Gaynor.^[9]

2.6.1 Der Lexer

Ein Lexer ist ein Programm, welches eine Eingabe in eine Liste von Tokens übersetzt. Dieser Prozess wird als die »lexikalische Analyse« bezeichnet.

Jeder Token stellt dabei eine grundlegende syntaktische Komponente der Programmiersprache dar. So sind zum Beispiel ein einfaches + oder auch ein <= in vielen Programmiersprachen gültige Tokens.

Leerzeichen, Zeilenumbrüche und Tabs, aber auch Kommentare werden häufig von der lexikalischen Analyse verworfen, da sie keinen syntaktischen Unterschied ausmachen.^[10, S. 38]

Als Beispiel soll der folgende C-Code dienen:^[10, S. 38]

Listing 2.1: Beispiel C-Code

```
1 while (i <= 100) {  
2     tot += a[i]; /* form vector total */  
3     i++;  
4 }
```

Die lexikalische Analyse würde hieraus folgende Tokens generieren:

- while (RESERVEDWORD)
- ((LPAREN)
- i (IDENTIFIER)
- <= (LESS_EQUAL)
- 100 (INTEGERCONSTANT)
-) (RPAREN)
- { (LBRACE)
- tot (IDENTIFIER)
- += (PLUS_EQUALS)
- a (IDENTIFIER)
- [(LSQUARE)

- `i` (IDENTIFIER)
- `]` (RSQUARE)
- `;` (SEMICOLON)
- `i` (IDENTIFIER)
- `++` (PLUSPLUS)
- `;` (SEMICOLON)
- `}` (RBRACE)

In Klammern wurden hierbei Bezeichnungen für die Gruppe angegeben, zu denen das Token gehört.

So kann man zum Beispiel erkennen, dass es sich sowohl bei dem Token `i`, als auch bei dem Token `tot` und dem Token `a` um Variablen handelt.

Hierbei ist es wichtig zu beachten, dass der Lexer keinerlei Prüfung auf Gültigkeit der `while`-Schleife vornimmt. Diese Aufgabe würde später dem Interpreter oder Compiler zukommen.

Da ein solcher allerdings für die statische Analyse nicht erstellt wird, entfällt hier eine aufwendige Prüfung auf syntaktische Korrektheit des gesamten Programms und es findet lediglich eine Prüfung auf gültige Tokens statt.

Um den Lexer dabei möglichst robust zu gestalten, ist es im Framework möglich, ungültige Tokens zu ignorieren.

Damit sinkt zwar die Qualität der Eingabe und damit auch die Qualität der Schwachstellenerkennung insgesamt, allerdings bringen auch neue Sprachkonstrukte den Parser hierdurch nicht direkt zum Absturz, wenn kein entsprechendes Update der Module vorgenommen wurde.

Um neue Tokens hinzuzufügen, werden reguläre Ausdrücke verwendet, durch welche die allgemeine Form eines Tokens beschrieben wird.

Dadurch, dass die Tokens die kleinsten Bausteine des Quelltextes sind, lässt sich somit in einer Art Divide and Conquer eine Vereinfachung der benötigten regulären Ausdrücke ermöglichen.

2.6.2 Der eigentliche Parser

Der Parser bekommt als Eingabe die Ausgabe des Lexers und versucht, die Struktur des Codes zu ermitteln. Zu diesem Zweck baut der Parser einen sogenannten »Parse Tree« auf.

Hierfür wird typischerweise entweder ein Top-Down-Ansatz (vergleiche Abschnitt 2.6.2.1) oder ein Bottom-Up-Ansatz (vergleiche Abschnitt 2.6.2.2 auf Seite 19) genutzt.

Beide Ansätze sollen nachfolgend kurz erklärt werden, um ein Verständnis für die Funktionsweise des eingesetzten Parsers sowie mögliche Schwierigkeiten bei der Implementierung besser abschätzen zu können.

Da allerdings der Fokus dieser Ausarbeitung nicht Compilerbau ist, werden beide Verfahren nur kurz erklärt. Eine ausführlichere Erklärung kann zum Beispiel in [10, S. 77] gefunden werden.

2.6.2.1 Top-Down

Ein Top-Down-Parser startet bei einem einzigen Knoten, welcher mit dem Startsymbol gekennzeichnet ist. Anschließend baut er nacheinander von links nach rechts Unterbäume auf.

Zur Veranschaulichung soll folgendes Beispiel genutzt werden:

Der Ausdruck $1 * 0 + 1$ soll ausgewertet werden. Folgende Regeln existieren in der Grammatik:

Listing 2.2: Grammatik einer einfachen arithmetischen Sprache

```
1 <expr> ::= <term> | <expr> + <term>
2 <term> ::= <digit> | <term> * <digit>
3 <digit> ::= 0 | 1
```

Die Auswertung könnte nun wie in Abbildung 2.3 auf der nächsten Seite aussehen.

Es ist hierbei wichtig zu beachten, dass die Entscheidung, am Anfang nicht die Seite $\langle \text{expr} \rangle ::= \langle \text{term} \rangle$, sondern $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle$ auszuwerten, im Normalfall über Prioritätsdefinitionen geregelt werden muss, die allerdings für diese Veranschaulichung übersprungen werden.

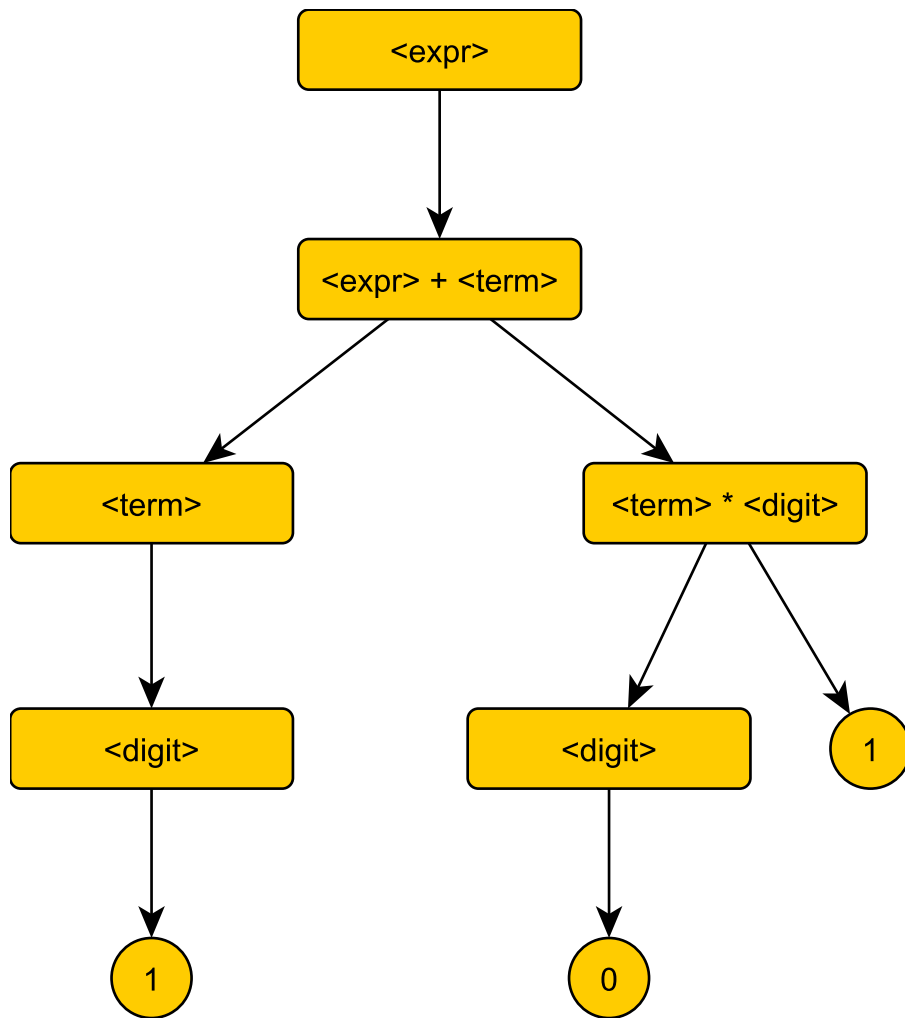


Abbildung 2.3: Aufbau eines Parse Trees mit Top-Down-Ansatz

2.6.2.2 Bottom-Up

Im Gegensatz zum Top-Down-Ansatz startet der Bottom-Up-Ansatz nicht mit dem Startsymbol, sondern mit dem ersten Token der Eingabe. Anschließend werden solange Tokens ausgewertet, bis das Startsymbol erreicht ist.

Zur Veranschaulichung soll auch hier wieder der Ausdruck aus Abschnitt 2.6.2.1 auf Seite 17 dienen. Dieses Mal könnte die Auswertung wie in Abbildung 2.4 aussehen.

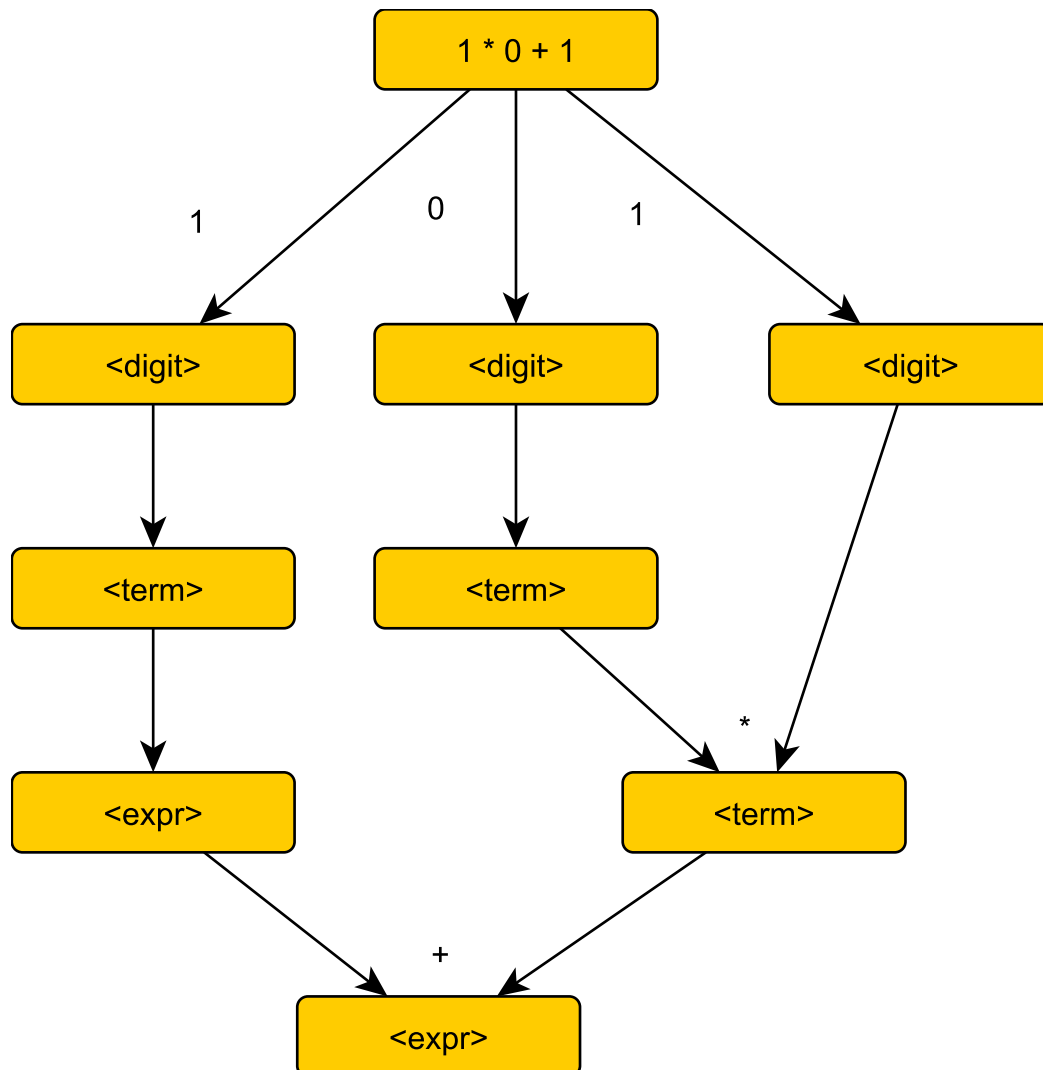


Abbildung 2.4: Aufbau eines Parse Trees mit Bottom-Up-Ansatz

2.6.3 Der AST

Bei einem AST⁸ handelt es sich um eine weitere Abstraktion über dem vom Parser generierten »Parse Tree«. Hierbei werden sämtliche Bestandteile entfernt, welche den Ablauf von Reduktionen im Baum oder sonstige Syntax-Eigenheiten widerspiegeln.[10, S. 26]

Im Vergleich zu Abbildung 2.3 auf Seite 18 könnte ein entsprechender AST zum Beispiel wie in Abbildung 2.5 aussehen.

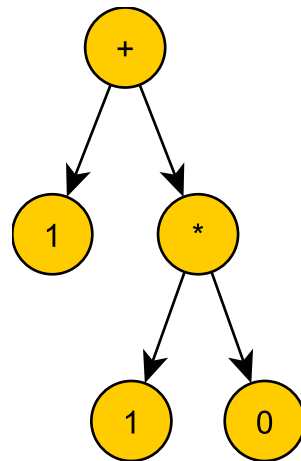


Abbildung 2.5: Beispiel eines AST

Wie man hieran sehen kann, wurden sämtliche Gruppenzugehörigkeiten entfernt und es bleibt nur noch die reine Programmlogik in einem Baum übrig, bei dem die Knoten die Verknüpfungen der Anweisungen miteinander und die Blätter deren Terminale, also die Operanden der Operatoren, darstellen.

2.7 Beispiele verwundbarer Anwendungen

Im Folgenden sollen einige Beispiele für verwundbare Anwendungen aufgezeigt werden.

Hierbei wird sowohl der Code selbst betrachtet, als auch untersucht, wie dieser als AST dargestellt werden kann. Anschließend werden die Schwachstellen anhand des ASTs nachvollzogen.

Hierdurch soll zum einen gezeigt werden, welche typischen Schwachstellen in einem Programm auftreten können, zum anderen soll allerdings auch schematisch dargestellt werden, wie diese Schwachstellen algorithmisch durch das Framework erkannt werden können, nachdem der Quelltext in geeigneter Form im Speicher vorliegt.

⁸Abstract Syntax Tree

Um die Beispiele möglichst einfach und verständlich zu halten, sind sie jeweils abstrakt gehalten und stark vereinfacht und auch die Auswertung wird nicht im Detail vorgenommen, da die resultierenden ASTs für menschliche Betrachter sehr unübersichtlich würden, ohne beim Verständnis des Ablaufs zu helfen.

2.7.1 Beispiel eines verwundbaren PHP-Skripts

Das folgende PHP-Skript ist anfällig für SQL-Injection-Angriffe.

Hierbei versucht ein Angreifer, eine SQL-Abfrage so zu erweitern, dass sein eigener Code ausgeführt wird.

Auch in der neuesten Ausgabe der OWASP⁹ Top 10 von 2017 stehen Injection-Angriffe wieder auf Platz 1.[11, S. 8]

Listing 2.3: Verwundbar für SQL Injections

```
1 $request = $_POST['request'];  
2 mysql_query("SELECT * FROM users WHERE id = $request");
```

In diesem Fall würde es reichen, im POST-Parameter `request` noch zusätzlich ein `OR 1=1` anzuhängen, um insgesamt eine immer wahre Abfrage zu erzeugen und somit sämtliche Benutzer zurückzugeben.

2.7.1.1 Auswertung als AST

Um die Schwachstelle im Skript zu erkennen, wurde der – stark vereinfachte – AST aus Abbildung 2.6 auf der nächsten Seite erstellt.

Hierbei wurden bereits einige Vereinfachungen und Gruppierungen vorgenommen.

So werden Benutzereingaben allgemein zusammengefasst als `user_input` und potenziell verwundbare Funktionen als `vuln_func`.

Dadurch lässt sich nachvollziehen, wie vom Benutzer kontrollierte Eingaben in verwundbare Funktionen eingeschleust werden können.

⁹Open Web Application Security Project

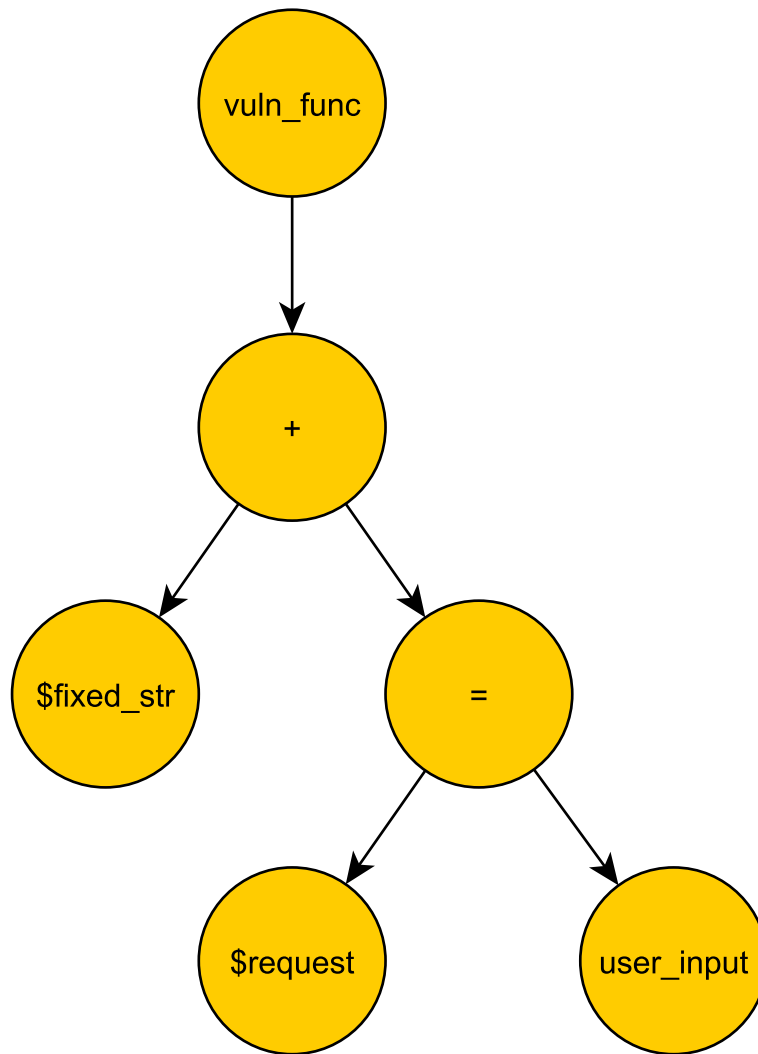


Abbildung 2.6: AST des verwundbaren PHP-Skripts

2.7.1.2 Erkennung der Schwachstelle

Die algorithmische Erkennung der Schwachstelle ist nach Erstellung des AST einfach möglich: Eine Benutzereingabe wird ohne Filterung an eine verwundbare Funktion weitergegeben.

Angelehnt an [12, S. 4] wird dabei die Benutzereingabe als Quelle (abgeleitet vom ursprünglichen »Source«) und die verwundbare Funktion als Senke (ursprünglich »Sink«) bezeichnet.

Um nun nachzuvollziehen, wie sich die Verunreinigung (ursprünglich: »Taint«) fortpflanzt, werden in Abbildung 2.7 auf der nächsten Seite die Knoten des AST markiert, welche durch die Quelle verunreinigt wurden.

Da in diesem Beispiel keine Filterfunktionen eingesetzt wurden, um die Verunreinigung zu beheben, kann sie sich bis zur Senke fortpflanzen.

2.7.2 Beispiel eines verwundbaren C-Programms

Das folgende C-Programm ist anfällig für Buffer Overflows.

Listing 2.4: Verwundbar für Buffer Overflows

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(int argc, char **argv)
7 {
8     char buffer[64];
9
10    gets(buffer);
11 }
```

Für dieses Programm genügt es, eine Eingabe mit mehr als 64 Zeichen zu übergeben, um den mittels `gets` befüllten Puffer auszufüllen und über dessen Grenzen hinaus zu schreiben.

Das Problem hierbei ist, dass `gets` keine Prüfung auf die Länge der Eingabe vornimmt, sodass es auch in der Manpage von `gets` heißt: »Never use `gets()` [...] it is extremely dangerous to use« (auf Deutsch: »Benutzen Sie niemals `gets()` [...] Es ist extrem gefährlich, es zu benutzen«).

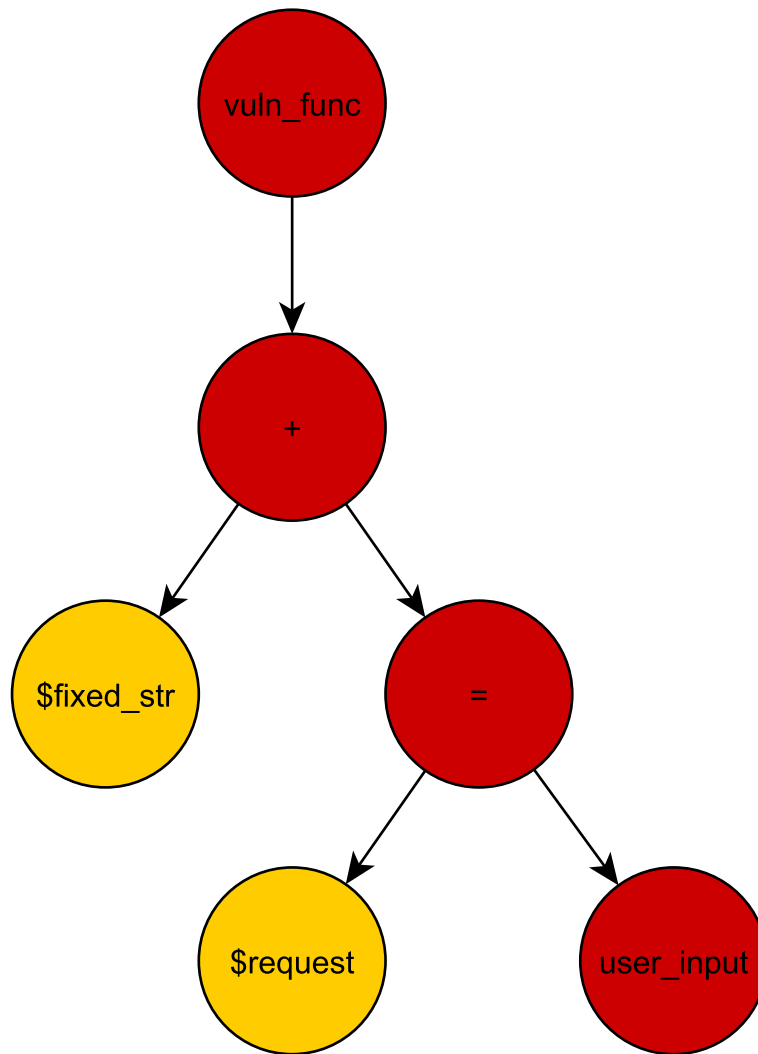


Abbildung 2.7: Die verunreinigten Knoten

Die Vorgehensweise zum Ausnutzen der Sicherheitslücke wird hier aus Platzgründen nicht erläutert, da es sich allerdings bei dem Beispielprogramm um ein Beispiel aus der Testumgebung »Protostar« von Exploit-Exercises handelt, kann eine Anleitung zum Ausnutzen zum Beispiel in [13] gefunden werden.

2.7.2.1 Auswertung als AST

In diesem Fall ist die Auswertung als AST noch kürzer als im Beispiel des verwundbaren PHP-Skripts, da die Verwendung der Funktion selbst die Schwachstelle darstellt und die Benutzereingabe bereits in ihr enthalten ist.

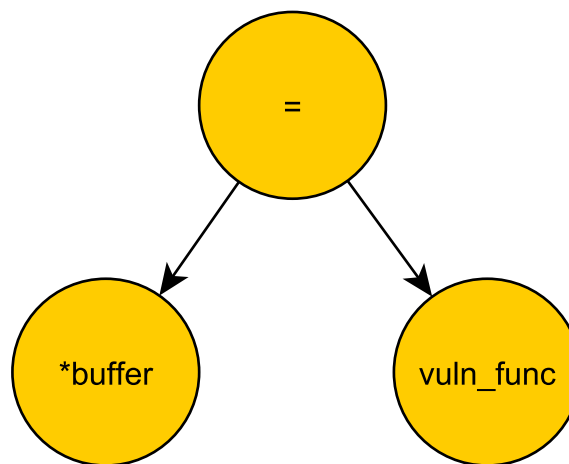


Abbildung 2.8: AST des verwundbaren C-Programms

Als eine Besonderheit fällt auf, dass der AST eine Zuweisung anzeigt, welche in dieser Form nicht explizit im Quelltext aufgeführt wird.

Dies liegt daran, dass es sich bei der Variable `buffer` um einen Zeiger handelt, woraufhin in der Funktion `gets` die vom Zeiger referenzierte Speicheradresse modifiziert wird. Es handelt sich daher hier um eine indirekte Zuweisung, weshalb auch der Variablenname mit dem Dereferenzierungsoperator `*` gekennzeichnet wurde.

Eine entsprechende Auflösung derartiger indirekter Zuweisungen muss daher unter Umständen auch vom Parser ermöglicht werden, wenn es wichtig ist zu wissen, ob eine Variable bei der Übergabe in eine andere Funktion kopiert wird (»Call by Value«) oder ob sie möglicherweise auch verändert wird (»Call by Reference«).

2.7.2.2 Erkennung der Schwachstelle

Auch in diesem Fall ist die algorithmische Erkennung der Schwachstelle einfach möglich: Die bloße Verwendung der `gets`-Funktion reicht aus, um die Schwachstelle zu ermitteln.

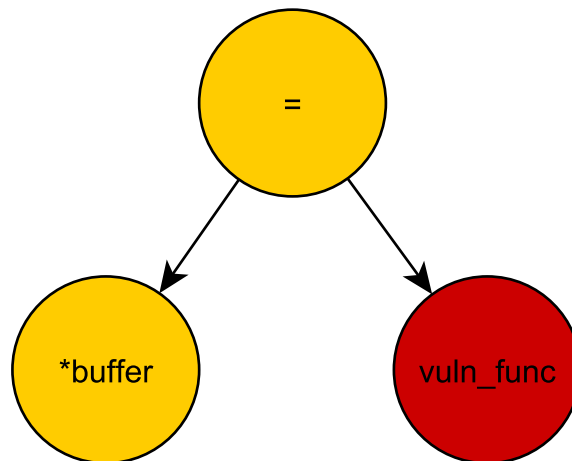


Abbildung 2.9: Der verunreinigte Knoten

Entsprechend wird in Abbildung 2.9 auch nur ein Knoten markiert, da keine Rückverfolgung der Quelle zur Senke notwendig ist, weil beides in einer Funktion liegt.

2.8 Grundprinzipien der statischen Analyse

Zur statischen Analyse ist, wie in Abschnitt 2.7.1.2 auf Seite 23 aufgeführt, eine Analyse des Kontrollflusses notwendig.

Um diese zu ermöglichen, sind allerdings einige Vorbereitungen notwendig.

1. Um die Unterscheidbarkeit zwischen Variablen und Funktionen zu gewährleisten, werden ihnen jeweils eindeutige Bezeichnungen zugeordnet.
2. Da zum Beispiel die Parameter bei einer Übergabe mittels »Call by Value«, also der Übergabe von Variableninhalten durch Kopie, statt durch eine Übergabe des Zeigers auf den Inhalt direkt, unabhängig voneinander sein sollten, müssen gegebenenfalls zusätzliche, versteckte Variablen eingeführt werden, um die Unabhängigkeit der Inhalte zu gewährleisten.
3. Mehrere Aufrufe einer Funktion können den Funktionsfluss ändern, zum Beispiel bei statischen Variablen in C oder Generatoren in Python.

Um dieses Problem zu lösen, können Funktionen entweder kontextabhängig analysiert werden oder es kann versucht werden, diese Einschränkungen durch symbolische Ausführung bei kontextunabhängiger Analyse zu lösen.

Wie aus den oben aufgeführten Problemen schnell ersichtlich wird, ist eine kontextunabhängige Analyse von Funktionen hinsichtlich des Implementierungsaufwands vorzuziehen

und bringt auch nur geringe Nachteile, die sich durch symbolische Ausführung beheben lassen.

Eines der Probleme bei symbolischer Ausführung besteht allerdings darin, dass es zu einer Explosion der möglichen Pfade durch eine Funktion kommen kann, wenn sämtliche Pfade probiert werden sollen. Eine Möglichkeit hiermit umzugehen, ist das Zusammenfassen ähnlicher Pfade.[\[14\]](#)

Eine weitere Möglichkeit, die in Abschnitt 2.10.4 auf Seite 32 näher beschrieben wird, ist es, Bedingungen und Schleifen jeweils als binäre Verzweigungen zu betrachten, bei denen jeder Zweig separat analysiert wird.

Dadurch wird das Problem der Pfadexplosion ebenfalls umgangen, gleichzeitig ist es außerdem nicht notwendig, eine Zusammenfassung von Pfaden zu versuchen, die ebenfalls aufwendig sein kann und nicht immer praktikabel ist.

Hierdurch werden zwar explizite Pfade zur Ausnutzung der Schwachstelle nicht offensichtlich und es kann zu mehr False Positives kommen, doch wird dieses Problem in Anbetracht der in Abschnitt 2.5.1 auf Seite 10 beschriebenen Problematik in Kauf genommen, da es für eine statische Analyse auf Sicherheitslücken besser ist, Sicherheitslücken zu erkennen, deren Ausnutzung derzeit noch nicht möglich ist, als Sicherheitslücken zu übersehen.

2.9 Satz von Rice

Der Satz von Rice ist ein Ergebnis aus der theoretischen Informatik und besagt, dass es keine Methode gibt, mit welcher man für alle Turingmaschinen verlässliche Aussagen über die von ihnen berechneten Funktionen machen kann.[\[15\]](#), S. 181]

Informell drückt es Michael I. Schwartzbach, Professor für Informatik an der Aarhus Universität in Dänemark, folgendermaßen aus: »all interesting questions about the behavior of programs are undecidable« ([\[16\]](#), S. 3]) (auf Deutsch: »alle interessanten Fragen über das Verhalten eines Programms sind unentscheidbar«).

Später sagt er dann über die Analyse von Datentypen: »Since this is an interesting question, we immediately know that it is undecidable.« ([\[16\]](#), S. 7]) (auf Deutsch: »Da es sich hierbei um eine interessante Frage handelt, wissen wir sofort, dass sie unentscheidbar ist.«).

Diese Aussagen können entmutigend erscheinen. Die Implikationen für die statische Analyse sollen daher in Abschnitt 2.9.2 auf der nächsten Seite näher betrachtet werden.

Um das Problem allerdings wirklich zu verstehen, muss es zuerst einmal formal betrachtet werden.

Theorem 1. [15, S. 185] Sei \mathcal{R} die Klasse aller Turing-berechenbaren Funktionen und sei $\mathcal{S} \subseteq \mathcal{R}$ mit $\mathcal{S} \neq \emptyset$ und $\mathcal{S} \neq \mathcal{R}$.

Dann ist die Sprache

$$C(\mathcal{S}) = \{w \mid \text{die von } M_w \text{ berechnete Funktion liegt in } \mathcal{S}\}$$

unentscheidbar.

2.9.1 Beweis

Für einen einfachen Teilbeweis des Satzes von Rice reicht es bereits aus, ihn auf das Halteproblem zu reduzieren, indem man ein Programm schreibt, welches in Abhängigkeit des Ergebnisses des Halteproblems eine Änderung in den Eigenschaften des Programms bewirkt.

So könnte man folgendes Programm schreiben:

Listing 2.5: Das Halteproblem entscheidet über den konstanten Wert einer Variable

```
1 x = 17;
2 if (TM(j)) {
3     x = 18;
4 }
```

Man sieht hieran, dass die Variable x nur dann konstant ist, wenn die Turingmaschine j anhält, was ein unentscheidbares Problem ist.[16, S. 3]

Natürlich ist mit diesem Teilbeweis noch nicht allgemein die Gültigkeit des Satzes von Rice bewiesen, auch wenn schnell ersichtlich ist, dass eine ähnliche Beweisführung für beliebige weitere Eigenschaften des Programms möglich wäre.

Auf den allgemeinen Beweis des Satzes von Rice wird allerdings an dieser Stelle aus Platzgründen verzichtet, er kann aber in [15, S. 181–185] nachvollzogen werden.

2.9.2 Implikationen für die statische Analyse

Der Satz von Rice scheint auf den ersten Blick zu bedeuten, dass eine statische Analyse von Quelltexten nicht möglich ist.

Bei näherer Betrachtung stellt sich allerdings heraus, dass lediglich eine eindeutige Entscheidung der Programmeigenschaften unmöglich ist.

Dagegen ist es für die Suche nach Sicherheitslücken vollkommen ausreichend, eine Annäherung zu finden, indem nach Mustern von Schwachstellen gesucht wird.

Mit diesen Mustern ist zum Beispiel gemeint, dass nach der Nutzung von bekanntermaßen verwundbaren Funktionen gesucht und geprüft wird, ob deren Eingaben vor der Nutzung korrekt abgesichert wurden.

Eine mangelnde Absicherung würde entsprechend auf eine Sicherheitslücke hindeuten.

Solch eine Mustersuche wird vom Satz von Rice nicht beeinträchtigt, allerdings wird durch ihn auf der anderen Seite klar, dass niemals eine in jedem Fall korrekte algorithmische statische Analyse von Quelltexten möglich sein wird.

Eine näherungsweise Analyse mit einer gewissen Irrtumswahrscheinlichkeit ist dagegen möglich.

Im Fall einer Schwachstellenanalyse sollte dabei versucht werden, den Fehler möglichst so einzugrenzen, dass eher False Positives, also Erkennung von Fehlern, die nicht existieren, als False Negatives, also Nichterkennung von Fehlern, die existieren, auftreten.

Der Grund hierfür ist, dass sich False Positives wie in Abschnitt 2.3 auf Seite 6 beschrieben behandeln lassen, wohingegen es fatal sein könnte, Fehler im Programmcode nicht zu erkennen, die später zu einer Übernahme des Systems durch einen Angreifer führen können.

2.10 Graphentheorie

Um Quelltexte algorithmisch prüfen zu können, ist es notwendig, sie in eine abstrakte, für den Computer verständliche Form zu überführen.

Hierzu wird die Graphentheorie genutzt, bei welcher der Zusammenhang von Objekten miteinander in einer abstrakten Struktur dargestellt wird.

Durch die Umwandlung in diese Repräsentation kann dabei eine Struktur für die Speicherung der Daten abgeleitet werden. Weiterhin können auch Erkenntnisse aus der Graphentheorie auf die Analyse der Quelltexte angewandt werden.

Für die Analyse von Quelltexten bietet es sich dabei an, die Quelltexte als eine Verbandsordnung (vergleiche Abschnitt 2.10.2 auf der nächsten Seite) zu betrachten, also eine spezielle Form der Halbordnung (vergleiche Abschnitt 2.10.1 auf der nächsten Seite).

Aus den resultierenden vereinfachten Graphen werden im Anschluss dann Kontrollflussgraphen (vergleiche Abschnitt 2.10.3 auf Seite 31) erstellt, anhand derer sich nicht nur die Relation der Knoten zueinander, sondern auch deren Richtung bestimmen lässt.

Hierdurch ist es schließlich möglich, den gesamten, für die Analyse notwendigen Programmablauf in einer wohldefinierten Struktur darzustellen.

Im weiteren Verlauf wird versucht, die hierbei entstehenden Graphen weiter zu vereinfachen, um die Komplexität der folgenden Analyse zu verringern, und dabei betrachtet, ob diese Vereinfachung negative Auswirkungen auf das Ergebnis der Analyse haben kann.

2.10.1 Halbordnung

Eine Relation \leq auf einer Menge H ist eine Teilmenge von $H \times H$. Wenn die folgenden Eigenschaften erfüllt sind:

- $x \leq x$ (Reflexivität)
- $x \leq y \wedge y \leq x \Rightarrow x = y$ (Antisymmetrie)
- $x \leq y \wedge y \leq z \Rightarrow x \leq z$ (Transitivität)

$\forall x, y, z \in H$, dann ist (H, \leq) eine Halbordnung.[17]

Wenn man noch einmal den AST aus Abbildung 2.6 auf Seite 22 betrachtet, fällt auf, dass hier sämtliche Eigenschaften einer Halbordnung erfüllt sind.

In der Tat wird durch die Vorgehensweise zur Erstellung eines solchen AST zwangsläufig eine Halbordnung entstehen.

Diese Erkenntnis erlaubt es, entsprechende bekannte Algorithmen aus der Graphentheorie auf das Problem der statischen Analyse anzuwenden.

Man kann die Struktur allerdings auch noch weiter spezifizieren, um somit weitere Spezialfälle ausnutzen zu können, wie im nachfolgenden Unterabschnitt gezeigt.

2.10.2 Verbandsordnung

Eine Verbandsordnung ist eine Halbordnung (H, \leq) , in der zu je zwei Elementen x, y sowohl $x \vee y$, das Supremum, also die kleinste obere Schranke der Menge, als auch $x \wedge y$, das Infimum, also die größte untere Schranke der Menge, existieren, wobei beide selbst nicht unbedingt Teil der Menge sein müssen.[18]

Betrachtet man nun einen Funktionsablauf vom Start bis zum Rücksprung, so ergibt jeder mögliche Pfad durch diese Funktion eine Verbandsordnung, bei der das Supremum den Funktionsstart und das Infimum einen zusätzlichen – nach dem ursprünglichen eingefügten – Rücksprung darstellen.

Hierdurch gelten folgende weitere Eigenschaften $\forall x, y, z \in H$: [17]

- $x \wedge x = x, x \vee x = x$ (Idempotenz)
- $x \wedge y = y \wedge x, x \vee y = y \vee x$ (Kommutativität)
- $(x \wedge y) \wedge z = x \wedge (y \wedge z), (x \vee y) \vee z = x \vee (y \vee z)$ (Assoziativität)

- $x \wedge (x \vee y) = x, x \vee (x \wedge y) = x$ (Absorptionsgesetz)

2.10.3 Kontrollflussgraphen

Ein CFG¹⁰ ist eine Graphdarstellung sämtlicher Programmpfade während der Ausführung. Hierbei wird der Graph wie in Abschnitt 2.10.2 auf der vorherigen Seite beschrieben um einen Start- und einen Rücksprungknoten erweitert.

Die Aufgabe des Kontrollflussgraphen ist es nun, den Programmablauf in seine Grundblöcke zu zerlegen.

Damit sind Blöcke gemeint, die nur Anweisungen enthalten, welche direkt aufeinander folgen und deren Programmcode direkt hintereinandersteht.[\[19, S. 5\]](#)

Als Beispiel soll der folgende Programmcode dienen:

Listing 2.6: Beispiel C-Programm

```
1 int a = 4,
2   b = a * 30,
3   c,
4   d = 5;
5 do {
6   c = b / a;
7   if (c >= 20) {
8     d = a * c;
9   }
10 } while (d < 100);
```

Um die Grundblöcke zu erhalten, müssen zuerst die Vorreiterzeilen identifiziert werden.[\[19, S. 9\]](#) Diese sind

- die erste Programmanweisung,
- explizite Ziele von Verzweigungen und
- implizite Ziele von Verzweigungen

Wenn alle Vorreiter identifiziert wurden, können die Basisblöcke als die Blöcke zwischen jeweils zwei Vorreitern identifiziert werden.

Hierzu werden zuerst die Anweisungen aus Listing 2.6 in ihre Komponenten aufgelöst:

¹⁰Control Flow Graph

Listing 2.7: Auflösung in die Komponenten

```
1      int a = 4;
2      int b = a * 30;
3      int c;
4      int d = 5;
5 LABEL1: c = b / a;
6      if c < 20 goto LABEL2
7      d = a * c;
8 LABEL2: if d >= 100 goto LABEL3
9      goto LABEL1
10 LABEL3: stop
```

Anschließend können gerichtete Kanten im CFG erkannt werden, indem geprüft wird, ob entweder

- eine Verzweigung von der letzten Anweisung eines Basisblocks zum Vorreiter des anderen Blocks führt oder
- der andere Block direkt auf den ersten folgt und der erste nicht in einer bedingungslosen Verzweigung endet

Als Resultat ergibt sich für das Programm aus Listing 2.6 auf der vorherigen Seite damit insgesamt der CFG aus Abbildung 2.10 auf der nächsten Seite.

Zwar ist auch eine Analyse des Kontrollflusses auf Anweisungsebene möglich, für die statische Suche nach Schwachstellen kann die blockbasierte Aufteilung jedoch Vorteile bringen, da somit der Programmablauf sehr einfach dargestellt und Blöcke bei Bedarf komplett ignoriert werden können, wenn sie keine anfälligen Anweisungen enthalten.

2.10.4 Vereinfachungen und deren Auswirkungen auf die Qualität der Analyse

Einige Vereinfachungen sind für die statische Analyse denkbar.

Als Erstes ist es möglich, bei der Analyse Bedingungen und Schleifen jeweils als binäre Verzweigungen zu betrachten, bei denen jeder Zweig separat analysiert wird.

Hierbei wird jeweils untersucht, ob sich die Möglichkeit einer Schwachstelle durch den betreffenden Zweig, unabhängig von allen anderen Verzweigungen innerhalb des Codeblocks, erhöht.

Eine solche Erhöhung einer Schwachstellenwahrscheinlichkeit ergibt sich zum Beispiel, wenn innerhalb eines Zweiges eine Funktion aufgerufen wird, Während es zu einer Senkung der Schwachstellenwahrscheinlichkeit führt, wenn Eingaben mit festen Werten überschrieben werden, da hiermit ausgeschlossen wird, dass diese Werte benutzerkontrolliert sind oder bleiben.

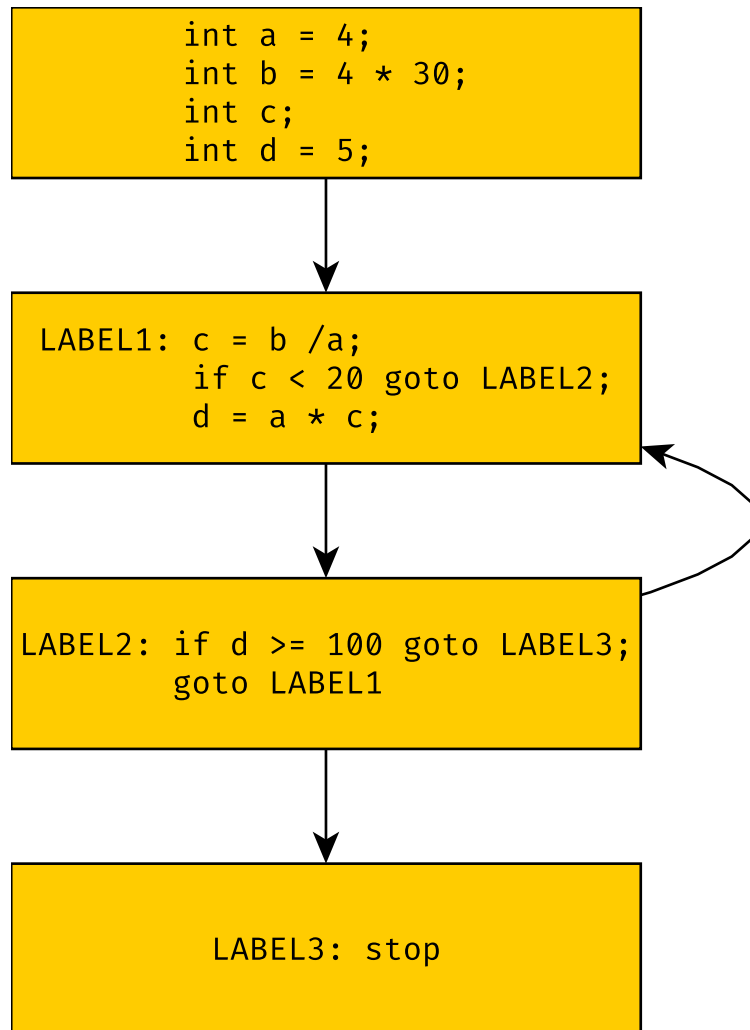


Abbildung 2.10: CFG des C-Programms

Hierdurch lässt sich das in Abschnitt 2.5.2.3 auf Seite 13 beschriebene Problem der Pfadexplosion umgehen. Auf der anderen Seite ist es hierdurch aber auch denkbar, dass toter Code analysiert wird, da selbst die beiden Zweige eines `if (false)` behandelt würden. Hierdurch könnte es zu False Positives kommen. Dies ist allerdings kein großes Problem, da – wie in Abschnitt 2.3 auf Seite 6 beschrieben – Programme existieren, mit denen derartige Fehler nach einmaliger manueller Analyse von weiteren Reports ausgenommen werden können.

Eine weitere Möglichkeit, die Analyse zu vereinfachen, ist es, irrelevante Codeblöcke zu ignorieren. Hierbei wird zuerst jeder Codeblock auf Senken untersucht und wenn keine gefunden wurden, wird der Block vorläufig zurückgestellt. Er wird nur dann erneut betrachtet, wenn in einem der darauffolgenden Blöcke eine Senke gefunden wurde. Sobald eine Senke gefunden wurde, werden die zurückgestellten Blöcke noch einmal in umgekehrter Reihenfolge untersucht, um die Verunreinigung zurückzuverfolgen. Diese Art der Vereinfachung hat keinerlei negative Auswirkungen auf die Qualität der Analyse, kann sie aber beschleunigen, da nicht mehr jeweils die gesamte Funktion, sondern nur noch relevante Codeblöcke mehrfach untersucht werden müssen.

2.11 Erkennung des Dateiformats

Bei der Suche nach Sicherheitslücken ist es wichtig zu erkennen, um welches Dateiformat es sich handelt. Ohne diese Erkennung kann nicht bestimmt werden, um was für eine Art von Quelltext es sich handelt – oder ob es sich überhaupt um einen Quelltext handelt – und es ist keine Erkennung von spezifischen Schwachstellen für eine Programmiersprache möglich.

Leider ist die Erkennung des Dateiformats kein einfaches Problem, weshalb im Folgenden zwei gängige Methoden vorgestellt und miteinander verglichen werden, um aufzuzeigen, wie eine Erkennung funktionieren kann und welche Einschränkungen möglich sind.

Nur wenn das Dateiformat schließlich richtig erkannt wurde, kann ein passendes Modul ausgewählt werden, durch welches das Framework eine Erkennung von Schwachstellen ermöglicht, indem die zu dem Modul gehörenden Regelsätze ausgewertet und somit sowohl Benutzereingaben, als auch verwundbare Funktionen erkannt werden.

Die Erkennung des Dateiformats ist allerdings auch wichtig, um den richtigen Parser auszuwählen und somit den Inhalt der Quelltexte korrekt in einer internen Datenstruktur zu repräsentieren, auf der die späteren Analysen aufbauen.

2.11.1 Dateiendung

Eine auf den ersten Blick einfache Möglichkeit, Dateiformate korrekt zu erkennen ist es, die Dateiendung zu betrachten. Durch die Dateiendung kann ein Anwender einen Hinweis

auf den Inhalt der Datei geben, allerdings ist die Dateiendung nicht ausschlaggebend für den Inhalt, da es sich hierbei lediglich um Metadaten handelt, die korrekt sein können oder nicht.

So können zum Beispiel PNG¹¹-Dateien mit der Endung eines JPEG¹² abgespeichert werden, was von vielen Bildbetrachtungsprogrammen akzeptiert wird, weil beides Bildformate sind.

Die Interpretation der Daten muss allerdings komplett unterschiedlich erfolgen, um erfolgreich sein zu können, da der interne Aufbau anders ist.

Auf der anderen Seite können Dateiendungen auch weggelassen werden, wenn die Programme, welche die Dateien verarbeiten nicht anhand der Dateiendung, sondern anhand der magischen Zahl (vergleiche Abschnitt 2.11.2 auf der nächsten Seite) versuchen, den Dateiinhalt zu ermitteln.

Ein weiteres Problem bei Dateiendungen ist die Groß- und Kleinschreibung. Während bei Windows-Systemen die Groß- und Kleinschreibung keinen Unterschied macht und `test.png` und `test.PNG` komplett gleich behandelt werden, ist diese Unterscheidung in Linux-Systemen relevant, sodass eine Vereinheitlichung der Dateiendungen vor der Prüfung notwendig ist.

Doch selbst wenn die Dateiendungen korrekt und in einheitlicher Schreibweise vorhanden sind, besteht noch das Problem, dass Dateiendungen nicht immer eindeutig sind. So können zum einen mehrere Dateiendungen auf das gleiche Dateiformat hinweisen, wie zum Beispiel `.cpp`, `.cxx`, `.cc`, `.c++`, `.cp`, `.C`, `.ii` für C++-Dateien, wobei hier wiederum die Großschreibung von `.C` wichtig ist, um C++- von C-Dateien zu unterscheiden.^[20] Zum anderen können aber auch mehrere Dateiformate für eine Dateiendung möglich sein. So kann eine Datei namens `test.B` sowohl für einen Quelltext in BASIC¹³ stehen, als auch für einen Quelltext in Modula-3 oder einen Quelltext in den esoterischen Programmiersprachen Brainf*ck oder Befunge.^[21] Selbst diese Liste ist noch nicht vollständig, zeigt aber das Problem, dass eine reine Erkennung anhand der Dateiendung zu Problemen bei der Erkennung führen könnte.

Auf der anderen Seite kann man in den meisten Fällen davon ausgehen, dass die Codebasis nicht so stark durchmischt ist, dass es zu einer Überschneidung von Dateiformaten kommt und die passenden Dateiendungen im Zweifelsfall manuell eingetragen werden können.

Eine reine Analyse der Dateiendungen ist aber aus den genannten Gründen nicht sinnvoll, sodass eine zusätzliche Prüfung notwendig ist, um entweder auch unbekannte Dateiendungen korrekt zuzuordnen (wenn zum Beispiel C++-Dateien mit der Endung `.cplusplus` gespeichert werden) oder eine eindeutige Zuordnung zu einem Dateiformat nicht möglich ist, wie bei den `.B`-Dateien beschrieben.

¹¹Portable Network Graphics

¹²Joint Photographic Experts Group

¹³Beginner's All-purpose Symbolic Instruction Code

2.11.2 Magische Zahl

Eine weitere – vorwiegend im Linux-Umfeld verbreitete – Methode zur Erkennung von Dateiformaten ist die Suche nach magischen Zahlen. Damit ist gemeint, dass Dateiformate durch spezielle Zeichenketten oder Zahlenfolgen markiert werden, die anschließend mit einer Datenbank von bekannten Zahlenfolgen verglichen werden, um festzustellen, um was für ein Dateiformat es sich handelt.

Bekannte Beispiele hierfür sind das Shebang in Shellskripten, $0x2321$ (8993_{10}) (`»#!«`), wodurch angedeutet wird, dass die ausführbare Datei, die direkt im Anschluss genannt wird, als Interpreter für das Skript genutzt werden soll.

Allerdings kann es auch bei der Nutzung magischer Zahlen zu Verwechslungen von Dateiformaten kommen, insbesondere, wenn es sich um kurze oder unvollständige Inhalte handelt, wie im Fall einer ausgelagerten und mittels `#include` in C/C++ eingebundenen Anweisung, wie Listing 2.8 und Listing 2.9 als Extrembeispiel verdeutlichen.

Listing 2.8: Include in einer Funktion in C

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4 #include "include_statement.c"
5 }
```

Listing 2.9: Ausgelagerte Statements in C

```
1 printf("Testprogramm.\n");
2 return 0;
```

Beide Dateien zusammen ergeben die erwartete Ausgabe »Testprogramm.«, das Ergebnis des Tools `file`, eines traditionellen Unix-Kommandos, welches anhand der magischen Zahl versucht, das Dateiformat einer Datei zu erkennen, ergibt allerdings folgende Ausgabe:

Listing 2.10: Ausgabe von `file`

```
1 include_statement.c: ASCII text, with CRLF line terminators
2 main_include.c:      C source, ASCII text, with CRLF line
   terminators
```

Würde man sich in diesem Fall also für die Analyse nur auf die von `file` als C-Dateien erkannten Dateien beschränken, würde der Inhalt der Datei `include_statement.c` nicht untersucht und Sicherheitslücken hierin blieben unentdeckt.

Zwar ist eine derartige Nutzung von `#include`-Anweisungen in der Regel eher unüblich, da allerdings in der Theorie beliebige Programmiersprachen durch das Framework untersucht

werden können sollen, ist nicht auszuschließen, dass es Programmiersprachen gibt, in welchen eine derartige Herangehensweise üblich ist und die ebenfalls nicht korrekt durch `file` erkannt werden würden.

Ein weiteres Problem kann auftreten, wenn mehrere Dateiformate in einer Datei gemischt auftauchen, wie es häufiger mit HTML¹⁴, CSS¹⁵ und JavaScript der Fall ist. So wird die Datei aus Listing 2.12 lediglich als

```
1 multiple_filetypes.html: HTML document, ASCII text
```

erkannt und sogar bei einer Aufteilung in separate Dateien, wie in Listings 2.13 bis 2.15 auf den Seiten 37–38 werden die einzelnen Dateiformate nicht korrekt erkannt:

Listing 2.11: Das Resultat von `file` bei separaten Dateien

```
1 separate_css.css: ASCII text
2 separate_filetypes.html: HTML document, ASCII text
3 separate_javascript.js: ASCII text
```

Listing 2.12: Mehrere Dateiformate in einer Datei

```
1 <!DOCTYPE html>
2 <html lang="de">
3 <head>
4     <meta charset="utf-8">
5     <title>Mehrere Dateitypen in einer Datei</title>
6     <style>
7         body {
8             background: red;
9         }
10    </style>
11 </head>
12 <body>
13     <script>alert(1);</script>
14 </body>
15 </html>
```

Listing 2.13: Separate Aufteilung in mehrere Dateien pro Dateiformat

```
1 <!DOCTYPE html>
2 <html lang="de">
3 <head>
```

¹⁴Hypertext Markup Language

¹⁵Cascading Style Sheets

```

4     <meta charset="utf-8">
5     <title>Separate Filetypes</title>
6     <link rel="stylesheet" href="./separate_css.css">
7 </head>
8 <body>
9     <script src="./separate_javascript.js"></script>
10 </body>
11 </html>

```

Listing 2.14: Separates CSS in einer Datei

```

1 body {
2     background: red;
3 }

```

Listing 2.15: Separates JavaScript in einer Datei

```

1 alert(1);

```

Insbesondere wenn die systemeigene Magic-Datei benutzt wird, kann es daher nicht nur zu für den Anwender schwer zu erkennenden Inkonsistenzen zwischen verschiedenen Systemen, sondern auch zu einer Nichterkennung verschiedener Dateiformate kommen, wodurch Sicherheitslücken in diesen übersehen werden könnten.

2.11.3 Vergleich der beiden Methoden

Wie aus den obigen Ausführungen ersichtlich wird, ist keine der beiden Methoden in jedem Fall zuverlässig und geeignet, das Dateiformat immer korrekt zu erkennen.

Um genau zu sein, ist eine eindeutige Erkennung des Dateiformats auch alleine schon deshalb nicht möglich, weil dieselbe Datei auf unterschiedliche Arten interpretiert werden kann, was im einfachsten Fall durch miteinander kompatible Interpreter/Compiler zu einem ähnlichen sichtbaren Ergebnis führt, bei Programmiersprachen mit gleicher Syntax, aber unterschiedlicher Semantik jedoch zu komplett gegenläufigen Resultaten führen kann. Auch dieser Fall ist in der Praxis eher unüblich, muss jedoch für eine genaue Betrachtung beachtet und abgeschätzt werden, um die richtige Vorgehensweise zu wählen.

Da dieses Problem durch die bloße Betrachtung des Dateinamens noch weit dramatischer ist als bei der Betrachtung der magischen Zahl, weil, wie in Abschnitt 2.11.1 auf Seite 34 beschrieben, die Dateiendung lediglich ein Metadatum ist, welches keinerlei Rückschlüsse auf den Inhalt der Datei selbst zulässt, sollte vorwiegend die magische Zahl zur Erkennung des Dateiformats verwendet werden.

Um wiederum deren Schwachstellen auszugleichen, sollte im ersten Schritt geprüft werden, ob eine bekannte Dateiendung existiert und in diesem Fall versucht werden, anhand

derer das Dateiformat zu erkennen, da erfahrungsgemäß Entwickler meistens die üblichen Dateiendungen verwenden.

Sollte dagegen die magische Zahl keine Ergebnisse liefern und auch keine Dateiendung vorhanden sein, gibt es wiederum zwei Möglichkeiten, mit dem Problem umzugehen.

Als Erstes ist es möglich, anhand einer Heuristik zu versuchen, das Dateiformat zu erraten. So ist davon auszugehen, dass in einem Verzeichnis mit mehreren hundert C++-Dateien eine von der magischen Zahl unerkannte und ohne Dateiendung abgelegte Datei vermutlich am ehesten ebenfalls eine C++-Datei sein könnte.

Hierdurch ergibt sich wiederum die neue Frage, ob nur die Dateiformate der Dateien aus dem aktuellen Ordner oder auch die aller weiteren in die Untersuchung einbezogenen Ordner Einfluss auf die Heuristik nehmen sollten.

Da in vielen Programmiersprachen eine Ordnung von Quelldateien in Module ordnerbasiert auf dem Dateisystem geschieht, werden für das Framework sämtliche untersuchten Dateien in die Heuristik einbezogen.

Als Alternative zu dieser Heuristik muss es jedoch auch möglich sein, dem Framework als Anwender mitzuteilen, welches Dateiformat im Zweifelsfall genutzt werden soll.

Dieses erzwungene Dateiformat würde dann die heuristische Erkennung ausschalten und statt dieser genutzt werden.

2.12 Erstellung von Regelsätzen

Die zugrunde liegenden Regelsätze sind eines der wichtigsten Kriterien, welche die Qualität der Erkennung beeinflussen.

Schwachstellen, für die kein Regelsatz existiert, können von der statischen Analyse nicht korrekt als solche erkannt werden, wenn Regeln zu spezifisch sind, können leichte Abänderungen im Quelltext, die keine Auswirkungen auf die Sicherheit haben, dafür sorgen, dass die Erkennung fehlschlägt und wenn die Erstellung und Wartung der Regeln so aufwendig ist, dass nur die mitgelieferten Regeln verwendet werden, verliert die Analyse schnell an Bedeutung, da neue Angriffsmuster nicht erkannt und durch andere Produkte gesucht werden müssen.

Aus diesem Grund ist es wichtig, zuerst zu klären, welchen Inhalt eine Regel überhaupt haben sollte und welche Details notwendig sind, um eine Erkennung zu gewährleisten, ohne die Qualität durch zu hohe Spezialisierung zu gefährden oder auf der anderen Seite durch zu ungenaue Regeln eine Vielzahl an Falschmeldungen zu produzieren.

In der eigentlichen Regel müssen dann sowohl die verwundbare Funktion selbst als auch die dazu passende Absicherung enthalten sein, um sicherzustellen, dass die Zuordnung

zwischen den beiden klar ist, da zum Beispiel die Absicherung vor SQL-Injections anders abläuft als vor XSS-Angriffen.

Auch ist es wichtig, die Benutzereingaben separat zu beschreiben, da diese unabhängig von den verwundbaren Funktionen und Absicherungen sind und lediglich anzeigen können, dass die Ausnutzung durch einen Angreifer vermutlich auf direktem Weg möglich ist.

Zudem ist wichtig festzulegen, ob die Regeln in einer einzelnen Datei abgespeichert werden, was die Installation und Weitergabe vereinfacht oder sie in mehrere Dateien aufzuteilen, wodurch eine einfachere Versionskontrolle der Regelsätze möglich ist.

Auch der strukturierte Aufbau der Regeln ist wichtig, da die Regeln programmatisch ausgelesen und interpretiert werden können müssen.

Damit Anwender später eigene Regelsätze anlegen können, um mit diesen ihre Quelltexte zu überprüfen, darf die Erstellung neuer Regeln – wie bereits angesprochen – ebenfalls nicht zu komplex werden.

2.12.1 Inhalt eines Regelsatzes

Im einfachsten Fall müsste ein Regelsatz lediglich Informationen darüber enthalten, wie eine Benutzereingabe in eine potenziell unsichere Funktion eingeschleust werden könnte und ob eine Absicherung der Benutzereingaben stattfindet.

Dieser Ansatz erscheint allerdings problematisch, da die Auswertung der Resultate immer von einem Menschen vorgenommen werden muss. Aus diesem Grund erscheint es sinnvoll, zumindest auch eine kurze Erklärung des Regelsatzes in Form eines Kommentars zu ermöglichen.

So könnte zu der Funktion `input()` in Python ein kurzer Erklärungstext mitgegeben werden, dass hierdurch beliebige durch einen Zeilenumbruch begrenzte Benutzereingaben an das Programm übergeben werden können, wobei in Python 2 eine direkte Auswertung der Eingabe vorgenommen wird, sodass diese Funktion in Python 2 selbst bereits eine Sicherheitslücke darstellt.[\[22\]](#)

Auch im Fall von verwundbaren Funktionen kann es wichtig sein, eine kurze Erklärung der Risiken mitzugeben, da es einem Benutzer hiermit möglich ist, selbst besser abzuschätzen, ob die Bedrohung auf das vorliegende Szenario anwendbar ist oder nicht.

Weiterhin muss angegeben werden, ob ein spezieller Parameter der Funktion anfällig für Angriffe ist oder die komplette Funktion, wie in Abbildung 2.11 auf der nächsten Seite an einem Beispiel gezeigt.

Da Absicherungen kontextbezogen sein sollten und es unter Umständen nicht immer möglich ist, die für den konkreten Anwendungsfall korrekte Absicherung in einer Regel ausreichend zu definieren, kann es hilfreich sein, eine kurze Erklärung der Absicherung zu erlauben und dort aufzuschreiben, in welchen Szenarien sie nützlich ist.

So ist zum Beispiel bei XSS-Injections eine Standardmethode zur Absicherung das HTML Entity Encoding, bei dem sämtliche HTML-Zeichen in eine gleichwertige, aber nicht vom Browser interpretierte Darstellung umgewandelt werden. Diese Absicherung kann allerdings unter Umständen nicht ausreichend sein, wenn nur einige der Zeichen kodiert werden, wie zum Beispiel in Python 3s `html.escape`, welches standardmäßig nur die Zeichen `&`, `<` und `>` umwandelt.^[23] Setzt man diese Kodierung allerdings in einem Kontext ein, in dem HTML-Attribute durch Benutzereingaben beeinflusst werden oder wenn man die Eingaben für GET-Parameter einsetzt, schützt die Kodierung nicht vor Angriffen.^[24]

Auch in Fällen, in denen eine Absicherung nicht möglich ist, wie bei der oben angesprochenen `input`-Funktion in Python 2 kann ein Kommentar hilfreich sein, der darauf hinweist, dass diese Funktion auf keinen Fall genutzt und durch die `raw_input`-Funktion ersetzt werden sollte, da ein Angreifer sonst recht einfach über Eingaben wie in Abbildung 2.11 beliebigen Code ausführen könnte.

```
/tmp » cat ./python2_input_vuln.py 11:07
#!/usr/bin/env python2
input()
/tmp » echo '__import__("os").system("uname -a")' | ./python2_input_vuln.py 11:07
Linux kali 4.18.0-kali2-amd64 #1 SMP Debian 4.18.10-2kali1 (2018-10-09) x86_64 GNU/Linux
/tmp » █ 11:07
```

Abbildung 2.11: Die `input`-Funktion in Python 2 erlaubt Command Injections.

Aus diesem Grund scheint es sinnvoll, zumindest noch ein optionales Kommentarfeld zu den jeweiligen Einträgen zu erlauben.

2.12.1.1 Definition von Benutzereingaben

Da Benutzereingaben unabhängig von verwundbaren Funktionen sind, müssen sie als allgemein verwendbarer Teil des jeweiligen Moduls für die einzelnen Programmiersprachen abgespeichert werden.

Benutzereingaben werden dabei häufig über Funktionen, wie in Abbildung 2.11 implementiert, können aber auch als Parameter beim Programmstart – wie der Parameter `char* argv[]` in der `main`-Methode von C- und C++-Programmen – implementiert werden.

Es ist hierbei wichtig zu beachten, dass Benutzereingaben nicht auf offensichtliche Eingaben wie in Abbildung 2.11 beschränkt sind, sondern auch durch eingelesene Dateien, API¹⁶-Schnittstellen oder vom Benutzer kontrollierte Umgebungsvariablen in ein Programm eingeschleust werden können.

Die Erkennung dieser Typen von Eingaben unterscheidet sich dabei deutlich:

¹⁶Application Programming Interface

1. Bei direkten Benutzereingaben muss geprüft werden, ob eine entsprechende Funktion genutzt wird.
2. Bei eingelesenen Dateien muss geprüft werden, ob der Benutzer Zugriff auf diese hat.
3. Bei API-Schnittstellen müssen die Parameter einer Funktion selbst betrachtet werden.
4. Bei Umgebungsvariablen muss geprüft werden, ob spezielle Variablen genutzt werden, falls die Programmiersprache diese wie im Fall von Bash und anderen Skriptsprachen direkt interpretieren, ohne eine entsprechende Funktion zu benötigen.

Leider lassen sich nur die ersten beiden Verfahren verallgemeinern, indem die entsprechenden Funktionen zum Einlesen von Benutzereingaben und zum Öffnen bzw. Lesen von Dateien in einer Standardliste aufgenommen werden.

Zusätzlich sollte daher eine projektbezogene Liste existieren, in der API-Schnittstellen und genutzte Umgebungsvariablen aufgelistet sind, damit auch deren Verwendung hinreichend geprüft werden kann.

Um allerdings das Risiko durch eine fehlende Erkennung von Benutzereingaben zu senken, sollte die Erkennung einer Eingabe, die in eine verwundbare Funktion führt, lediglich als Erhöhung des erkannten Risikos gewertet werden und nicht die Voraussetzung dafür sein, dass die verwundbare Funktion im Risikoreport aufgeführt wird.

2.12.1.2 Definition verwundbarer Funktionen

Bei der Definition von verwundbaren Funktionen gibt es zwei wichtige Unterscheidungen:

1. Funktionen, die inhärent verwundbar sind, wie in Abbildung 2.1 auf Seite 5 und
2. Funktionen, die nur dann verwundbar sind, wenn an der falschen Stelle Variablen als Eingabe genutzt werden, wie in Listing 2.3 auf Seite 21

Aus diesem Grund muss die Definition verwundbarer Funktionen es erlauben, nicht nur einen Funktionsnamen anzugeben, sondern auch, welcher Parameter der Funktion verwundbar ist, wenn er mit unkontrollierten Eingaben gespeist wird.

So ist der folgende PHP-Code nicht verwundbar, obwohl die Funktion `mysql_query`, wie in Listing 2.3 auf Seite 21 gezeigt, durchaus problematisch sein kann.

Listing 2.16: Nicht verwundbare Nutzung von `mysql_query`

```
1 mysql_query("SELECT * FROM users WHERE name = 'admin'");
```

Solange hier an keiner Stelle eine Benutzereingabe in die Anweisung einfließt, kann sie nicht direkt für Angriffe verwendet werden.

Auf der anderen Seite ist es teilweise relevant, an welcher Stelle Benutzereingaben in Funktionen eingespeist werden, um zu unterscheiden, ob es sich um eine mögliche Sicherheitslücke handelt oder nicht.

Als Beispiel hierfür soll die in PHP 5 und 7 vorhandene Funktion **assert** dienen.[25]

Listing 2.17: Prototyp der **assert**-Funktion in PHP 5 und 7

```
1 bool assert(mixed $assertion [, string $description ])
```

Wird hierbei die Assertion selbst von einem Angreifer kontrolliert, so kann er potenziell beliebigen Code ausführen, wie zum Beispiel

Listing 2.18: Code Execution durch Assertion

```
1 assert('shell_exec("uname -a"); true;');
```

Kontrolliert ein Angreifer dagegen nur den zweiten Parameter, die Beschreibung der Assertion, so kann er höchstens versuchen, seinen Text als Code in Logdateien einzuschleusen und darauf hoffen, dass das zur Betrachtung der Logdateien verwendete Programm die Einträge nicht als unsichere Eingaben behandelt, eine Technik, die allgemein als »Log Poisoning« bezeichnet wird.[26]

Derartige indirekte Angriffe können zwar ebenfalls in einen Regelsatz aufgenommen werden, es sollte aber hieran klar geworden sein, dass es einen großen Unterschied macht, welcher Parameter von einem Angreifer kontrolliert wird, um festzustellen, wie groß der mögliche Schaden ist.

Die Abwehr des ersten Angriffs wird häufig auch als »Output Sanitization« bezeichnet, womit gemeint ist, dass Daten vor der Weiterverwendung oder der Ausgabe an den Benutzer bereinigt werden, da die Form der Bereinigung stark von der Weiterverarbeitung abhängt.[27]

Auf der anderen Seite würde es sich bei der Abwehr des zweiten Angriffs um eine »Input Sanitization« handeln, also eine Bereinigung der Daten direkt nach dem Eingang, was nur in seltenen Fällen sinnvoll möglich ist, ohne die Art der Weiterverarbeitung zu kennen.[27]

Aus diesem Grund wird die Erstellung von Regelsätzen zur »Output Sanitization« empfohlen und es werden keine Regelsätze zur »Input Sanitization« im Rahmen dieser Masterarbeit erstellt und mit dem Framework ausgeliefert.

2.12.1.3 Definition von Absicherungen

Absicherungen stehen vor verwundbaren Funktionen und sorgen dafür, dass die Eingaben, welche in die verwundbaren Funktionen eingespeist werden, in der verwundbaren Funktion keinen Schaden anrichten können.

Ebenso wie bei der Definition verwundbarer Funktionen muss natürlich auch bei Absicherungen geprüft werden, welcher Parameter durch sie bereinigt wird.

Als einfaches Beispiel sei wieder die Funktion `html.escape` von Python 3 genannt.

Wenn diese Funktion aufgerufen wird, bevor die Benutzereingabe in einer Webseite angezeigt wird, kann somit verhindert werden, dass Code von einem Angreifer als Code interpretiert wird und stattdessen durch die Umwandlung in die HTML Entity Encoding der Code als Daten interpretiert wird, wodurch ein Angriff abgewehrt wird.

Hierzu ergeben sich allerdings wieder einige wichtige Beobachtungen, ohne die eine korrekte Erkennung von Absicherungen nicht möglich ist.

- Die verwundbare Funktion und die Absicherung müssen zwingend zusammenpassen, ansonsten kann die Absicherung nicht greifen, wie in Abschnitt 2.12.1 auf Seite 40 beschrieben, wo der Aufruf der Funktion `html.escape` zwar im Normalfall hilft, wenn komplette HTML-Tags gefiltert werden sollen, nicht mehr aber, wenn auch Attribute abgesichert werden sollen.
- Die Menge an Absicherungen ist nicht zwangsläufig während der kompletten Analyse konstant. Dies liegt daran, dass während der Analyse unter Umständen festgestellt wird, dass Funktion *A* zuerst Funktion *B* und anschließend eine verwundbare Funktion aufruft. Wenn Funktion *B* nun allerdings eine Absicherung für die in Funktion *A* verwendete verwundbare Funktion beinhaltet und so aufgerufen wird, dass die Eingabe für die verwundbare Funktion in *A* zuerst durch Funktion *B* gefiltert wird, dann ist Funktion *B* ebenfalls eine indirekte Absicherung für die verwundbare Funktion aus *A* und muss entsprechend bewertet werden.

Der erste Punkt führt dazu, dass verwundbare Funktionen und zugehörige Absicherungen zwangsläufig eine gemeinsame Verbindung benötigen, sodass es am sinnvollsten erscheint, sie gemeinsam zu definieren.

Der zweite Punkt bedeutet, dass ein einmaliger Scan aller Funktionen vermutlich nicht ausreichend sein wird, da hierdurch indirekt absichernde Funktionen nicht immer korrekt erkannt werden.

Es ist daher notwendig, zuerst einen Scan auf verwundbare und absichernde Funktionen zu machen und wenn verwundbare Funktionen gefunden wurden, muss im nächsten Schritt geprüft werden, ob vor deren Aufruf in der Senke die Verschmutzung – direkt oder indirekt – abgesichert wird.

Auf der anderen Seite kann gerade auch in der sicheren Entwicklung das Prinzip »Fail Fast«, nach dem Systeme bei Fehlern möglichst frühzeitig abbrechen sollten, um die Fehlererkennung zu vereinfachen und Fehlerfortpflanzung zu vermeiden,[28] als Best Practice angesehen werden, weshalb eine Absicherung über mehrere Stufen zu einer höheren Risikoeinschätzung führen sollte, als eine möglichst direkte Absicherung.

Um dies mathematisch darzustellen, könnte die Erkennung einer verwundbaren Funktion zuerst einen Risikofaktor von 50 Prozent bedeuten. Wenn zusätzlich noch eine Benutzereingabe existiert, deren Ergebnisse in sie eingespeist werden, steigt der Risikofaktor um 50 Prozentpunkte auf 100 Prozent.

Nimmt man nun eine Absicherungsfunktion dazu, sollte die Risikoeinschätzung natürlich entsprechend sinken, zum Beispiel bei direkter Absicherung um 50 Prozentpunkte zurück auf 50 Prozent.

Je mehr Stufen zwischen der Absicherung und der verwundbaren Funktion liegen, desto höher sollte das Risiko eingeschätzt werden.

Für das Framework wird die folgende Funktion eingesetzt, sodass bei einer direkten Absicherung keine Beeinträchtigung des Risikoabfalls vorliegt, das Risiko aber mit steigender Anzahl indirekter Absicherungsfunktionen immer weiter wächst.

$$\text{Risiko}(x) = \underbrace{0,5}_{\text{Verwundbare Funktion}} + \left(\frac{\min \left(\underbrace{\frac{x}{5}}_{\text{maximaleDistanz}}, 1 \right)}{2} \right)$$

Eine grafische Darstellung dieser Funktion ist in Abbildung 2.12 zu finden.

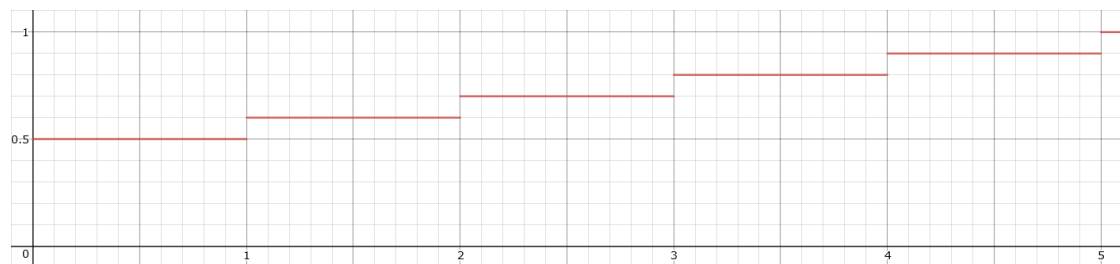


Abbildung 2.12: Wachstumsfunktion für das Risiko durch indirekte Absicherung

Eine Benutzereingabe ohne Absicherung wird hierbei wie eine Absicherung mit maximaler Distanz behandelt, eine verwundbare Funktion ohne Benutzereingabe wie eine direkte Absicherung.

Da eine Funktion entweder direkt aufgerufen werden kann oder nicht, ergibt sich eine Wachstumsfunktion, in der die Distanz als Integer übergeben wird, um das Risiko zu errechnen.

Damit ergibt sich für eine Funktion, in der eine verwundbare Funktion mit einer Benutzereingabe aufgerufen wird und wo die Benutzereingabe sofort abgesichert wird, noch ein Restrisiko von 0,5 oder 50 Prozent.

Der gleiche Wert wird dadurch erreicht, dass eine verwundbare Funktion existiert, die allerdings nicht von einer Benutzereingabe betroffen ist.

Wenn dagegen eine verwundbare Funktion mit Benutzereingabe existiert, deren Benutzereingabe erst in zwei Unterfunktionen bereinigt wird, so beträgt das Restrisiko noch 0,7 oder 70 Prozent.

Es ist hierbei nicht möglich, einen Risikowert von 0 Prozent zu erreichen, wenn verwundbare Funktionen genutzt werden. Dies ist eine bewusste Entscheidung, da in den meisten Fällen Alternativen existieren, die statt potenziell verwundbarer Funktionen eingesetzt werden sollten.

Eine entsprechende Ab- oder Aufwertung des Risikos durch weiter entfernte Benutzereingaben scheint auf der anderen Seite nicht sinnvoll.

Um diese Entscheidung allerdings dem Benutzer zu überlassen, wird die maximale Entfernung als optionaler Parameter mit dem Standardwert von 5 bei der Analyse vom Benutzer abgefragt.

Wenn keine Alternativen existieren, sollte den Entwicklern stets bewusst sein, dass sie mit der Nutzung der jeweiligen Funktion sehr vorsichtig sein müssen, um keine Sicherheitslücken zuzulassen.

Auf der anderen Seite können Benutzereingaben zwar vollständig abgesichert werden, da jedoch die Erkennung von Benutzereingaben selbst fehleranfällig sein kann, kann auch eine Absicherung sämtlicher Eingaben in die verwundbare Funktion nicht dafür sorgen, dass die Nutzung dieser Funktion in jedem Fall sicher ist.

2.12.2 Speicherung einzelner Regeln

Da ein regelbasiertes Framework zur statischen Analyse in hohem Maß von der Qualität der hinterlegten Regeln abhängig ist, müssen alle diese Regeln betreffenden Eigenschaften genau überlegt und miteinander verglichen werden, um eine möglichst gute Lösung zu erreichen.

Hierbei ist eine wichtige Frage, wie die einzelnen Regeln im Framework abgespeichert werden sollen.

Auf der einen Seite ist die Speicherung in einer einzelnen großen Regeldatei pro Modul möglich, auf der anderen Seite könnte es sinnvoller sein, mehrere kleine Regeldateien zu

erstellen, die unabhängig voneinander bearbeitet und erweitert bzw. wenn notwendig sogar entfernt werden können.

Im Folgenden werden daher die Vor- und Nachteile dieser beiden Methoden jeweils kurz aufgeführt und die beiden Ansätze miteinander verglichen.

2.12.2.1 Speicherung in einer großen Regeldatei

Eine Speicherung sämtlicher Regeln in einer Datei hätte den Vorteil, dass der Transport dieser Datei einfacher möglich ist.

Statt mehrere hundert kleine, müsste in diesem Fall nur noch eine große Datei übertragen werden, um Regelsätze zu aktualisieren.

Dies hätte sowohl beim Transport selbst durch den Overhead, als auch beim Parsen der Dateien einen kleinen Geschwindigkeitsvorteil, da der Overhead in beiden Fällen einmalig wäre, statt bei jeder einzelnen Regeldatei aufzutreten.

Ein weiterer Vorteil einer großen Regeldatei wäre die mögliche Kategorisierung innerhalb der Datei. Man könnte hier zum Beispiel einzelne Abschnitte in der Datei unterbringen, welche nach SQL-Injections, XSS-Schwachstellen oder sonstigen suchen, und müsste, wenn ein Treffer genauer untersucht oder eine Regel korrigiert werden muss, nur in der entsprechenden Kategorie suchen, um die betreffende Stelle zu finden.

Auf der anderen Seite ist es bei großen Regeldateien schwieriger, Aktualisierungen nachzuvollziehen, wenn keine Versionsverwaltungssoftware eingesetzt wird.

Lediglich eine große Datei zu aktualisieren könnte dafür sorgen, dass Benutzer gehemmt sind, eigene Änderungen an der Datei vorzunehmen, da sie befürchten müssen, dass ihre eigenen Änderungen an dieser beim nächsten Update überschrieben werden oder sie manuell die Änderungen zusammenfügen müssen.

Weiterhin könnten sie hierdurch den Überblick über noch fehlende Regeln verlieren, da sie somit zwar alle Regeln auf einen Blick haben, aber nicht ohne weiteres sehen können, zu welcher Kategorie ihnen noch Regeln fehlen.

Auch die Erweiterung um eigene Regeln würde durch eine große Regelsatzdatei erschwert, da die eigenen Regeln unter Umständen projektabhängig sind, weil zum Beispiel Frameworks eingesetzt werden, die ihre eigenen Funktionen zur Benutzereingabe, eigene verwundbare Funktionen und eigene Absicherungen unterstützen und die – entweder aus Gründen der Zeitersparnis oder weil der Quelltext nicht vorliegt – nicht bei jedem Scan mit untersucht werden sollen.

2.12.2.2 Speicherung in mehreren kleinen Dateien

Die Speicherung einzelner Regeln in kleinen Dateien hat den Vorteil, dass es sehr einfach ist, einzelne Regelsätze hinzuzufügen, Änderungen vorzunehmen und nachzuvollziehen und Regeln bei Bedarf zu entfernen.

Auch das Debuggen von Regeln wird vereinfacht, wenn neben der Ausgabe der Schwachstelle auch der Name der Regeldatei, durch welche die Schwachstelle entdeckt wurde, ausgegeben wird.

Die in Abschnitt 2.12.2.1 auf der vorherigen Seite angesprochenen Einbußen an Geschwindigkeit sollten vernachlässigbar sein, sofern es sich nicht um tausende einzelne Dateien handelt.

Auf der anderen Seite hätte die Speicherung in Einzeldateien vor allem den Nachteil, dass es schwierig ist, einen Überblick über zu viele einzelne Dateien zu bewahren.

Aus diesem Grund erscheint es sinnvoll, nicht nur ein Verzeichnis mit allen Regeldateien zu erstellen, sondern beliebige Unterverzeichnisse zu erlauben, die dann zum Beispiel thematisch geordnet sein können.

Hiermit wäre es einem Benutzer möglich, schnell zwischen bestimmten Sicherheitslücken zu wechseln und somit gezielt nach Problemen zu suchen.

Auch eine Einteilung nach Priorität der Schwachstelle, also zum Beispiel eine niedrige Priorität bei Information Disclosures, eine hohe Priorität bei Code Executions, wäre denkbar und könnte somit vom Benutzer frei nach den eigenen Bedürfnissen gestaltet werden.

2.12.2.3 Vergleich der beiden Ansätze

Nach Abwägung der jeweiligen Vor- und Nachteile scheint es sinnvoller, die Regelsätze in mehrere kleine Dateien aufzuteilen, welche nach Bedarf in Unterverzeichnissen kategorisiert werden können.

Eine Vorgabe für die Kategorisierung wird hierbei nicht gemacht, damit die Benutzer selbst die für sie optimale Kategorisierung herausfinden können.

Hiermit lässt sich der Vorteil der großen Regeldatei – die einfache Kategorisierung – und der Vorteil der kleinen Regeldateien – die bessere Änderbarkeit und Nachvollziehbarkeit – mit minimalen Nachteilen – dem Zusatzaufwand durch den Overhead vieler kleiner Dateien – gut miteinander kombinieren.

2.12.3 Mögliche Formate für den Aufbau von Regeln

Das Format für den Aufbau von Regeln ist essenziell wichtig für den Erfolg des Projekts.

Zuerst muss das Format in einer einheitlichen Struktur vorliegen, um möglichst einfach vom Framework eingelesen und verwertet werden zu können.

Weiterhin ist es allerdings auch wichtig, dass das Format zum einen mächtig genug ist, um sämtliche Anforderungen, also die Repräsentation von Benutzereingaben (Quellen), verwundbaren Funktionen (Senken) und Absicherungen ausreichend genau darzustellen.

Zum anderen muss es jedoch auch einfach und verständlich genug sein, dass ein typischer Benutzer in der Lage ist, mit möglichst wenig Einarbeitungszeit eigene Regelsätze zu erstellen oder vorhandene Regelsätze anzupassen.

Außerdem steigt bei komplexeren Formaten zur Speicherung strukturierter Daten die Gefahr von Sicherheitslücken beim Parsen.

Daher könnte ein Angreifer, der in der Lage ist, eigene Regelsätze auf einem Server zu veröffentlichen, eine Sicherheitslücke im Regelsatzparser ausnutzen, um somit die Kontrolle über den Server zu übernehmen.

Natürlich lassen sich derartige Sicherheitslücken durch saubere Programmierung verhindern, die zusätzliche Komplexität der Lösung erschwert allerdings auch die Testbarkeit und sollte daher gut überlegt sein.

Aus diesem Grund werden im Folgenden einige verbreitete Formate für die Speicherung strukturierter Daten vorgestellt und miteinander verglichen, um eine Lösung zu finden, welche sämtliche Anforderungen hinreichend erfüllt und dabei eine möglichst simple Struktur hat, sodass sie auch für unbedarfte Nutzer übersichtlich ist.

Für einen anschaulichen Vergleich der möglichen Formate werden jeweils Beispiele für Benutzereingaben, verwundbare Funktionen und deren zugehörige Absicherungen dargestellt.

Da davon auszugehen ist, dass Regelsätze auch untereinander ausgetauscht werden und daher nicht in jedem Fall garantiert werden kann, dass die Autoren vertrauenswürdig sind, werden auch, wo vorhanden, Schwachstellen und mögliche Sicherheitslücken durch die verschiedenen Formate aufgezeigt und kurz angesprochen, wie man diese verhindern könnte.

Als Nebeneffekt dieser Betrachtung kann man hierdurch auch direkt weitere Regeln für die Erkennung von Schwachstellen und deren Absicherungen generieren, auch wenn die Formate nicht für die Speicherung von Regeln genutzt werden.

2.12.3.1 CSV

Eines der einfachsten Formate zur Speicherung strukturierter Daten, welches eine sehr hohe Verbreitung aufweist, ist das CSV¹⁷-Format.

Wie der Name bereits andeutet, werden hier einzelne Werte von Kommas (standardmäßig, alternativ sind auch Semikolons, Tabs und weitere Trennzeichen möglich und üblich) in einer Textdatei gespeichert.

Die Einfachheit des Formats sowie die hohe Verbreitung sorgen hierbei dafür, dass eine Vielzahl an Programmen existiert, welche die komfortable Bearbeitung von CSV-Dateien ermöglichen.

CSV-Dateien erlauben dabei eine Speicherung von Daten in einzelnen Zeilen, wobei jede Zeile einen separaten Datensatz darstellt.

Optional ist auch die Speicherung von Kennungen für die einzelnen Felder in der ersten Zeile möglich.

Durch die Trennung mit lediglich einem (beliebigen) Trennzeichen sind CSV-Dateien ohne dafür ausgelegte Werkzeuge ab einer bestimmten Größe nur noch mit viel Mühe lesbar.

Weiterhin kann es zu Problemen mit Zeichensätzen, wechselnden Trennzeichen, innerhalb eines Feldes verwendeter Trennzeichen und leeren Feldern kommen.

Vor allem letzteres kann problematisch werden, wenn mehrere optionale Felder durch nicht optionale Felder getrennt sind. In diesem Fall müsste das Framework raten, ob es sich bei dem aktuellen Feld um ein optionales Feld handelt oder nicht.

Listing 2.19: Definition von Benutzereingaben für Python 3

```
1 Objekttyp;Funktionsname;Parameter;Parameterwert;Kommentar
2 urlparse;query;None;None;GET Request bei Nutzung von urlparse
```

Listing 2.20: Definition von verwundbaren Funktionen und Absicherungen für Python 3

```
1 Objekttyp;Funktionsname;Absicherungsobjekt;
  Absicherungsfunktionsname;Absicherungsparameter;
  Absicherungsparameterwert;Kommentar
2 Template;substitute;html;escape($VARIABLE, $QUOTE);$QUOTE;/((
  quote=)?True/;Gibt parametrisierte Texte aus, Gefahr von XSS
```

An diesen beiden Beispielen sieht man das Format, in dem ein Regelsatz definiert werden kann.

Nach einer Beschreibungszeile, die lediglich Metadaten enthält, können der Objekttyp – sofern anwendbar, ansonsten wird das Schlüsselwort `None` verwendet – der Name der

¹⁷Comma-Separated Values

Funktion mit Parametern, der relevante Parameter für die Auswertung der Sicherheitsanalyse und der für die Sicherheit relevante Wert dieses Parameters und schließlich ein Kommentar abgespeichert werden.

Der Objekttyp ist dabei relevant, um Funktionen verschiedener Objekte voneinander zu unterscheiden. Wenn stattdessen **None** eingetragen wird, so wird der Objekttyp ignoriert, was zum Beispiel in Programmiersprachen ohne Objektorientierung sinnvoll ist.

Beim Funktionsnamen wird der komplette Aufruf dargestellt, wobei optional auch Parameter mit einem vorangestellten »\$« markiert werden können, die anschließend einen festen Wert zugewiesen bekommen können, der für die Erkennung erforderlich ist.

Zwischen Schrägstrichen geschriebene Werte werden als reguläre Ausdrücke erkannt, wie im Absicherungsparameterwert, in dem der Parameter **\$QUOTE** entweder mit oder ohne vorangestelltes **quote=** stehen kann, wie es bei Python üblich ist, um Schlüsselwortargumente von positionalen Argumenten zu unterscheiden.

Der Parameter **\$VARIABLE** stellt hierbei einen Platzhalter dar, in welchen die zu untersuchende Variable zur Überprüfung der Absicherung eingetragen wird.

Um das Problem der Erkennung optionaler Felder zu behandeln, muss die Beschreibungszeile zwangsläufig benutzt werden, um dem Framework zu ermöglichen, fehlende Spalten korrekt zu erkennen.

Wie bereits angesprochen ist die Verwaltung von CSV-Daten per Hand etwas mühselig, wie bereits an den kurzen Beispielen oben gezeigt.

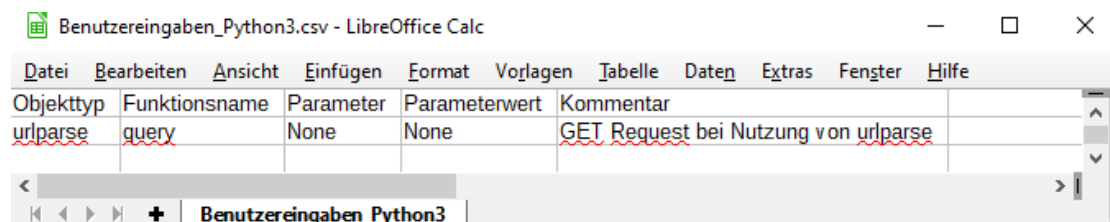


Abbildung 2.13: Definition der Benutzereingaben in Python 3 in LibreOffice

Objekttyp	Funktionsname	Absicherungsobjekt	Absicherungsfunktionsname	Absicherungsparameter	Absicherungsparameterwert	Kommentar
Template	substitute	html	escape(s, \$QUOTE)	\$QUOTE	//quote=?True/	Gibt parametrisierte Texte aus, Gefahr von XSS

Abbildung 2.14: Definition der verwundbaren Funktionen und deren Absicherungen in Python 3 in LibreOffice

Allerdings lassen sich CSV-Daten auch in Programmen wie LibreOffice bearbeiten, wie in Abbildungen 2.13 bis 2.14 auf dieser Seite gezeigt, wodurch die Bearbeitung weitaus angenehmer möglich ist.

2.12.3.2 JSON

Ein weiteres sehr einfaches Format zur Speicherung strukturierter Daten ist JSON¹⁸.

Entgegen dem Namen ist JSON zwar an JavaScript angelehnt, kann jedoch auch in beliebigen anderen Sprachen zum Datenaustausch eingesetzt werden.

JSON ist dabei aussagekräftiger als CSV, da verschiedene Datentypen wie Nullwerte, Boolesche Werte, Zahlen, Zeichenketten, Arrays und Objekte existieren.[\[29\]](#)

Hierdurch ist es für automatisierte Systeme einfacher, die Daten korrekt zu interpretieren.

Ein weiterer Vorteil von JSON ist, dass die Schreibweise vergleichsweise einfach lesbar ist.

Listing 2.21: Definition von Benutzereingaben für Python 3

```
1 {
2     "urlparse":
3     {
4         "Funktionen":
5         [
6             {
7                 "Funktion": "query",
8                 "Parameter": null,
9                 "Parameterwert": null,
10                "Kommentar": "GET Request bei Nutzung von
11                    urlparse"
12            }
13        ]
14    }
```

Listing 2.22: Definition von verwundbaren Funktionen und Absicherungen für Python 3

```
1 {
2     "Template":
3     {
4         "Funktionen":
5         [
6             {
7                 "Funktionsname": "substitute",
8                 "html":
9                 {
```

¹⁸JavaScript Object Notation

```

10         "Funktionsname": "escape($VARIABLE, $QUOTE)
11           ",
12         "Parameter": "$QUOTE",
13         "Parameterwert": "/(quote=)?True/"
14     },
15     "Kommentar": "Gibt parametrisierte Texte aus,
16       Gefahr von XSS"
17 }
18 }

```

An diesen beiden Beispielen sieht man das Format, in dem ein Regelsatz definiert werden kann.

Bei dem gesamten Eintrag handelt es sich um ein Objekt, welches die benötigten Informationen enthält. Innerhalb des Objekts steht der Objekttyp, mit dem gearbeitet wird, gefolgt von einem Array aus Objekten, in denen die genaue Beschreibung der Benutzereingaben bzw. der verwundbaren Funktionen und deren Absicherungen stehen.

Die Schreibweise als Array hat dabei den Vorteil, dass Benutzer die Möglichkeit haben, auch mehrere Funktionen zu definieren, die auf einem Objekt gefährlich sein können, statt pro Funktion eine eigene Regel schreiben zu müssen.

Innerhalb des Arrays werden die Funktionen dann als ein Gesamtobjekt betrachtet, um zu verhindern, dass alle Eigenschaften als separate Einträge im Array gewertet werden.

Das Objekt `html` in der Definition der verwundbaren Funktionen und Absicherungen beschreibt dabei den Objekttyp der Absicherung und dessen Eigenschaften.

Da eine Speicherung von regulären Ausdrücken in JSON nicht vorgesehen ist, werden diese als Strings abgespeichert, welche wieder mittels `»/«` begrenzt werden, um anzudeuten, dass es sich um reguläre Ausdrücke handelt.

Insgesamt erscheint diese Schreibweise sehr leserlich und weniger fehleranfällig in der Interpretation der Daten als CSV.

Für JSON existiert die Erweiterung JSONP¹⁹, durch die JSON-Daten über ein `<script>`-Element eingebunden und inklusive eines Funktionsaufrufs ausgegeben werden können.^[30]

Diese Erweiterung ist allerdings zum einen gefährlich, da mit ihr Daten über Domaingrenzen hinweg ausgetauscht werden können, zum anderen ist sie aber auch nicht erforderlich für die Zwecke des Frameworks, weshalb sie, wenn notwendig, bei der Entwicklung global deaktiviert werden sollte.

¹⁹JavaScript Object Notation with Padding

2.12.3.3 YAML

Obwohl JSON einerseits besser lesbar ist als CSV, ist die Lesbarkeit doch andererseits durch einige syntaktische Eigenheiten, wie die Verwendung von geschweiften Klammern für Objekte und eckigen Klammern für Arrays sowie die explizite Zusammenfassung von Einträgen innerhalb eines Arrays als Objekte, teilweise eingeschränkt und kann zu Verwirrungen und Syntaxfehlern führen.

Eine Serialisierungssprache, welche versucht, eine bessere Lesbarkeit für Menschen zu gewährleisten und gleichzeitig weiterhin einfach zu implementieren und zu nutzen zu sein, ist YAML²⁰.

YAML ist dabei eine Obermenge zu JSON, was bedeutet, dass jedes gültige JSON-Dokument gleichzeitig auch ein gültiges YAML-Dokument ist.[\[31\]](#)

Dabei versucht YAML eine bessere Lesbarkeit und ein vollständigeres Informationsmodell zu bieten, indem zum Beispiel durch Erweiterungen auch das Laden von Objekten möglich ist, wobei Objekte sogar bei Bedarf automatisch nachgeladen werden können.

Dies vereinfacht auf der einen Seite zwar die Speicherung von Daten und erlaubt einen dynamischen Inhalt von Einträgen, auf der anderen Seite ergeben sich allerdings hierdurch natürlich auch schwerwiegende Sicherheitslücken.

Als Beispiel soll das Modul `pyyaml` von Python 3 genutzt werden.

PyYAML bietet hierbei die Funktion `yaml.load` an, durch welche diese Erweiterung automatisch aktiviert ist.

```
/tmp >> cat ./evil.yaml                                     11:10
'test': !!python/object/apply:os.system ['uname -a']
/tmp >> cat ./vuln.py                                       11:10
#!/usr/bin/env python3
import yaml

with open('evil.yaml', 'r') as stream:
    yaml.load(stream)
/tmp >> ./vuln.py                                           11:10
Linux kali 4.18.0-kali2-amd64 #1 SMP Debian 4.18.10-2kali1 (2018-10-09) x86_64 GNU/Linux
/tmp >> □                                                  11:10
```

Abbildung 2.15: PyYAMLs `yaml.load` ist verwundbar für Code Execution

Wie in Abbildung 2.15 zu sehen, lässt sich hierdurch von einem Angreifer, der Kontrolle über die YAML-Dateien hat, beliebiger Code ausführen.

PyYAML bietet allerdings auch die Möglichkeit, mittels `yaml.safe_load` nur simple Objekte wie Integer oder Listen laden zu lassen und weitere Objekte nur dann zuzulassen, wenn sie zuvor explizit im Code als `yaml.SafeLoader` gekennzeichnet wurden.[\[32\]](#)

²⁰YAML Ain't Markup Language

Entsprechend müsste beim Laden von YAML-Dateien vom Framework darauf geachtet werden, dass nur vertrauenswürdige Objekte als Daten interpretiert werden, um diese Schwachstelle zu beheben.

Als Beispiele für Regelsätze in YAML werden wieder Regeln für die Erkennung von Benutzereingaben und für die Erkennung von verwundbaren Funktionen und deren Absicherungen gezeigt.

Listing 2.23: Definition von Benutzereingaben für Python 3

```
1 ---
2 urlparse:
3     Funktionen:
4     - Funktion: query
5       Parameter:
6       Parameterwert:
7       Kommentar: GET Request bei Nutzung von urlparse
```

Listing 2.24: Definition von verwundbaren Funktionen und Absicherungen für Python 3

```
1 ---
2 Template:
3     Funktionen:
4     - Funktionsname: substitute
5     Absicherungen:
6       html:
7         Funktionsname: "escape($VARIABLE, $QUOTE)"
8         Parameter: "$QUOTE"
9         Parameterwert: !!python/object/apply:re.compile ["(
10           quote=)?True"]
11     Kommentar: Gibt parametrisierte Texte aus, Gefahr von XSS
```

Wie man sehen kann, sind die YAML-Dateien in der Tat kürzer und aussagekräftiger, als die JSON-Dateien.

Die erhöhte Lesbarkeit durch das Python-Objekt lässt sich allerdings bestreiten und erhöht die Kopplung zwischen Daten und Code, was eher hinderlich ist, da es zum einen Benutzer abschrecken könnte und zum anderen zu Verwirrung führen könnte, wenn Benutzer versuchen, eigene Funktionen zu definieren, die allerdings nicht korrekt funktionieren, wenn sie nicht als `yaml.SafeLoader` definiert wurden.

Es scheint daher sinnvoll, wenn YAML eingesetzt werden sollte, die bereits in den CSV- und JSON-Dateien gezeigte Syntax für reguläre Ausdrücke zu verwenden.

Auf der anderen Seite ist durch die Möglichkeit, Aliase zu definieren, die Gefahr gegeben, dass Angriffe wie die aus XML²¹ bekannte »Billion Laughs Attack« ermöglicht werden.

²¹Extensible Markup Language

So könnte das folgende YAML-Dokument zum Absturz des Parsers führen, da hiermit insgesamt etwa drei Gigabyte Arbeitsspeicher benötigt werden, um sämtliche Aliase aufzulösen.[\[33\]](#)

Listing 2.25: Billion Laughs Attacke gegen YAML

```
1 a: &a ["lol","lol","lol","lol","lol","lol","lol","lol","lol","lol"]
2 b: &b [*a,*a,*a,*a,*a,*a,*a,*a,*a,*a]
3 c: &c [*b,*b,*b,*b,*b,*b,*b,*b,*b,*b]
4 d: &d [*c,*c,*c,*c,*c,*c,*c,*c,*c,*c]
5 e: &e [*d,*d,*d,*d,*d,*d,*d,*d,*d,*d]
6 f: &f [*e,*e,*e,*e,*e,*e,*e,*e,*e,*e]
7 g: &g [*f,*f,*f,*f,*f,*f,*f,*f,*f,*f]
8 h: &h [*g,*g,*g,*g,*g,*g,*g,*g,*g,*g]
9 i: &i [*h,*h,*h,*h,*h,*h,*h,*h,*h,*h]
```

Der Angriff funktioniert hierbei so, dass Aliase definiert werden, welche sich anschließend gegenseitig referenzieren.

Zuerst wird eine Liste **a** erstellt, welche den Alias **&a** erhält und aus zehn Einträgen »lol« besteht. Diese Liste wird wiederum zehnmal in Liste **b** referenziert, die zehnmal in Liste **c** referenziert wird usw., sodass schlussendlich 10^9 Einträge entstehen, die insgesamt etwa drei Gigabyte Arbeitsspeicher benötigen.

Auf der anderen Seite werden diese Aliase in PyYAML nur dann aufgelöst, wenn der komplette Baum durchlaufen wird, sodass die Gefahr als eher gering einzuschätzen ist.

Trotzdem könnte es sinnvoll sein, Aliase sicherheitshalber zu deaktivieren, zum Beispiel durch Nutzung eines angepassten Loaders, wie in [\[34\]](#) zu sehen.

2.12.3.4 XML

Eine der am weitesten verbreiteten Serialisierungssprachen ist vermutlich XML. Im Gegensatz zu YAML versucht XML dabei, keine reine Serialisierungssprache zu sein, sondern stellt eine komplette Auszeichnungssprache dar.

Hierdurch ist XML noch einmal weit mächtiger, als die bisher vorgestellten Formate, wobei allerdings fraglich ist, ob die zusätzlichen Möglichkeiten von XML im Kontext dieses Frameworks erforderlich sind.

Auch das Parsen von XML hat wie schon das Parsen von YAML seine Tücken, da es unter anderem möglich ist, bestehende und sogar externe Entitäten zu referenzieren.

Zwei Beispiele sollen dabei diese Schwachstellen verdeutlichen.

Listing 2.26: Billion Laughs-Attacke

```

1 <?xml version="1.0"?>
2 <!DOCTYPE lolz [
3     <!ENTITY lol "lol">
4     <!ELEMENT lolz (#PCDATA)>
5     <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;
        ;&lol;">
6     <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&
        lol1;&lol1;&lol1;">
7     <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&
        lol2;&lol2;&lol2;">
8     <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&
        lol3;&lol3;&lol3;">
9     <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&
        lol4;&lol4;&lol4;">
10    <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&
        lol5;&lol5;&lol5;">
11    <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&
        lol6;&lol6;&lol6;">
12    <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&
        lol7;&lol7;&lol7;">
13    <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&
        lol8;&lol8;&lol8;">
14 ]>
15 <lolz>&lol9;</lolz>

```

Der in Listing 2.26 gezeigte Angriff ist bekannt als »Billion Laughs Attack« und stellt eine Denial of Service-Attacke gegen den Parser dar.

Die Funktionsweise der Attacke ist dabei identisch mit der Abschnitt 2.12.3.3 auf Seite 54 beschriebenen.

Um diesen Angriff abzuwehren, ist es möglich, DTD²² und damit die Definition eigener Entitäten komplett zu unterbinden.^[35]

Eine weitere verbreitete Attacke gegen XML-Parser ist XEE²³.

Hierbei wird – wie der Name bereits andeutet – eine externe Entität referenziert, welche genutzt werden kann, um Daten auszuspähen oder ebenfalls, um den Parser lahmzulegen.

²²Document Type Description

²³XML External Entity

Listing 2.27: Diese XEE legt den Parser lahm

```

1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE foo [
3     <!ELEMENT foo ANY >
4     <!ENTITY xxe SYSTEM "file:///dev/urandom" >
5 ]><foo>&xxe;</foo>

```

Wie in Listing 2.27 zu sehen, kann durch das Einbinden von externen Entitäten eine Datei geöffnet werden, welche nie geschlossen wird, sodass der Parser endlos versucht, den Inhalt der Entität zu lesen.

Ersetzt man den Pfad `/dev/urandom` dagegen gegen `/etc/passwd`, so wird bei der Anzeige der Entität der Inhalt der `/etc/passwd`-Datei mit sämtlichen Benutzern des Systems angezeigt.^[36] Auch dieser Angriff kann durch das Deaktivieren von DTD verhindert werden.

Wenn schließlich diese Angriffe verhindert werden, könnten die bekannten Regelsätze zur Erkennung von Benutzereingaben und verwundbaren Funktionen und deren Absicherungen wie folgt aussehen.

Listing 2.28: Definition von Benutzereingaben für Python 3

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <urlparse>
3     <Funktionen>
4         <Funktion>query</Funktion>
5         <Parameter/>
6         <Parameterwert/>
7         <Kommentar>GET Request bei Nutzung von urlparse</Kommentar>
8     </Funktionen>
9 </urlparse>

```

Listing 2.29: Definition von verwundbaren Funktionen und Absicherungen für Python 3

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <Template>
3     <Funktionen>
4         <Funktionsname>substitute</Funktionsname>
5         <html>
6             <Funktionsname>escape($VARIABLE, $QUOTE)</Funktionsname>
7             <Parameter>$QUOTE</Parameter>
8             <Parameterwert>/(quote=)?True/</Parameterwert>
9         </html>
10        <Kommentar>Gibt parametrisierte Texte aus, Gefahr von XSS</
            Kommentar>
11    </Funktionen>
12 </Template>

```

Wie man sehen kann, sind die Regelsätze sehr eindeutig und gut lesbar.

Auf der anderen Seite werden allerdings wie bei JSON viele syntaktische Elemente benötigt, die in keinem direkten Bezug zum Inhalt der Regel stehen.

Zwar wird hierdurch die Interpretation für den Computer vereinfacht, das Schreiben von Regeln wird allerdings andererseits erschwert.

2.12.3.5 Vergleich der Formate und Auswahl

Wie man an den vorherigen Beispielen sehen kann, haben komplexere Formate einerseits den Vorteil, dass die Lesbarkeit für Mensch und Maschine stark verbessert wird.

Auf der anderen Seite bieten sich aber auch immer mehr Angriffsmöglichkeiten auf den Parser derartig komplexer Sprachen.

Im direkten Vergleich zeigt sich, dass subjektiv gesehen YAML die beste Lesbarkeit von allen vorgestellten Formaten bietet, allerdings auch an einigen schwerwiegenden Sicherheitslücken leidet.

CSV dagegen hat die schlechteste Lesbarkeit, ist dafür aber auch zu simpel, als dass es anfällig für Sicherheitslücken wäre.

JSON hat eine ausreichende Lesbarkeit, kann aber durch Einsatz von JSONP bereits zu ernststen Sicherheitsproblemen führen, wenn die entsprechende Erweiterung aktiviert ist.

Auch XML ist vergleichsweise ausreichend gut lesbar, allerdings sind die Regelsätze hier derart stark strukturiert, dass der Bezug zum eigentlichen Inhalt ein wenig verloren geht. Genau wie die anderen Formate – von CSV abgesehen – weist XML allerdings einige Sicherheitslücken auf.

Durch die vergleichsweise beste Lesbarkeit und die Möglichkeit, die Sicherheitslücken durch entsprechende Einstellungen bzw. Erweiterungen abzusichern, fällt die Entscheidung für das Format der Regeln daher auf YAML.

Wie allerdings ebenfalls festgestellt wurde, führt der Einsatz regulärer Ausdrücke in den Regeln zu vielen Problemen. Da keine symbolische Ausführung stattfindet, kann hierbei nicht immer erkannt werden, welcher konkrete Wert während der Laufzeit wirklich eingesetzt wird.

Entsprechend werden die Regeln so angepasst, dass die Parameterliste entweder den Wert `null` enthalten kann, welcher das YAML-Äquivalent zu Pythons `None` darstellt, oder den Wert `$TAINT`, um anzudeuten, dass an dieser Stelle eine benutzerdefinierte Variable eingespeist wird.

2.12.4 Erstellung universeller Regeln

Um die Erstellung von Regeln zu vereinfachen, wird im Folgenden untersucht, ob es möglich ist, universelle Regeln für die Erkennung von Schwachstellen in Quelltexten zu erstellen, welche unabhängig von der verwendeten Programmiersprache Sicherheitslücken erkennen.

Eine einfache Möglichkeit wäre es, ein Pseudomodul hinzuzufügen, dessen Regeln unabhängig von der erkannten Programmiersprache – und somit auch ohne jegliche Informationen über die Struktur dieser Sprache – nach Mustern sucht, die als Benutzereingaben, Absicherungen oder verwundbare Funktionen erkannt werden.

Dieser Ansatz wurde allerdings bereits in Abschnitt 2.5.2.1 auf Seite 11 als zu ungenau und aufwendig abgelehnt, wobei hier zusätzlich noch erschwerend hinzukommt, dass die Regeln sprachunabhängig sein sollten, wodurch sich das benötigte Muster weiter verkompliziert.

Da dieser Ansatz keinerlei Vorteile durch die Nutzung des entwickelten Frameworks erfahren würde, wäre es hierbei einfacher, mittels spezialisierter Suchprogramme wie `grep`²⁴ vorzugehen.

Eine andere Möglichkeit, universell einsetzbare Suchmuster für verwundbare Funktionen zu erstellen, wäre es, einen bijektiven Universalcompiler für sämtliche verwendeten Programmiersprachen zu erstellen.

Dieser müsste nicht nur eine beliebige Programmiersprache in eine Metasprache übersetzen, welche anschließend auf Sicherheitslücken untersucht und wiederum in die ursprüngliche Programmiersprache zurück abgebildet werden können müsste, sondern es müssten auch Sprachkonstrukte umgewandelt und Abstraktionen von höheren Programmiersprachen aufgelöst werden, um eine Vergleichbarkeit aller Programmiersprachen zu gewährleisten.

Eine Erstellung eines derartigen Universalcompilers ist allerdings nicht nur zum einen aus den oben genannten Gründen enorm aufwendig, sondern durch die Entfernung der Abstraktionen höherer Programmiersprachen fehleranfällig und führt zu den gleichen Schwierigkeiten, wie sie bei statischen Analysemethoden auf der Ebene von Binärdateien statt der eigentlichen Quelltexte auftreten.

Wie man also sieht, ist die Erstellung universeller Regeln leider nicht mit vertretbarem Aufwand möglich.

²⁴`global/regular expression/print`

2.13 Allgemeine Analyse der Codekomplexität

Neben der direkten Suche nach Schwachstellen im Code kann es auch sinnvoll sein, die Komplexität des Codes zu untersuchen, um zu versuchen, potenziell verwundbaren Code aufzuspüren.

Die Untersuchung der Codekomplexität ist dabei kein großer zusätzlicher Aufwand, da bei der statischen Analyse ohnehin der komplette Code untersucht wird und dessen Bestandteile analysiert werden.

Da die benutzte Metrik lediglich auf der Anzahl der Anweisungen, der Verzweigungen und der einzelnen Codeblöcke beruht, ist es ein Leichtes, diese Daten während des Scans zu erfassen und im Anschluss eine Bewertung auszugeben.

Eine hohe Komplexität deutet zwar natürlich nicht automatisch auf Sicherheitslücken hin, wie allerdings die Kryptografen Niels Ferguson und Bruce Schneier schon 1999 anmerkten: »Security's worst enemy is complexity« (auf Deutsch: »Komplexität ist der schlimmste Feind der Sicherheit«).[\[37\]](#)

Aus diesen Gründen erscheint es sinnvoll, die Codekomplexität zumindest als Resultat mit auszugeben und damit auf Codestellen hinzuweisen, welche noch einmal manuell von einem erfahrenen Programmierer auf Sicherheitslücken untersucht werden sollten.

Da die Komplexität selbst wie angemerkt keine Sicherheitslücke an sich darstellt, sondern lediglich die Wahrscheinlichkeit für das Auftreten von Lücken erhöht, wird sie wertungsfrei ausgegeben, sofern eine bestimmte Komplexität von einer Funktion überschritten wird.

Die zu diesem Zweck im Framework implementierte Metrik ist die von Thomas J. McCabe 1996 vorgestellte zyklomatische Komplexität, auch bekannt als die McCabe-Metrik.

Zwar gibt es auch noch viele weitere bekannte Metriken, um Codekomplexität zu messen, wie zum Beispiel die Halstead-Metrik von Maurice H. Halstead aus dem Jahr 1977, allerdings sind diese zum einen aufwendiger zu messen, zum anderen aber sollte der Fokus des Frameworks auch nicht darauf liegen, komplexe Codestellen zu finden, was bereits eine Vielzahl anderer Programme ermöglichen, sondern Sicherheitslücken aufzuspüren und hierzu sollte es ausreichend sein, anhand einer Metrik auf möglicherweise zu komplexe Funktionen hinzuweisen.

2.13.1 Berechnung der McCabe-Metrik

Die McCabe-Metrik wird für jede Funktion berechnet als $v(G) = e - n + 2$, wobei e die Anzahl der Kanten und n die Anzahl der Knoten im CFG darstellen.

Zur Verdeutlichung soll ein Beispiel dienen.

Listing 2.30: Der einfache euklidische Algorithmus in Python

```
1 def euclid(n, m):
2     if n > m:
3         r = m
4         m = n
5         n = r
6     r = m % n
7     while r != 0:
8         m = n
9         n = r
10        r = m % n
11    return n
```

Der vollständige CFG zu dem Programm in Listing 2.30 sieht wie in Abbildung 2.16 auf der nächsten Seite aus.

In diesem Fall wurden die einzelnen Anweisungen nicht in Blöcke zusammengefasst, da die McCabe-Metrik jede einzelne Anweisung als einen Knoten zählt und somit eine Zusammenfassung das Resultat verzerren würde.

Dem Beispiel folgend kann man sehen, dass in dem CFG insgesamt $n = 12$ Knoten und $e = 13$ Kanten existieren, die sich ergebende zyklomatische Zahl ist also $v(G) = 13 - 12 + 2 = 3$.

Hierbei ist wichtig zu beachten, dass sich die Anzahl der Knoten je nach Messung unterscheiden kann, da im ursprünglichen Paper vorgesehen ist, einen Pseudoknoten zwischen den Zeilen 5 und 6 einzufügen, um anzudeuten, dass es sich bei der Verzweigung um zwei verschiedene Zweige handelt, selbst wenn einer der beiden komplett leer ist, wie in diesem Fall, wo der `else`-Zweig nicht explizit aufgeschrieben wurde.[\[38, S. 10\]](#)

Die meisten gängigen Testprogramme für die McCabe-Metrik ignorieren diesen leeren Zweig,[\[39\]](#) da es allerdings für die statische Analyse erforderlich ist, auch leere Zweige als alternative Pfade durch eine Methode zu erkennen, wird für das Framework die im ursprünglichen Paper vorgeschlagene Variante genutzt.

Der Unterschied beider Varianten ist allerdings gering genug, um vernachlässigbar zu sein, und da die Erkennung leerer Zweige durch die Grammatik erfolgt, ist es durchaus möglich, dass unterschiedliche Grammatiken zu unterschiedlichen Ergebnissen kommen.

Die zyklomatische Zahl gibt für diesen Fall an, dass im Maximum drei Testfälle benötigt werden, um eine komplette Zweigüberdeckung zu erreichen, was als geringe Komplexität interpretiert werden kann.

Als Faustregel gibt das Originalpaper dabei an, dass die zyklomatische Zahl einen Wert von 10, bei erfahrenen Programmierern und häufigen Codereviews maximal 15, nicht überschreiten sollte, da ansonsten die Komplexität zu hoch wäre, um noch mit vertretbarem Aufwand Testfälle zu entwerfen.[\[38, S. 25\]](#)

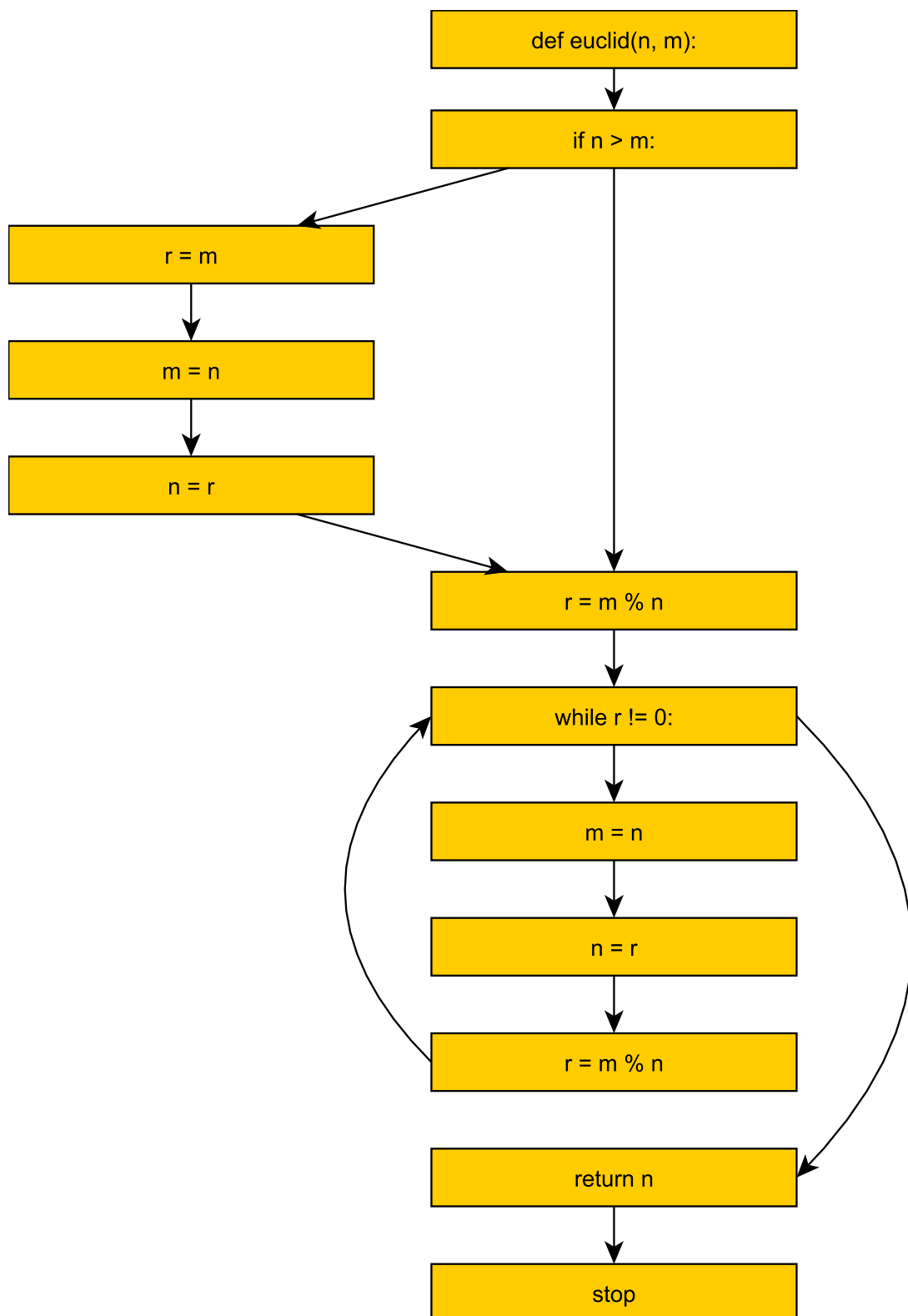


Abbildung 2.16: Der vollständige CFG des einfachen euklidischen Algorithmus

2.13.1.1 Grundlagen der McCabe-Metrik

Die in der McCabe-Metrik genutzte zyklomatische Zahl basiert auf der ersten Bettizahl.

Bei der i -ten Bettizahl eines topologischen Raums X handelt es sich um

$$b_i(X) = \dim_{\mathbb{Q}} H_i(X, \mathbb{Q})$$

für $i = 0, 1, 2, \dots$, wobei $H_i(X, \mathbb{Q})$ die i -te singuläre Homologiegruppe mit Koeffizienten in den rationalen Zahlen bezeichnet.[\[40\]](#)

Anschaulich geben die Bettizahlen an, wie viele k -dimensionale, nicht zusammenhängende Flächen im topologischen Raum X existieren, sodass die erste Bettizahl, b_0 die Anzahl der Zusammenhangskomponenten von X angibt.[\[40\]](#)

Die erste Bettizahl eines Graphen G mit e Kanten, n Knoten und k Zusammenhangskomponenten ergibt dabei $b_1(G) = m - n + k$, da das Hinzufügen einer neuen Kante entweder die Anzahl an Zusammenhangskomponenten reduziert oder die Anzahl an Selbstreferenzen erhöht, sodass das Verhältnis immer konstant bleibt.[\[41\]](#)

Auf dieser Grundlage kann die zyklomatische Zahl berechnet werden, wobei als Vereinfachung bei der Berechnung der gerichtete CFG als ungerichteter Graph interpretiert wird, da die gleiche Berechnung auch die Anzahl der unabhängigen Pfade durch einen stark zusammenhängenden gerichteten Graphen angibt.[\[38, S. 20\]](#)

Da es innerhalb einer Funktion grundsätzlich nur eine Zusammenhangskomponente geben kann, da die Zeilen des Quelltextes jeweils aufeinander folgen, kann dabei die Anzahl der Zusammenhangskomponenten $k = 1$ gesetzt werden.

Allerdings stellen Programmfunktionen standardmäßig keine starken Zusammenhangskomponenten dar, sodass die Anzahl der Kanten um eine virtuelle Kante vom End- zum Anfangsknoten ergänzt wird, wie in Abbildung 2.17 auf der nächsten Seite mit einer gestrichelten Kante angedeutet.

Insgesamt ergibt sich also für die McCabe-Metrik hiermit die Berechnung

$$v(G) = e + 1 - n + \underbrace{k}_{=1} = e - n + 2.$$

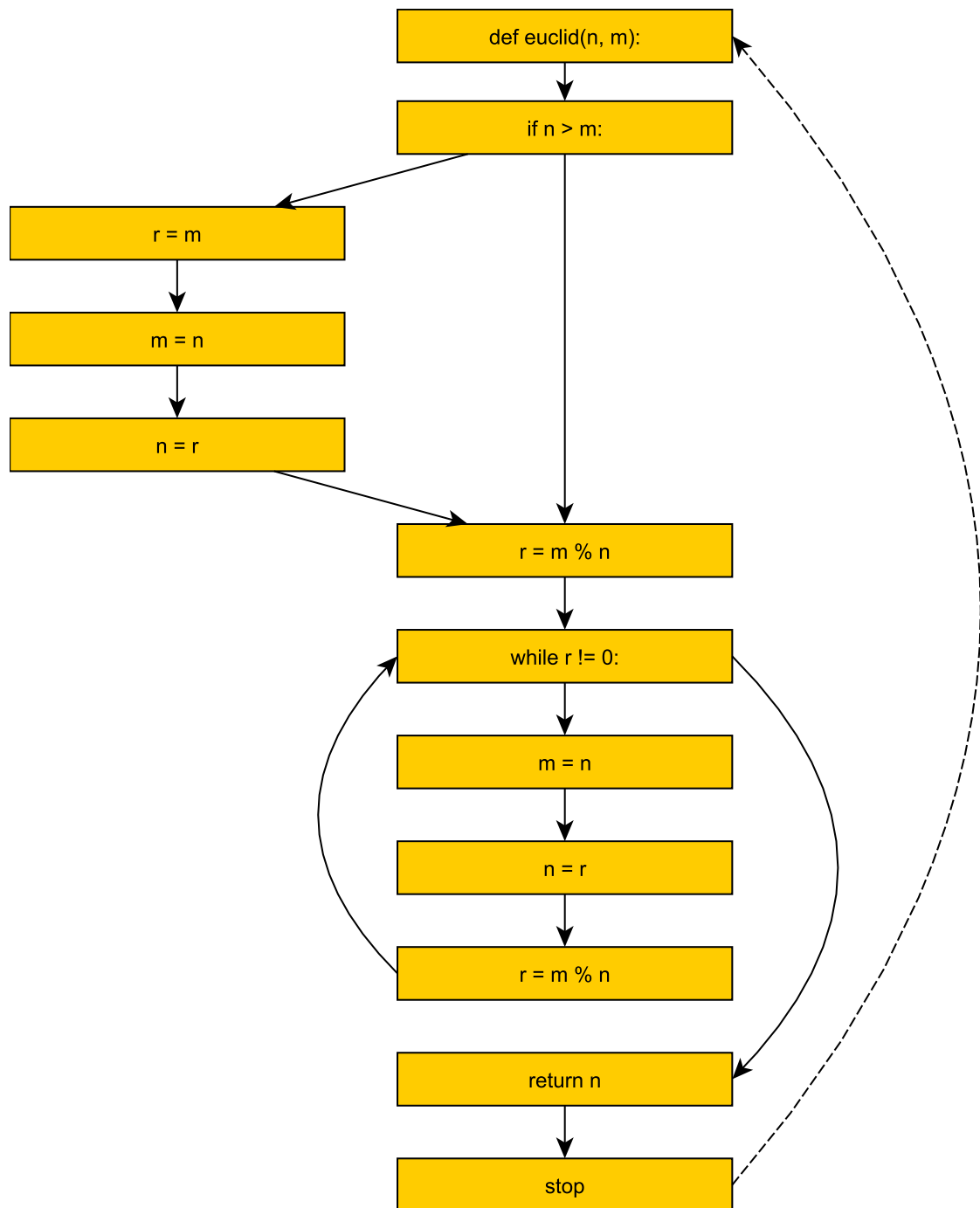


Abbildung 2.17: Der CFG wird um eine virtuelle Kante vom End- zum Startknoten erweitert

2.13.1.2 Aussagekraft

Zwar wird die McCabe-Metrik im Originalpaper als Methode angepriesen, Softwarekomplexität zu messen,[38, S. 5] hierdurch werden allerdings Komplexität und die Anzahl der unabhängigen Wege durch ein Programm miteinander gleichgesetzt, was ein Trugschluss ist, wie im Folgenden aufgezeigt wird.

Trotzdem wird die McCabe-Metrik für dieses Framework eingesetzt, da eine direkte Messung von Softwarekomplexität nicht möglich ist und sich die McCabe-Metrik trotz ihrer Schwächen als guter Indikator für Bugs in Software etabliert hat.[38, S. 123]

Eine dieser Schwächen, welche unabhängig von der Gleichsetzung von Komplexität und möglichen Programmpfaden ist, ist die unterschiedliche Messung der zyklomatischen Komplexität in verschiedenen Programmen.

Als Beispiel hierfür kann der Code in Listing 2.30 auf Seite 62 gesehen werden, bei dem – je nach Messung – leere `else`-Zweige entweder als eigener Knoten gezählt werden oder nicht.

Da das Originalpaper von Watson und McCabe aus dem Jahr 1996 stammt, werden weiterhin einige neuere Programmierkonzepte, wie Lambda-Funktionen, nicht aufgeführt und deren Bewertung ist damit der jeweiligen Implementation überlassen.[39]

Eine grundlegendere Schwachstelle der Metrik ist jedoch die Gleichsetzung der Pfade durch das Programm mit der Komplexität des Programms. Hierdurch werden zum Beispiel folgende Codes als gleichermaßen »kompliziert« bewertet, auch wenn die meisten Programmierer vermutlich anderer Ansicht sind.[39]

Listing 2.31: Eine einfache Gewichtsfunktion

```
1 String getWeight(int i) {  
2     if (i <= 0) {  
3         return "no weight";  
4     }  
5     if (i < 10) {  
6         return "light";  
7     }  
8     if (i < 20) {  
9         return "medium";  
10    }  
11    if (i < 30) {  
12        return "heavy";  
13    }  
14    return "very heavy";  
15 }
```

Listing 2.32: Die Summe aller Primzahlen

```
1 int sumOfNonPrimes(int limit) {  
2     int sum = 0;  
3     OUTER: for (int i = 0; i < limit; ++i) {  
4         if (i <= 2) {  
5             continue;  
6         }  
7         for (int j = 2; j < i; ++j) {  
8             if (i % j == 0) {  
9                 continue OUTER;  
10            }  
11        }  
12        sum += i;  
13    }  
14    return sum;  
15 }
```

Beide Codestücke haben eine zyklomatische Zahl von 5, allerdings ist die Komplexität der beiden stark unterschiedlich.

Wie oben bereits angemerkt, existiert leider keine Metrik, welche die reale Komplexität von Code messen kann.

Aus diesem Grund wird der Berechnung der McCabe-Metrik im Kontext des Frameworks auch kein Risikowert zugewiesen, sondern es wird lediglich ein Hinweis auf die betroffenen Codestellen gegeben, sodass erfahrene Programmierer selbst prüfen müssen, ob die Komplexität eines Programmabschnitts zu groß ist, um Sicherheitslücken auszuschließen.

3. Praktische Umsetzung

Im Folgenden werden wichtige Schritte der praktischen Umsetzung bei der Implementierung des Frameworks erklärt.

Um die Umsetzung strukturiert zu gestalten, werden hierbei zuerst Planungsschritte aufgezeigt und im Anschluss daran die einzelnen Bestandteile nacheinander entwickelt, wobei jeweils zugehörige Unittests erstellt werden, um die korrekte Funktionalität sicherzustellen.

3.1 UML-Klassendiagramm

Das Klassendiagramm aus Abbildung 3.1 auf der nächsten Seite gibt die grundlegenden Klassen und Funktionen des Frameworks an.

Die Main-Klasse wurde zur besseren Sichtbarkeit farbig hinterlegt.

Nachdem in der Main-Klasse die Kommandozeilenoptionen eingelesen und verarbeitet wurden (vergleiche Abschnitt 3.5 auf Seite 72), werden die einzelnen Dateien eingelesen (vergleiche Abschnitt 3.6 auf Seite 73) und deren Dateityp bestimmt (vergleiche Abschnitt 3.7 auf Seite 74).

Von dort aus werden die einzelnen Dateien durch den Parser für die Analyse vorbereitet (vergleiche Abschnitt 3.8 auf Seite 78), indem sie vom Parser in einen Parsetree, einen AST und schließlich einen CFG umgewandelt werden.

Der resultierende CFG kann dann analysiert werden, indem zuerst die passenden Regelsätze geladen werden (vergleiche Abschnitt 3.9 auf Seite 87) und anhand derer die Quellen, Senken und Absicherungen ermittelt werden (vergleiche Abschnitt 3.10 auf Seite 89).

Diese werden abgespeichert und später zur Generierung des Reports genutzt (vergleiche Abschnitt 3.12 auf Seite 95).

Der Report wird weiterhin um eine Analyse der Codekomplexität ergänzt (vergleiche Abschnitt 3.11 auf Seite 95), wobei die Funktionen ausgegeben werden, deren zyklomatische Zahl einen bestimmten Grenzwert überschreiten.

3.2 Entwicklung in einer isolierten Umgebung

Da die Entwicklung des Frameworks in Python durchgeführt wird und verschiedene Abhängigkeiten genutzt werden, scheint es sinnvoll, die Entwicklung in einer isolierten Umgebung durchzuführen, um sicherzustellen, dass immer die passenden Bibliotheken genutzt werden und es zu keinen Problemen führt, wenn eine der Abhängigkeiten verändert wurde.

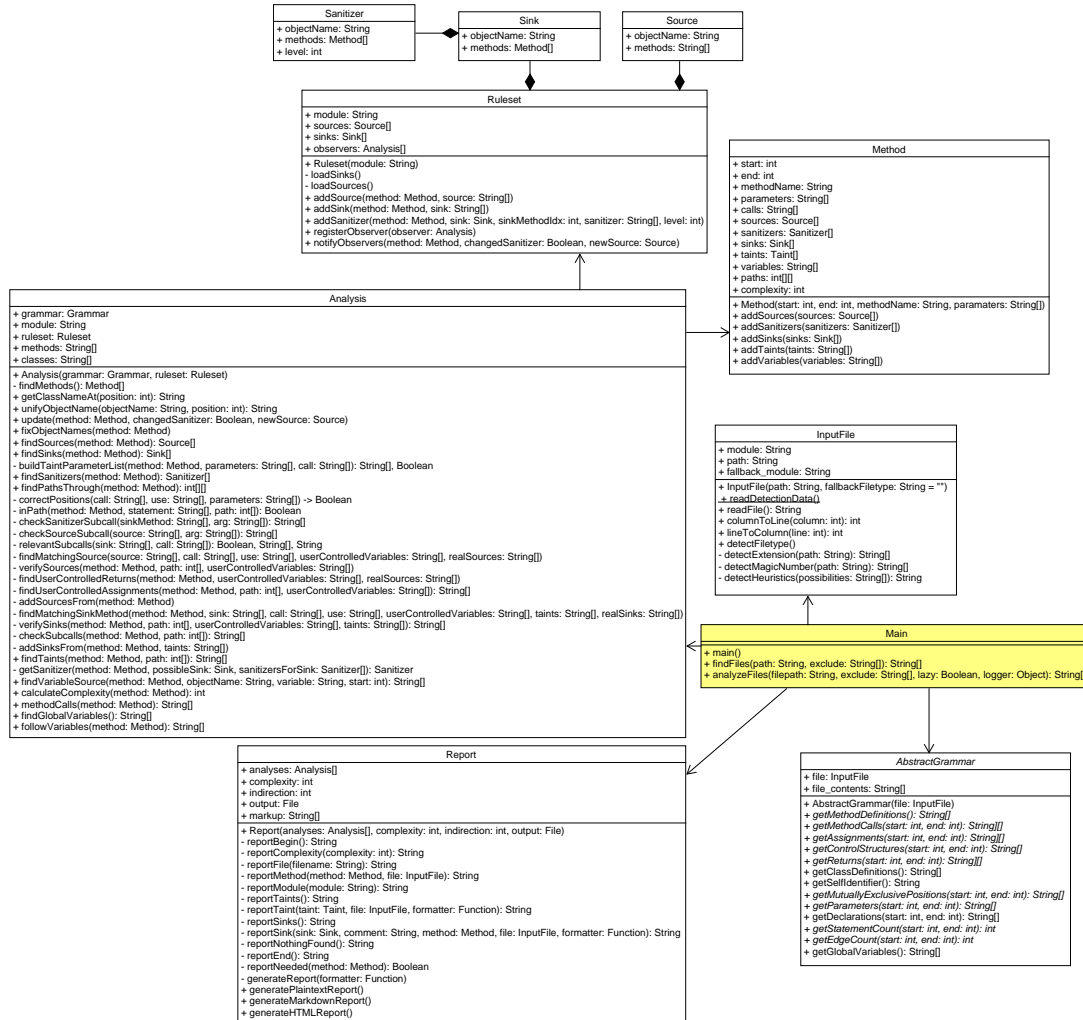


Abbildung 3.1: UML-Klassendiagramm des Frameworks

In neueren Versionen von Python (ab 3.6) ist hierfür das Modul **venv** in der Standardbibliothek enthalten, welches die Erstellung einer isolierten Umgebung erlaubt.

3.2.1 venv

venv ist eine in den Standard aufgenommene Alternative zum verbreiteten **Virtualenv** von Ian Bicking.^[42] Es wird seit Python 3.3 offiziell als neuer Standard empfohlen.^[43]

3.2.2 Pipenv

Obwohl die Nutzung einer virtuellen Umgebung mittels **venv** bereits große Vorteile gegenüber einer Installation ohne isolierte Umgebung bietet, ist es trotzdem sinnvoll, zusätzlich zu **venv** noch **Pipenv** einzusetzen.

Dies liegt daran, dass **Pipenv** den Paketmanager **PIP**²⁵ und die isolierte Umgebung von **venv** miteinander kombiniert und zusätzlich Hashes für sämtliche genutzten Pakete anlegt, durch welche zum einen die Installation deterministisch wird und zum anderen verhindert wird, dass Pakete manipuliert werden, sodass die Sicherheit erhöht wird, was insbesondere bei sicherheitskritischen Anwendungen wie einem statischen Codescanner wichtig ist.^[44]

Zur Installation werden Module daher hinzugefügt mit dem Befehl

Listing 3.1: Hinzufügen neuer Module in Pipenv

```
1 pipenv install PACKAGE
```

bzw. bei Paketen, welche nur für die Entwicklung notwendig sind und nicht für den späteren Produktiveinsatz

Listing 3.2: Hinzufügen neuer Module in Pipenv

```
1 pipenv install PACKAGE --dev
```

Nach Abschluss der Entwicklung werden sämtliche Abhängigkeiten und Dateihashes abgespeichert über

Listing 3.3: Erstellung der Pipfile.lock

```
1 pipenv lock
```

²⁵PIP Installs Packages

Skripte können ausgeführt werden mittels

Listing 3.4: Ausführen von Skripten in Pipenv

```
1 pipenv run COMMAND [ARGS] ...
```

Für die spätere Nutzung müssen Endanwender zuerst ebenfalls Pipenv installieren und dann mittels

Listing 3.5: Installation der Pakete für Endanwender

```
1 pipenv install --ignore-pipfile
```

die für die Laufzeit benötigten Pakete installieren.

Um den Installationsprozess übersichtlich darzustellen, wurden sämtliche notwendigen Schritte in der Datei `INSTALL.md` aufgelistet.

3.2.3 Erstellung eines Installationsskripts

Um die Installation des Frameworks zu vereinfachen, wurde ein Installationsskript angelegt.

Durch dieses Skript wird zuerst `Pipenv` auf dem System installiert und anschließend genutzt, um die Projektabhängigkeiten zu installieren.

Hierdurch sollte es möglich sein, auf beliebigen Systemen mit installiertem Python 3 das Framework deterministisch zu installieren und auszuführen.

3.3 Codestil

Der Codestil orientiert sich an PEP²⁶ 8, dem offiziellen Styleguide von Python[45], sowie am Numpy Docstring Guide für Erklärungstexte in Modulen, Klassen und Funktionen.[46]

Obwohl Python eine dynamische Typisierung hat, werden durchgängig »Type Hints« genutzt, welche andeuten, um was für einen Datentyp es sich handelt.[47]

Da Type Hints allerdings nicht durch die Programmiersprache erzwungen werden, sind diese Hinweise vor allem für die bessere Lesbarkeit gedacht.

Zusätzlich findet eine Codeprüfung mittels Flake8 ([48]) und Pylint ([49]) statt, um einen sauberen und möglichst gut lesbaren Code sicherzustellen.

²⁶Python Enhancement Proposal

3.4 Unittests

Um die Funktionalität des Frameworks sicherzustellen und Regressionsfehler zu vermeiden, werden für alle wichtigen Funktionen des Frameworks Unittests erstellt, mithilfe derer Testeingaben mit zu erwartenden Ergebnissen verglichen werden.

Hierdurch ist es möglich, Fehler in bestimmten Funktionen frühzeitig zu entdecken und sie zu beheben, damit es nicht zu Störungen im produktiven Betrieb kommt.

Wie auch bei der statischen Analyse ist es allerdings unmöglich, mit Unittests den korrekten Ablauf eines Programms zu garantieren, sodass sie lediglich als zusätzliche Maßnahme zur Qualitätssicherung verwendet werden, statt als Garantie einer korrekten und fehlerfreien Funktion.

3.5 Einlesen der Kommandozeilenoptionen

Über die Kommandozeilenoptionen kann der Benutzer angeben, welche Funktionen des Frameworks er aufrufen möchte.

Hierbei ist vor allem wichtig, dass der Nutzer einen Pfad angeben kann, in welchem die Dateien zu finden sind, welche durch das Framework untersucht werden sollen.

Weitere Parameter sind optional und dienen der Konfiguration des Suchlaufs.

So kann über den Parameter `-i` oder `--indirection` angegeben werden, ab wie vielen Stufen indirekten Aufrufs eine Absicherung nicht mehr als solche gewertet wird, da sie zu weit von der Senke entfernt ist.

Der Parameter `-c` oder `--complexity` gibt an, wie komplex eine Funktion sein darf, bevor sie in den Report aufgenommen wird. Hierbei wird als Kennzahl die zyklomatische Zahl verwendet, mit den in Abschnitt 2.13.1.2 auf Seite 66 beschriebenen Einschränkungen.

Der Parameter `-f` oder `--filetype` kann genutzt werden, um die Erkennung eines Dateityps zu erzwingen, wenn keine automatische eindeutige Erkennung anhand der Dateiendung oder der magischen Zahl möglich ist.

Der Parameter `-o` oder `--output` gibt eine Ausgabedatei für den Report an. Wenn der Parameter weggelassen wird, wird der Report im Textformat auf der Standardausgabe ausgegeben. Wird stattdessen eine Datei mit der Endung `.markdown` oder `.md` angegeben, wird der Report im Markdown-Format ausgegeben oder bei einer Datei mit Endung `.htm` oder `.html` im HTML-Format und bei `.txt` im Textformat. Abweichende Dateiendungen produzieren eine Warnung und eine Ausgabe im Textformat.

3.5.1 Erstellung einer Hilfe

Die eingebaute Programmhilfe, welche standardmäßig mit dem Parameter `-h` oder `--help` angezeigt wird, ist im POSIX²⁷-Format erstellt.[\[50\]](#)

Hierdurch ist die Nutzung zum einen für erfahrenere Nutzer intuitiv möglich, da dieser Standard eine hohe Verbreitung hat, zum anderen ist es hiermit auch einfach möglich, die Optionen mittels `docopt` zu parsen, welches – anders als die sonst üblichen Standards `optparse` und `argparse` – keine Hilfetexte basierend auf dem Code erstellt, sondern anhand des Hilfetextes selbstständig die Parameter erkennt.[\[51\]](#)

3.5.2 Übergabe von Kommandozeilenoptionen mittels `docopt`

Durch die Nutzung von `docopt` ist es möglich, Kommandozeilenoptionen direkt in der Hilfe des Programms zu definieren.

Allerdings ist es hiermit nicht direkt möglich, auch die Korrektheit der Eingaben zu prüfen.

Zu diesem Zweck wird `docopt` im Framework mit `schema` kombiniert.

`schema` ist ein Modul zur Validierung von Python-Datenstrukturen, welches versucht, Daten in ein bestimmtes Format zu konvertieren. Wenn dies fehlschlägt, wird eine Fehlermeldung ausgegeben.[\[52\]](#)

3.6 Einlesen der zu untersuchenden Quelltexte

Um die zu untersuchenden Quelltexte einzulesen, wird die Python-Funktion `walk` eingesetzt, welche rekursiv Verzeichnisse durchläuft und dabei jeweils das aktuelle Verzeichnis, die enthaltenen Unterverzeichnisse und die enthaltenen Dateien zurückliefert.

Da die Unterverzeichnisse selbst für dieses Framework nicht relevant sind, wird der zugehörige Wert ignoriert, was konventionsgemäß in Python durch einen Unterstrich ausgedrückt wird.

Um hierbei nicht sämtliche Dateinamen gleichzeitig im Speicher halten zu müssen, wurde die Funktion zum Einlesen der Quelltexte als Generator geschrieben, was bedeutet, dass jeweils nur der aktuelle Wert zurückgegeben und die Funktion bei einem erneuten Aufruf an der Stelle fortgesetzt wird, wo die Ausführung beim letzten Mal durch das Schlüsselwort `yield` unterbrochen wurde.

Als Ergebnisse werden der (relative) Dateipfad und der Inhalt der Datei als Liste von Strings zurückgegeben.

²⁷Portable Operating System Interface for Unix

3.7 Auswahl der Module für verschiedene Programmiersprachen

Um für jeden Quelltext die richtigen Regelsätze zu bestimmen, muss das Framework die Programmiersprache des Quelltextes korrekt erkennen und die zugehörigen Regelsätze laden.

Da dieses Problem wie in Abschnitt 2.11.3 auf Seite 38 beschrieben nicht ohne weiteres eindeutig lösbar ist, werden verschiedene Ansätze miteinander kombiniert.

Zur Erkennung der Dateiformate wird im Unterverzeichnis »modules« des Projekts eine Textdatei mit dem Namen »detection.txt« benötigt.

In dieser Datei steht, für welche Dateiendung und welche Erkennung durch die magische Zahl ein Modul ausgewählt werden sollte.

Das Format ist hierbei so aufgebaut, dass zuerst das Modul, anschließend ein Doppelpunkt, dann die Dateiendungen mit vorangestelltem Punkt, getrennt durch Kommas und nach einem Semikolon der Anfang der Ausgabe des Magic-Befehls steht.

Zur Erläuterung soll das Beispiel aus Listing 3.6 dienen.

Listing 3.6: Beispieleinträge in der detection.txt

```
1 c: .c, .h; C source
2 cpp: .cc, .cpp, .hpp; C source
```

Wie man hieran sehen kann, kann es durchaus passieren, dass die Erkennung des Magic-Befehls uneindeutig ist, weshalb zuerst die magische Zahl ermittelt wird und anschließend, wenn diese nicht eindeutig ist, die Ergebnisse durch die Dateiendung eingeschränkt werden.

Sollte keine der beiden Methoden zu einem Ergebnis führen, so wird eine Heuristik basierend auf der Mehrheit der erkannten Dateiformate im aktuellen Verzeichnis eingesetzt, sofern der Anwender keine Rückfalllösung angegeben hat.

3.7.1 Erkennung des Dateiformats anhand der Endung

Um das Dateiformat anhand der Endung zu erkennen, wird die Funktion `os.path.splitext` genutzt.

Diese Funktion trennt den Dateinamen von der Erweiterung und gibt beides in einem Tupel zurück.

Als Besonderheit gibt die Funktion bei Dateien, deren Namen mit einem Punkt beginnen und die keine eigene Dateierweiterung haben, den Dateinamen nicht als Erweiterung,

sondern als Dateiname zurück, sodass zum Beispiel `.bashrc` nicht als namenlose Datei mit der Endung `.bashrc` erkannt wird, sondern als Datei mit dem Namen `.bashrc` ohne Endung.

Sämtliche passenden Module zu der gefundenen Dateierweiterung werden anschließend in einer Liste zurückgegeben.

3.7.2 Erkennung des Dateiformats mittels der magischen Zahl

Für die Erkennung mittels der magischen Zahl wird das Paket `python-magic` verwendet, welches intern die Funktion des Unix-Standardbefehls `file` nachstellt.

Da hierfür unter Windows zusätzliche Dateien benötigt werden, wurde ein Installationskript für Windows erstellt, welches abhängig von der Architektur des Systems (32 Bit oder 64 Bit) die entsprechenden Dateien aus einem Unterordner in das aktuelle Verzeichnis kopiert, damit sie von `python-magic` gefunden werden können.

Es ist hierbei wichtig zu erwähnen, dass durch die für die 32 Bit-Version von Windows benötigte DLL²⁸ die Lizenz des kompletten Projekts von der sehr freien MIT²⁹-Lizenz auf die weit striktere GPL³⁰ wechselt, da ein wesentlicher Bestandteil nun unter GPL steht. Alle weiteren verwendeten Bestandteile sind unter der MIT-Lizenz lizenziert, sodass diese auch für das Projekt verwendet werden könnte.

Für die spätere Auslieferung könnte es daher sinnvoll sein, die Dateien für die 32 Bit-Version von Windows nicht mitzuliefern und lediglich einen Hinweis auf die Quelle zu geben, damit die Anwender sie sich selbst beschaffen können.

Da die Verbreitung von 32 Bit-Windows-Systemen ohnehin immer mehr abnimmt, sollte dies zu keinen großen Einschränkungen führen, erlaubt es allerdings, eine freiere Lizenz für das Gesamtprojekt zu wählen.

Ebenso wie bei der Erkennung des Dateityps anhand der Erweiterung wird auch hier eine Liste von möglichen Modulen zurückgegeben.

3.7.3 Erkennung des Dateiformats anhand einer Heuristik

Um anhand der Heuristik ein Dateiformat zu erkennen, werden die anderen Dateien im gleichen Verzeichnis mittels der vorherigen Methoden ermittelt und anschließend das am häufigsten erkannte Dateiformat verwendet.

²⁸Dynamic Link Library

²⁹Massachusetts Institute of Technology

³⁰GNU General Public License

Hierbei werden keine Unterordner durchsucht, da es vielfach üblich ist, Projektdateien, die in verschiedenen Programmiersprachen geschrieben sind, in unterschiedlichen Ordnern unterzubringen.

Weiterhin kann eine Liste möglicher Dateiformate für die Heuristik übergeben werden, wenn die Erkennung zwar möglich, aber nicht eindeutig war.

In diesem Fall wird die Liste der möglichen Module auf diese Module eingeschränkt, um eine Falscherkennung weiter zu minimieren.

Sollte keine weitere Datei im gleichen Verzeichnis erkannt werden, so wird eine Warnung ausgegeben und die Datei bekommt kein Modul zugewiesen.

3.7.4 Erzwungene Erkennung unbekannter Dateiformate

Man kann der Klasse zur Behandlung von Eingabedateien einen String mitgeben, welcher ein Modul festlegt, welches genutzt werden soll, wenn die Erkennung anhand der Dateiondung und der magischen Zahl fehlschlägt.

Wenn diese Funktion aktiviert wird, wird die Erkennung anhand der Heuristik automatisch deaktiviert.

Sollte die Erkennung mittels der Dateiondung oder der magischen Zahl ein eindeutiges Ergebnis liefern, so wird dieses bevorzugt.

Wenn die Erkennung mittels Dateiondung oder magischer Zahl kein eindeutiges Ergebnis zurückliefert, sondern eine Vielzahl von möglichen Erkennungen existiert, das erzwungene Modul aber nicht in dieser Liste enthalten ist, wird eine Warnmeldung ausgegeben. Die Erkennung wird allerdings trotzdem mit dem erzwungenen Modul durchgeführt.

3.7.5 Darstellung als Programmablaufplan

Der Algorithmus zur Erkennung des Dateiformats besteht aus mehreren Teilen, die miteinander zusammenarbeiten, um in möglichst vielen Fällen ein korrektes Ergebnis zu liefern. Ein vereinfachter Programmablaufplan des gesamten Algorithmus wie in Abbildung 3.2 auf der nächsten Seite kann die Zusammenhänge besser veranschaulichen.

Die Details sind den jeweiligen Unterabschnitten zu entnehmen.

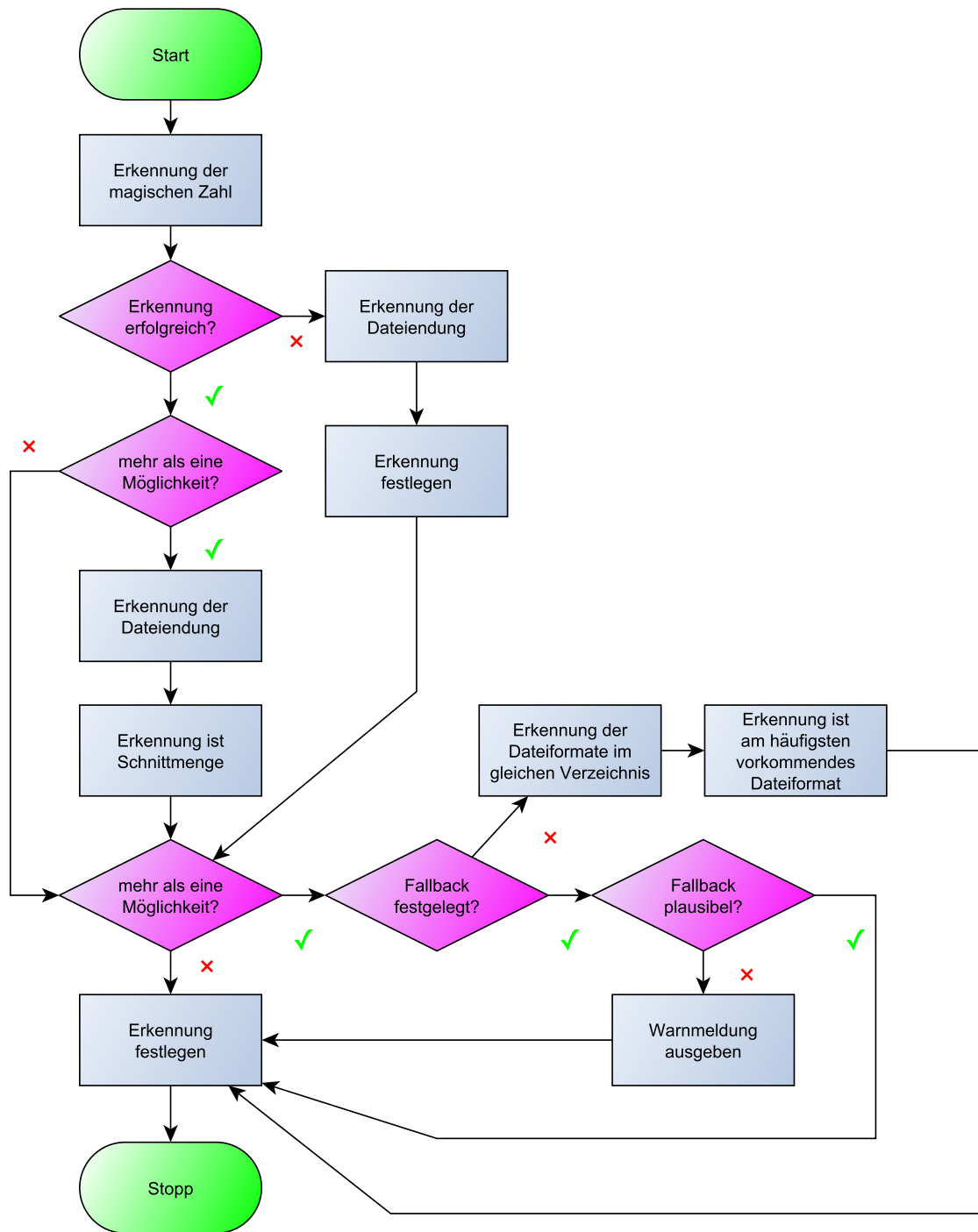


Abbildung 3.2: Gesamtalgorithmus zur Erkennung des Dateiformats

3.8 Parsen der Quelltexte

Um die Quelltexte auf Sicherheitslücken untersuchen zu können, müssen sie zuerst durch einen Parser in ein entsprechendes Format umgewandelt werden, indem wie in Abschnitt 2.6 auf Seite 14 beschrieben für die Syntax unnötige Bestandteile entfernt und einzelne Bestandteile der Syntax mit ihrem Kontext extrahiert werden.

Um den Parser zu realisieren, existieren in Python einige Bibliotheken, die jeweils ihre Vor- und Nachteile haben und daher im Folgenden miteinander verglichen werden.

Anschließend müssen die Grammatiken für die zu untersuchenden Sprachen erstellt werden, was ebenfalls exemplarisch vorgeführt wird, um späteren Benutzern eine Anleitung zu geben, wie sie eigene Grammatiken erstellen und warten können.

Nachdem die Grammatiken erstellt wurden, werden sie genutzt, um Quelltexte einzulesen. Hierzu wird darauf eingegangen, in welcher Form sie schließlich für das Framework vorliegen und wie damit gearbeitet werden kann.

Im letzten Schritt wird aus dieser Repräsentation ein AST generiert, anhand dessen die eigentliche Schwachstellensuche durchgeführt wird.

3.8.1 Auswahl einer Bibliothek zum Parsen

In der Python-Welt existieren viele verschiedene Bibliotheken, die es ermöglichen, einen Parser zu entwickeln.

Da sich die Ziele dieser Bibliotheken allerdings teilweise erheblich unterscheiden, weil einige der Projekte mehr Wert auf die Geschwindigkeit beim Parsen, einige mehr auf die Mächtigkeit der Regelsätze und einige mehr auf die einfache Les- und Schreibbarkeit legen, ist es wichtig, die Projekte miteinander zu vergleichen und ein Projekt zu finden, welches zum einen mächtig genug ist, um den Anforderungen dieser Masterarbeit zu genügen, zum anderen aber auch einfach genug in der Handhabung ist, dass es auch von späteren Anwendern weiterentwickelt und gewartet werden kann.

Da sämtliche hier vorgestellten Bibliotheken entweder direkt unter der MIT-Lizenz stehen oder eine ähnlich freie Lizenz nutzen, bei der lediglich ein Hinweis auf die verwendete Bibliothek und die Lizenz vorhanden sein muss, hat die Lizenz der benutzten Bibliothek in diesem Fall keinen Einfluss auf die Lizenz des Frameworks.

3.8.1.1 PyParsing

PyParsing versucht, einen alternativen Ansatz zur üblichen Entwicklung von Grammatiken mit Lex und Yacc³¹, zwei Standardwerkzeugen für die Compilerentwicklung, oder mittels regulärer Ausdrücken, zu schaffen.[53]

PyParsing erlaubt es dabei, Grammatiken direkt in Python-Code zu schreiben und versucht, eine gute Les- und Schreibbarkeit zu ermöglichen, indem viele Hilfsklassen bereitgestellt werden, um die Compilerentwicklung zu beschleunigen.

Die Nachteile von PyParsing sind, dass die Bibliothek vor allem auf einfache Grammatiken ausgelegt ist, die Performance bei komplexeren Grammatiken schlecht ist ([54]) und die zu schreibende Grammatik sich nicht am Industriestandard Lex und Yacc orientiert.

3.8.1.2 PLY

PLY³² ist eine rein in Python geschriebene Reimplementierung von Lex und Yacc, welche sich vor allem durch eine gute Fehlerdiagnose auszeichnet.[55]

Da sich PLY stark am Industriestandard orientiert und gleichzeitig hilfreiche Fehlermeldungen ausgibt, ist es erfahrenen Compilerentwicklern möglich, vergleichsweise einfach einen Compiler in Python zu entwickeln.

PLY versucht dabei, schnell, speichereffizient und ausreichend mächtig für real existierende Programmiersprachen wie C, C++ oder Ada zu sein.[55]

Als Nachteil von PLY ist anzusehen, dass die Erstellung eines Parsers sehr aufwendig ist und stark auf regulären Ausdrücken beruht, welche als Docstrings von Methoden geschrieben werden, die durch eine spezielle Kennzeichnung im Modul von PLY erkannt und interpretiert werden.

Hierdurch ist es auch bei PLY notwendig, die Eigenarten der Bibliothek zu lernen, um eigene Grammatiken zu erstellen.

Außerdem ist es aufwendiger, vorhandene Grammatiken zu warten, da die Komplexität durch die starke Nutzung regulärer Ausdrücke schnell ansteigt, wie in Abbildung 2.2 auf Seite 12 im Grundlagenkapitel dargestellt.

³¹Yet Another Compiler-Compiler

³²Python Lex-Yacc

3.8.1.3 SLY

SLY³³ ist eine Abwandlung von PLY, die aufgrund ihrer moderneren Implementierungsweise vom Autoren der beiden Bibliotheken empfohlen wird.[\[55\]](#)

Die Syntax wurde gegenüber PLY vereinfacht, da statt den Docstrings Dekorierer genutzt werden, anhand derer die Tokens innerhalb der Methoden interpretiert werden.

SLY hat allerdings den gleichen Nachteil wie PLY, dass die erstellten Grammatiken stark auf regulären Ausdrücken basieren, was die Wartbarkeit erschwert.

Weiterhin ist das SLY-Projekt noch vergleichsweise jung, sodass nur wenige Beispiele und Anleitungen existieren, was die Erstellung eigener Grammatiken für Anwender maßgeblich erschwert.

3.8.1.4 Parsimonious

Parsimonious ist entstanden, um ein Problem von LALR(1)³⁴-Parsern, bei denen nur ein Token voraus geschaut werden kann, wie Yacc zu lösen: uneindeutige Syntaxregeln.

Als Beispiel beschreibt der Autor in einer Präsentation, dass er einen Link mit dem Format `[[Titel|Link]]` parsen wollte. LALR(1)-Parser stießen in diesem Fall auf Probleme, wenn der Anfang des Textes zwar wie ein Link aussah, allerdings das Token zum Beenden des Links fehlte, wie im Beispiel von `[[Titel|Link [[doch kein Link.`[\[56\]](#)

In diesem Fall sollte der Parser erkennen, dass es sich um keinen korrekten Link handelt und er entsprechend stattdessen den Text als normalen Text behandeln sollte.

Da LALR(1)-Parser allerdings nur bis zum nächsten Token vorausschauen können, tritt in diesem Fall ein Fehler auf, da sie bereits die Erkennung als Link vorausgesetzt haben und jetzt keinen korrekten Link vorfinden.

Parsimonious versucht weiterhin, ebenfalls gut les- und schreibbare Grammatiken zu ermöglichen, indem es sich an einer vereinfachten EBNF³⁵ orientiert.

Auch hier ist allerdings einer der Hauptnachteile, dass durch die starke Verwendung von regulären Ausdrücken die Wartbarkeit leidet.

³³Sly Lex-Yacc

³⁴Look Ahead Left-to-Right

³⁵Erweiterte Backus-Naur-Form

3.8.1.5 Vergleich und Begründung der Entscheidung

Von den vorgestellten Bibliotheken scheint PyParsing am besten für die Zwecke dieses Frameworks geeignet zu sein.

Dies liegt vor allem daran, dass es einfach ist, Grammatiken zu schreiben, sodass auch Anwender ohne Kenntnisse im Compilerbau in der Lage sind, Grammatiken für die von ihnen eingesetzten Sprachen zu erstellen, anstatt hoffen zu müssen, dass jemand anders diese Grammatiken für sie erstellt, oder die statische Analyse für diese Sprachen ausfallen zu lassen.

Zwar ist PyParsing weniger mächtig, als die anderen vorgestellten Bibliotheken, allerdings ist es für das entwickelte Framework bereits ausreichend, wenn die Definition von Modulen, Objekten und Funktionen sowie der Aufruf dieser und Zuweisungen korrekt erkannt werden können, um Schwachstellen zu erkennen.

Eine komplette Grammatik, mit welcher der gesamte Quelltext untersucht werden kann, ist zum einen sehr aufwendig zu erstellen, zum anderen allerdings auch weniger robust, da in anderen Versionen der Programmiersprache die Syntax geändert sein kann, wodurch die erstellte Grammatik nicht mehr fehlerfrei durchläuft.

Wie in Abschnitt 1.2 auf Seite 2 beschrieben, ist es weiterhin nicht Ziel des Frameworks, die korrekte Syntax eines Quelltextes sicherzustellen, sodass es einfacher ist, zu versuchen, nur die relevanten Teile des Quelltextes zu parsen und die restlichen Teile zu ignorieren.

Diese Entscheidung bedeutet allerdings nicht, dass Anwender nicht auch die Möglichkeit haben, andere Bibliotheken zum Parsen der Quelltexte zu verwenden.

Stattdessen erlaubt es das Design über die abstrakte Basisklasse `AbstractGrammar`, beliebigen eigenen Code zu verwenden, um die Grammatik für den Parser zu implementieren, solange die entsprechenden Methoden der Basisklasse implementiert sind und die erwarteten Rückgabewerte haben.

Hiermit wird wieder eines der Hauptziele des Frameworks, eine möglichst hohe Flexibilität für den Endanwender, gefördert.

Im nächsten Unterabschnitt wird jedoch nur auf die Erstellung von Grammatiken mittels PyParsing eingegangen, da dies wie oben erwähnt am realistischsten der Situation für Endanwender entspricht.

3.8.2 Erstellung der Grammatiken

Um aufzuzeigen, wie einfach es ist, eine neue Grammatik für das Framework zu schreiben, wird im Folgenden beispielhaft erläutert, wie die Grammatik zum Parsen von C-Quelltexten erstellt wurde.

Es ist hierbei wichtig zu beachten, dass keine vollständige Grammatik erstellt wird, sondern nur die für die spätere Schwachstellenerkennung notwendigen Bestandteile aufgenommen werden.

Eine derartige unvollständige Grammatik hat dabei einige Vorteile, aber auch einige Nachteile gegenüber vollständigen Grammatiken.

Zum einen sind unvollständige Grammatiken weitaus schneller zu erstellen, als vollständige, da viele Bestandteile weggelassen werden können, was auch die Wartung vereinfacht.

Außerdem sind unvollständige Grammatiken robuster gegenüber Änderungen in der Syntax der Programmiersprache, da neue syntaktische Möglichkeiten nicht zum Abbruch des Parsens führen, sondern einfach überlesen und ignoriert werden.

Für die Zwecke dieses Frameworks ist es dabei nur erforderlich, Zuweisungen, Funktionsdeklarationen, Funktionsaufrufe sowie Rückgabewerte korrekt zu erkennen, um somit analysieren zu können, wie der Kontrollfluss des Programms aussieht und ob es hierbei zu Schwachstellen kommen könnte.

Auf der anderen Seite haben unvollständige Grammatiken den Nachteil, dass es hiermit schwieriger ist, die Codekomplexität mittels der in Abschnitt 2.13.1 auf Seite 61 vorgestellten McCabe-Metrik zu bestimmen, da wie oben erwähnt Kontrollstrukturen nicht unbedingt erkannt werden.

Es bietet sich daher an, eine Art Pseudoerkennung für Kontrollstrukturen einzuführen, bei der lediglich ein zu der Kontrollstruktur gehöriges Schlüsselwort gesucht wird und die Anzahl der Treffer gezählt wird.

3.8.2.1 Erstellung einer unvollständigen Grammatik für C

Im Folgenden wird erklärt, wie vorgegangen werden kann, um mittels PyParsing eine unvollständige Grammatik für C-Programme zu erstellen.

Zwar ist es wie in Abschnitt 3.8.1.5 auf der vorherigen Seite auch möglich, beliebige andere Bibliotheken zu benutzen, um eine Grammatik zu erstellen, jedoch wird PyParsing als einfache und mächtige Alternative aufgezeigt, um zu demonstrieren, dass es hiermit problemlos möglich ist, Grammatiken für eigene, bisher noch nicht vom Framework unterstützte Programmiersprachen zu erstellen.

Selbstverständlich kann im Folgenden keine vollständige Anleitung zur Erstellung eines Parsers mit PyParsing erstellt werden, es wird allerdings versucht, auf einige unter Umständen verwirrende oder uneindeutige Konstrukte hinzuweisen und sie zu erklären sowie eine Grundlage zu schaffen, anhand derer Endanwender unter Zuhilfenahme der bisherigen Grammatiken sowie der offiziellen Dokumentation unter [\[54\]](#) schnell in der Lage sein sollten, eigene Grammatiken zu erstellen.

Das Programm zum Erstellen der Grammatik muss dabei im Verzeichnis `modules` innerhalb des gewünschten Moduls als `grammar.py` erstellt werden.

Um eine grobe Struktur vorzugeben, ist es sinnvoll, von der zuvor bereits erwähnten abstrakten Oberklasse `AbstractGrammar` zu erben, welche die vom Framework erwarteten Methoden enthält.

Der Konstruktor der Klasse sollte als Eingabe ein `InputFile`-Objekt erwarten, welches an den Konstruktor der Oberklasse weitergereicht wird.

Anschließend kann die eigentliche Beschreibung der Grammatik erfolgen.

Es bietet sich an, hierbei möglichst viele Hilfsstrukturen zu erstellen, um eine einfache Erweiterbarkeit und die größtmögliche Übersichtlichkeit zu gewährleisten.

So könnte als Erkennung für eigene Bezeichnungen für Variablen und Funktionen in C Folgendes genutzt werden:

Listing 3.7: Erkennung von Bezeichnungen in C

```
1 self.ident = Word(alphas, alphanums + "_")
```

Hierdurch werden Wörter erkannt, welche mit einem Buchstaben beginnen und anschließend eine Kombination aus Buchstaben, Zahlen und dem Unterstrich haben.

Diese Erkennung kann anschließend weiterverwendet werden, zum Beispiel bei der Erkennung von Variablentypen.

Listing 3.8: Erkennung von Variablentypen

```
1 self.vartype = Suppress(Combine(Optional(oneOf("signed " \
2                                     "unsigned"))) +
3                               self.ident +
4                               Optional(Word("*")),
5                               adjacent=False))
```

In dieser Erkennung werden die Variablentypen mittels `Suppress` unterdrückt, da sie für die Erkennung von Schwachstellen nicht relevant sind.

Anschließend werden Variablentypen definiert als eine Kombination von einem optionalen Schlüsselwort `signed` oder `unsigned`, dem eben definierten Erkennungstypen für Bezeichnungen und schließlich einem oder mehreren Sternen, welche die Variable als Zeiger identifizieren.

Um die Grammatik später im Programm einfacher verarbeiten zu können, können nach den Tokendefinitionen Bezeichnungen hinzugefügt werden, wodurch es später im Code

möglich ist, per Schlüsselwort auf das Token zuzugreifen, indem in Klammern nach der Definition ein Bezeichner eingetragen wird.

Der Schlüsselwortparameter `adjacent` der `Combine`-Klasse wird dabei auf `False` gesetzt, um dafür zu sorgen, dass Leerzeichen zwischen den einzelnen Tokens nicht die Erkennung verhindern.

Eine weitere wichtige Eigenheit von PyParsing sind die sogenannten Forward-Deklarationen.

Da es sich bei PyParsing um einen Recursive Descent-Parser handelt, kann es vorkommen, dass Tokens verwendet werden, welche zum Zeitpunkt der Verwendung noch nicht definiert sind.

Um dieses Problem zu beheben, können diese Tokens mittels `Forward()` definiert und im späteren Verlauf mit Erkennungen befüllt werden.

Als Beispiel hierfür soll die Deklaration von Funktionskörpern dienen, bei denen es notwendig ist, Statements und Funktionsprototypen vorab zu definieren, die allerdings zu diesem Zeitpunkt in der Grammatik noch nicht existieren.

Listing 3.9: Definition der Erkennung von Funktionskörpern

```
1 self.stmt = Forward()
2 self.prototype = Forward()
3 self.func_body = Group(
4     OneOrMore(Group(SkipTo(self.stmt, failOn=self.prototype,
5                           include=True))))
```

Wie hier zu sehen ist, werden Funktionskörper definiert als eine Gruppe aus einer oder mehreren Gruppen, welche wiederum aus Statements bestehen.

Um zu erreichen, dass der Parser nicht abstürzt, wenn er auf einen unbekannten Token stößt, wird hierbei die `SkipTo`-Klasse verwendet, welche so lange Tokens liest und direkt verwirft, bis sie ein weiteres Statement erreicht.

Wenn `SkipTo` dabei über einen Funktionsprototypen läuft, wird die Rekursion an dieser Stelle beendet und der Prototyp wird nicht in die Erkennung aufgenommen.

Der Schlüsselwortparameter `include` führt hierbei dazu, dass der Parser die unbekannten Tokens nicht einfach überliest, sondern auch die Auswertung des Quelltextes erst nach den übersprungenen Tokens weiter fortsetzt.

Um schließlich die Forward-Deklarationen mit Erkennungsmustern zu befüllen, wird mittels `<<` im späteren Programm der Inhalt festgelegt.

Listing 3.10: Befüllen einer Forward-Deklaration

```
1 self.stmt << (self.func_call("func_call") |  
2             self.assignment("assignment") |  
3             self.return_("return")) + Suppress(";")
```

Nachdem sämtliche benötigten Strukturen und Tokens definiert wurden, müssen noch einige Methoden erstellt werden, welche von der **AbstractGrammar** vorgegeben wurden und im späteren Programm für die Auswertung genutzt werden.

Da diese Methoden beliebige Abschnitte der Quelltexte durchsuchen können sollen, sollte die Funktion **scanString** der jeweiligen Tokendefinition genutzt werden, durch welche das Parsen nicht unbedingt am ersten Zeichen der Eingabe starten muss, sondern beliebige Zeichen am Anfang überliest, bis die korrekte Eingabe gefunden wurde.

Es kann weiterhin sinnvoll sein, vor dem Beginn des Parsens mittels **ParserElement.enablePackrat()** das sogenannte Packrat-Parsing zu aktivieren, wodurch Ergebnisse des Parsevorgangs memoisiert werden, was zu einer deutlichen Performancesteigerung bei umfangreicheren Quelltexten führen kann.

Sollte die eigene Grammatik allerdings Parseraktionen durchführen, welche in PyParsing dazu dienen, das Ergebnis des Parsevorgangs direkt weiterzuverarbeiten, so könnten sich durch Packrat schwer auffindbare Fehler ergeben.

Da Parseraktionen allerdings in den bisher erstellten Grammatiken nie notwendig waren, werden sie hier nicht näher besprochen und das Aktivieren von Packrat wird empfohlen.

3.8.3 Repräsentation im Programm

Um die Ergebnisse des Parsers im Programm darzustellen, wird von PyParsing eine eigene Datenstruktur verwendet, auf welche man wie eine Kombination aus Listen und Hashmaps (in Python als »Dictionaries« bezeichnet) zugreifen kann.

Die Listen enthalten dabei die erkannten Tokens, denen keine Bezeichnung zugeordnet wurde, die Dictionaries dagegen enthalten als Schlüssel die Bezeichnung und als Wert das Token.

So kann man in der folgenden Auswertung einer C-Funktion sowohl über den Index, also mittels **parsetree[0][0]**, auf den Funktionsnamen zugreifen als auch über die Bezeichnung, also mittels **parsetree[0]['name']**.

Listing 3.11: Repräsentation einer einfachen C-Funktion

```
1 (('add',  
2  (['a'], {'name': ['a']}),  
3  (['b'], {'name': ['b']}),
```

```

4      ([(['', 'c', ([('a', 'b'], {})]),
5      {'lvalue': ['c'],
6      'expression': [(['a', 'b'], {})]},
7      'assignment': [(['c', ([('a', 'b'], {})]), {}]])),
8      ([(['', 'c'], {'return_value': ['c'], 'return': [(['c'], {})]})),
9      {})],
10     {'name': ['add'],
11     'args': [([(['a'], {'name': ['a']}]), ([('b'], {'name': ['b']}])), {}]],
12     'body': [([(['', 'c', ([('a', 'b'], {})]),
13     {'lvalue': ['c'],
14     'expression': [(['a', 'b'], {})]},
15     'assignment': [(['c', ([('a', 'b'], {})]), {}]])),
16     ([(['', 'c'], {'return_value': ['c'], 'return': [(['c'], {})]})),
17     {}]])),
18     0,
19     58)

```

Dies erleichtert die spätere Auswertung ungemein, da es möglich ist, mittels einfacher Abfragen zu untersuchen, ob ein Token an einer bestimmten Stelle im Parsetree existiert.

Die verwendeten Bezeichnungen für die Tokens müssen dabei einheitlich sein, damit bei der späteren Analyse auf sie zurückgegriffen werden kann.

Es ist daher sinnvoll, sich bei der Erstellung neuer Grammatiken nach Möglichkeit an bestehenden Grammatiken zu orientieren und auch deren Bezeichnungen zu übernehmen, um ein gutes Ergebnis zu erzielen.

Sollten die Bezeichnungen fehlen, kann das Framework im Nachhinein nicht mehr identifizieren, um welches Token es sich handelt, und kann daher keine Schwachstellenanalyse durchführen.

Die beiden Zahlen am Ende geben an, in welcher Spalte der Funktionsbeginn bzw. das Funktionsende erkannt wurden.

Wenn andere Bibliotheken zur Erstellung der Grammatiken benutzt werden sollen, müssten deren Ergebnisse so umgewandelt werden, dass sie mit der hier vorliegenden Form kompatibel sind, bevor sie vom Framework verwendet werden können.

Da Python Duck-Typing einsetzt, ist es hierbei lediglich erforderlich, dass die Ergebnisse sich anhand der Bezeichner und Listenindizes zuordnen lassen und die zugehörigen Angaben zur Start- und Endposition im String in einer Liste an der zweiten, bzw. dritten Stelle auftauchen, um eine Kompatibilität mit dem restlichen Framework zu gewährleisten.

3.8.4 Analyse des AST

Für die Analyse des AST werden in der Klasse `AbstractGrammar` einige abstrakte und auch konkrete Methoden vorgegeben, welche helfen, relevante Informationen aus dem Parsetree zu extrahieren.

Bei den abstrakten Methoden handelt es sich zum Beispiel um `getMethodDefinitions()`, welche sämtliche Methoden innerhalb einer Datei zurückgibt, `getMethodCalls(start, end)`, welche sämtliche Methodenaufrufe zwischen zwei Spalten der Eingabe zurückgibt, `getAssignments(start, end)`, welche alle Zuweisungen zwischen zwei Spalten der Eingabe zurückgibt und `getReturns(start, end)`, welche alle Rückgabewerte einer Methode zurückgibt.

Diese Methoden müssen von der jeweiligen Grammatik konkret implementiert werden und werden von den folgenden konkreten Methoden verwendet, um eine Analyse des Datenflusses zu ermöglichen.

In der Analyse-Klasse werden diese Methoden dann benutzt.

`followVariables(method)` gibt eine Übersicht über die Zuweisungen und Funktionsaufrufe, in welchen Variablen verwendet werden.

Um den Ursprung der Variablen zurückzuverfolgen, kann `findVariableSource(method, objectName, variable, start)` verwendet werden, wodurch Variablen ab der Position `start` zurück zur letzten Zuweisung innerhalb der gleichen Methode verfolgt werden.

Durch diese Funktionen ist es möglich, nachzuvollziehen, wie sich Variablen im Code ausbreiten und damit herauszufinden, ob eine Verunreinigung möglich ist.

3.9 Parsen der Regelsätze

Die Regelsätze werden wie in Abschnitt 2.12.3.5 auf Seite 59 beschrieben in YAML geschrieben und in Python mittels PyYAML ausgewertet.

Hierzu ist es zuerst wichtig, zu entscheiden, für welches Modul die Regelsätze geladen werden sollen.

Für diese Entscheidung dient das Ergebnis aus Abschnitt 3.7 auf Seite 74, welches der Regelsatzklasse `Ruleset` im Konstruktor mitgegeben wird.

Anhand dieses Moduls werden anschließend die Regeln für Senken und deren zugehörige Bereinigungen sowie die Regeln für die Quellen geladen.

Um dies zu erreichen, werden alle Dateien mit den Endungen `.yaml` oder `.yml` im Unterverzeichnis `modules/MODULE/sinks` bzw. `modules/MODULE/sources` mittels PyYAML ausgelesen.

Hierbei wird, aus den in Abschnitt 2.12.3.3 auf Seite 54 beschriebenen Gründen, die Methode `yaml.safe_load` eingesetzt, um Code Execution durch bösartige Regeln zu verhindern.

Das Ergebnis wird anschließend an die entsprechenden Klassen für die Repräsentation der einzelnen Regeln übergeben.

3.9.1 Definition von Schwachstellen

Für die Definition einer Schwachstelle mit den zugehörigen Absicherungen wird die Klasse `Sink` eingesetzt.

Diese prüft zunächst den Objektnamen aus der YAML-Regel, wobei wie in Abschnitt 2.12.2.3 auf Seite 48 beschrieben jeweils nur ein Objekt pro Regeldatei beschrieben wird, sodass der erste Eintrag des übergebenen Dictionaries als Objektnamen erkannt wird.

Im Anschluss kann auf die Methoden über den Schlüssel `Methods` zugegriffen werden.

Um auch die Absicherungen von einem Dictionary in Objekte der Klasse `Sanitizer` zu übertragen, ist es erforderlich, zuerst den Originaleintrag zwischenspeichern und anschließend aus den Einträgen zu den Absicherungen jeweils `Sanitizer`-Objekte zu erstellen, welche die Dictionary-Einträge überschreiben.

Um schließlich sicherzustellen, dass auch die weiteren Inhalte der Regeldatei existieren, werden Assertions eingesetzt, welche mithilfe der Python-Funktion `all()` und List Comprehension, einer kompakten Technik zur Erzeugung von Listen, prüfen, ob alle Methoden ebenfalls einen `Methodname`, `Parameters` und `Comment` enthalten.

Da diese Assertions unter Umständen bei einer Vielzahl von Regelsätzen die Ausführung ein wenig verlangsamen können, ist es möglich, diese entweder mittels `pipenv run python -O main.py` oder mittels der Umgebungsvariable `PYTHONOPTIMIZE=TRUE` zu deaktivieren.

In diesem Fall könnte es allerdings passieren, dass im späteren Verlauf der Ausführung Fehler auftreten, sodass diese Option nur genutzt werden sollte, wenn die Performance des Frameworks problematisch ist.

3.9.2 Definition von Sicherungen

Die Definition von Sicherungen erfolgt ebenso wie die Definition von Schwachstellen, indem der erste Schlüssel im übergebenen Dictionary als Objektnamen und dessen Methoden als `Methods` erkannt werden.

Auch hier werden Assertions eingesetzt, um das Format der Regeldatei zu überprüfen, die wie oben beschrieben deaktiviert werden können.

3.9.3 Definition benutzerkontrollierter Eingaben

Auch die Definition benutzerkontrollierter Eingaben verhält sich analog zu der Definition von Sicherungen, da auch diese Klasse lediglich der einfacheren Datenhaltung dient.

Ebenso wie in den Schwachstellen und Sicherungen werden auch hier Assertions eingesetzt, um das Format zu prüfen.

3.10 Suche nach Schwachstellen

Um Schwachstellen zu erkennen, werden in der Analyse zuerst alle Klassen und Methoden der Eingabedatei erkannt.

Anschließend wird eine Datenflussanalyse durchgeführt, bei der versucht wird, sämtliche Verwendungen von Variablen zurückzuverfolgen und somit aufzulisten, an welcher Stelle welche Variable eingesetzt wird.

Die Liste der Variablen wird dabei mit allen Verwendungen abgespeichert und für alle weiteren Analyseschritte verwendet.

Da es bei der objektorientierten Programmierung allerdings vorkommen kann, dass Variablennamen mehrdeutig sind, müssen anschließend die Objektbezeichnungen innerhalb der Variablenliste angeglichen werden.

Als Beispiel hierfür seien die Schlüsselwörter `this` in Java oder `self` in Python genannt.

Diese beiden Schlüsselwörter deuten an, dass der Kontext der referenzierten Variable die aktuell umgebende Klasse ist.

Wenn nun die Analyse versucht, eine Verwendung dieser Variablen in anderen Klassen zu suchen, muss zuerst die Selbstreferenz aufgelöst und durch den passenden Klassennamen ersetzt werden.

Ferner müssen für manche Sprachen, wie Python, teilweise noch die Dateinamen als optionaler Teil des Objektnamens vorangestellt werden, da ein Zugriff auch über den Modulnamen möglich ist, welcher standardmäßig dem Dateinamen entspricht.

Eine derartige Angleichung der Objektnamen würde zwar die Interpretation der Anweisung ändern, da das Framework allerdings eine rein statische Analyse durchführt, ist dies die einzige Möglichkeit, Methoden und Variablen auch über mehrere Klassen hinweg zuverlässig zu erkennen.

Nachdem sämtliche Variablen gefunden und vereinheitlicht wurden, kann begonnen werden, mithilfe der Regelsätze nach Schwachstellen in den Quelltexten zu suchen.

Um diese Regelsätze allerdings dynamisch erweitern zu können, da zum Beispiel der Aufruf einer Senke mit Parametern, welche von den Parametern der aufrufenden Methode

abhängen, die aufrufende Methode selbst wiederum zu einer Senke macht, sind zwei Entwurfsmuster notwendig, welche im Folgenden kurz erklärt werden sollen: Das Observer-Pattern und Dependency Injection.

3.10.1 Das Observer-Pattern

Um eine dynamische Erweiterung der Regelsätze zu ermöglichen, muss die Analyse nach jeder Änderung einer Regel erneut durchgeführt werden, um sicherzustellen, dass auch die neuen Regeln während des Suchlaufs beachtet werden.

Um dies zu bewerkstelligen, wird im Framework eine einfache Variante des Observer-Patterns eingesetzt.

Beim Observer-Pattern gibt es ein zu beobachtendes Objekt und mehrere Beobachter, die über Änderungen am Objekt informiert werden müssen.

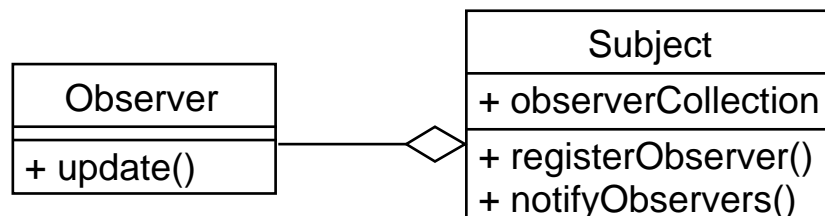


Abbildung 3.3: Einfache Form des Observer-Patterns

Das Prinzip wird in Abbildung 3.3 veranschaulicht.

Das zu beobachtende Objekt erhält eine Sammlung von Observern, welche über Änderungen am internen Zustand informiert werden sollen.

Zu diesem Zweck müssen sich die Observer bei ihrer Erstellung mittels der Funktion `registerObserver()` beim Subjekt registrieren.

Sobald sich der interne Zustand des Subjekts ändert, müssen nun die Observer mittels `notifyObservers()` darüber informiert werden, indem auf jedem Observer in der `observerCollection` die Funktion `update()` aufgerufen wird.

Es gibt dabei verschiedene Varianten des Observer-Patterns. Für das Framework wurde eine Push-Variante gewählt, bei welcher der `update()`-Funktion der Observer direkt die geänderten Daten mitgegeben werden.

Weiterhin wurde auf Funktionen zum Löschen von Observern verzichtet, da es hierfür keinen Bedarf gibt, auch wenn diese Funktion im Pattern eigentlich enthalten ist.

3.10.2 Dependency Injection

Ein weiteres Problem, welches durch die dynamischen Regelsätze bedingt ist, besteht darin, dass die Regelsätze nicht mehrfach instantiiert werden dürfen, da es ansonsten möglich ist, dass ein Objekt eines Regelsatzes nicht alle neu hinzugefügten Regeln enthält und somit eine unvollständige Analyse durchführt.

Um dieses Problem zu beheben, wurde Dependency Injection eingesetzt.

Bei Dependency Injection werden einem Objekt die Abhängigkeiten eines anderen Objekts mitgeliefert, sodass es sich seine Abhängigkeiten nicht mehr selbst suchen muss.

Im konkreten Fall erwartet der Konstruktor der Analyseklasse eine zusätzliche Referenz auf seinen zu nutzenden Regelsatz, sodass es möglich ist, die Regelsätze einmalig für alle benötigten Programmiersprachen zu initialisieren und anschließend diese bereits initialisierten Regelsätze in die Analyse weiterzugeben, anstatt zu versuchen, aus der Analyse heraus selbst den passenden Regelsatz zu finden.

Eine Alternative zur Dependency Injection wäre es zwar gewesen, ein Singleton-, bzw. in diesem Fall ein Multiton-Pattern einzusetzen, da Regelsätze für verschiedene Module möglich sind, allerdings werden Singletons häufig als Anti-Pattern bezeichnet,[\[57\]](#) da sie wie versteckte globale Variablen agieren und somit den Code und das Testen schwerer verständlich machen.

3.10.3 Ablauf eines Suchdurchlaufs

Beim ersten Suchdurchlauf werden alle potenziell interessanten Funktionen abgespeichert, die als Senken, Quellen oder Absicherungen fungieren könnten.

Im ersten Schritt wird dabei nach Quellen gesucht, indem geprüft wird, welche Funktionsaufrufe sich innerhalb der aktuellen Methode befinden.

Wenn eine Funktion gefunden wird, deren Bezeichnung und Parameteranzahl zu einer der Regeln aus den Regelsätzen passt, wird der Funktionsaufruf als potenzielle Quelle in der Methode abgespeichert.

Nach dem gleichen Prinzip können auch die Senken einer Methode ermittelt werden.

Bei den Absicherungen dagegen ist es notwendig, die Absicherungen jeder einzelnen Senke zu suchen, da die Absicherungen direkt von den Senken abhängig sind.

Wichtig ist hierbei, dass bei keiner der Prüfungen sofort sicher ist, ob sie wirklich eine der oben genannten Funktionen haben oder nicht. Dies liegt daran, dass zu diesem Zeitpunkt noch nicht überprüft werden kann, ob die Parameter der Funktionen benutzerkontrolliert sind oder nicht, da diese Funktion erst bei der Suche nach Verschmutzungen zum Einsatz kommt.

Globale Variablen werden immer wie zusätzliche Parameter einer Funktion behandelt und somit als potenziell benutzerkontrolliert angesehen, da es nicht möglich ist, die Veränderung dieser Werte in einer statischen Analyse vorherzusagen.

3.10.3.1 Notwendigkeit der erneuten Prüfung und Suche nach Verschmutzungen

Zur Suche nach Verschmutzungen müssen die gefundenen potenziell interessanten Funktionen erneut überprüft werden, da es nun möglich ist, deren Parameter mit einer Liste von benutzerkontrollierten Variablen zu vergleichen.

Hierzu müssen zuerst die vom Benutzer potenziell kontrollierbaren Variablen ermittelt werden, also die Parameter der Funktion, Variablen, welche in Quellen verwendet werden, und Zuweisungen von benutzerkontrollierten Variablen an neue Variablen.

Im Anschluss können dann diese bekanntermaßen durch den Benutzer kontrollierten Variablen genutzt werden, um zu prüfen, ob die Parameter einer Senke wirklich durch einen Benutzer kontrolliert werden oder nicht.

Sollten benutzerkontrollierte Eingaben in eine Senke einfließen, so muss geprüft werden, ob die entstandene Verschmutzung vor der Nutzung abgesichert wurde.

Da die Absicherung allerdings wie in Abschnitt 2.12.1.3 auf Seite 44 beschrieben niemals als 100 %ige Sicherheit angesehen werden sollte, wird die Absicherung lediglich als zusätzliche Information an die Verschmutzung angehängt.

Im Anschluss daran wird die Senke zur Liste der »echten« Senken hinzugefügt, selbst wenn keine Benutzereingabe hierfür erkannt wurde, da noch immer die Möglichkeit besteht, dass die Benutzereingabe lediglich nicht durch die Regelsätze erfasst wurde.

Dies ist erneut ein Versuch, eine maximale Erkennung zu gewährleisten und lieber zu viel zu erkennen, als Sicherheitslücken unentdeckt zu lassen.

Wenn dagegen sicher ist, dass es sich um eine Verschmutzung handelt, die Senke also mit benutzerkontrollierten Daten befüllt wird, so wird versucht zu erkennen, von welchen Parametern der Methode die Parameter der Senke abhängig sind, und die Methode selbst mit ihren Parametern, welche an den Stellen, an denen sie in die Parameter der Senke einfließen, durch `$TAINT` ersetzt werden, als neue Senke aufgenommen, wenn alle Parameter der Senke durch die Parameter der Methode beeinflusst werden.

Es ist hierbei wichtig, dass nicht nur geprüft wird, ob ein Parameter der Methode von der Senke abhängig ist, sondern ob alle Parameter der Senke von denen der Methode abhängen, da die Regelsätze so konzipiert sind, dass Benutzer für den Fall, dass es egal ist, ob Parameter A oder Parameter B einer Senke benutzerkontrolliert ist, zwei einzelne Regeln erstellen sollten, bei denen jeweils einer der Parameter kontrolliert wird, sowie bei Bedarf eine weitere zusätzliche Regel für den Fall, dass beide Parameter kontrolliert werden sollen.

3.10.3.2 Generierung des Kontrollflussgraphen

Die Generierung des Kontrollflussgraphen ergibt sich aus mehreren Bestandteilen.

Zum einen wird bei der Erkennung der Variablen und Methodenaufrufe abgespeichert, an welcher Position sich die jeweiligen Treffer befinden.

Zum anderen wird eine Funktion namens `find_variable_source()` genutzt, welche in der Lage ist, anhand eines Variablennamens und einer Startposition alle Zuweisungen dieser Variable an andere Variablen zu erkennen und somit rekursiv herauszufinden, was die ursprüngliche Quelle der Variable war.

Als Beispiel soll im Folgenden versucht werden, die ursprüngliche Quelle der Variable `test` herauszufinden, welche in Zeile 7 genutzt wird.

Listing 3.12: Beispiel für eine Variablenrückverfolgung

```
1 #include <stdio.h>
2
3 int main(int argc, char * argv[]) {
4     char * foo = argv[1];
5     char * bar = foo;
6     char * test = bar;
7     printf(test);
8 }
```

Um nun herauszufinden, ob die Variable `test` von einer Benutzereingabe beeinflusst wird, muss die Variable an der Startposition in Zeile 7, dem Aufruf der `printf`-Funktion, zurückverfolgt werden.

Hierfür wird zuerst rückwärts nach der ersten Zuweisung gesucht, welche der Variable einen neuen Wert zuweist, da diese Zuweisung den Wert der Variable überschreibt.

Erkannt wird hierdurch die Variable `bar` in Zeile 6, sodass die Verfolgung nun rekursiv weitergeht und versucht, den Ursprung der Variable `bar` zu erkennen.

Auch hier wird wieder rückwärts nach der ersten Zuweisung gesucht und die Zuweisung von `foo` an `bar` in Zeile 5 entdeckt.

Die Rekursion startet daher erneut und findet die Zuweisung von `argv[1]` an `foo` in Zeile 4.

`argv` wiederum wird anschließend als Parameter der Funktion in Zeile 3 erkannt und es ergibt sich ein Verlauf von `argv` → `foo` → `bar` → `test`, die Variable ist also benutzerkontrolliert.

Zusammen mit den anderen Informationen über die Positionen von Variablen und Methodenaufrufen ergeben sich hiermit alle vom Framework für den Kontrollflussgraphen benötigten Informationen.

3.10.3.3 Exklusive Kontrollstrukturen

Der Kontrollflussgraph kann sich weiterhin ändern, indem Kontrollstrukturen benutzt werden, welche sich gegenseitig ausschließen.

Zu diesem Zweck wird im Folgenden der Begriff der *exklusiven Kontrollstruktur* eingeführt, womit solche Kontrollstrukturen gemeint sind, bei denen es nur möglich ist, jeweils eine ihrer Alternativen zu durchlaufen, aber niemals mehrere.

Das bekannteste Beispiel hierfür sind Bedingungen mittels `if`, `if else` und `else`, wobei es bei jedem Durchlauf nur möglich ist, entweder in den `if`- oder in den `if else`- oder in den `else`-Zweig zu gehen, aber niemals in mehr als einen gleichzeitig.

Auf der anderen Seite erlauben *inklusive Kontrollstrukturen* wie `for` oder `while`, dass das Programm sowohl in die Schleife wechselt, als auch anschließend den danach stehenden Code ausführt, auch wenn es nicht zwingend notwendig ist, dass die Schleife überhaupt durchlaufen wird.

Diese verschiedenen Pfade durch eine Methode werden vom Framework mittels der Methode `find_paths_through` erkannt und können anschließend getrennt voneinander durchlaufen werden, um zu verhindern, dass eine Quelle innerhalb einer Bedingung als Verschmutzung für eine Senke im Alternativblock erkannt wird.

3.10.4 Speicherung der Suchergebnisse

Die Suchergebnisse werden in Objekten für die untersuchten Methoden selbst abgespeichert.

Dabei werden sowohl die Start- und Endposition der Methode, der Methodenname und die Parameter als auch sämtliche Funktionsaufrufe und Variablen sowie alle Quellen, Senken, Absicherungen und Verschmutzungen mit abgespeichert, um zum Beispiel eine mehrfache Suche nach allen Variablen oder Funktionsaufrufen, die sich während der Analyse nicht ändern, zu verhindern.

Da es weiterhin möglich ist, neue Quellen, Senken, Absicherungen oder Verschmutzungen hinzuzufügen, existieren Methoden, um die bisherigen Treffer zu erweitern.

Ein Entfernen dieser Informationen ist dagegen nicht vorgesehen, weshalb sie entweder komplett ersetzt oder mit den entsprechenden Methoden ergänzt werden müssen.

3.11 Analyse der Codekomplexität

Für die Analyse der Codekomplexität wird wie in Abschnitt 2.13.1 auf Seite 61 beschrieben die McCabe-Metrik eingesetzt.

Um die Berechnung durchzuführen, wird die Anzahl der Knoten und der Kanten bestimmt und anschließend wie zuvor angegeben miteinander verrechnet.

Da die Bestimmung von Kanten und Knoten abhängig von der jeweiligen Sprache ist, muss deren Bestimmung in den einzelnen Grammatiken stattfinden.

Hierbei ist wichtig zu beachten, dass unterschiedliche Typen von Kontrollstrukturen zu einer unterschiedlichen Anzahl möglicher Kanten führen.

So führt eine Schleife zum Beispiel zu insgesamt drei Kanten, da sowohl der Schritt vom Start der Schleife zur ersten Anweisung innerhalb des Schleifenrumpfs, als auch der Schritt vom Start der Schleife zur Anweisung nach der Schleife, wenn sie nicht (mehr) durchlaufen wird, als auch der Schritt vom Ende des Schleifenrumpfs zurück zum Schleifenkopf betrachtet werden müssen.

Auf der anderen Seite können exklusive Kontrollstrukturen die Anzahl der Kanten nur um jeweils zwei Kanten erhöhen, da sie definitionsgemäß entweder betreten werden können oder nicht.

3.11.1 Einschränkungen der Komplexitätsprüfung

Die Komplexitätsprüfung ist dadurch eingeschränkt, dass die korrekte Berechnung der Komplexität in hohem Maß abhängig von der Implementierung innerhalb der Grammatik ist.

Es ist leider nicht möglich, hier eine feste Vorgabe zu machen, ohne den Erstellern zukünftiger Grammatiken vorzuschreiben, mit welcher Methode und welchen Bezeichnungen ihre eigenen Grammatiken erstellt werden müssen.

Da es allerdings das Ziel des Frameworks ist, eine einfache Erweiterung durch Benutzer zu ermöglichen, sodass Benutzer auch selbst innerhalb kurzer Zeit noch nicht existierende Grammatiken erstellen können, kann nicht immer gewährleistet werden, dass die Berechnung der Komplexität in jedem Fall das korrekte Ergebnis bringt.

3.12 Erstellung eines Reports

Um die Ausgaben des Frameworks sinnvoll auswerten zu können, ist es notwendig, diese ein wenig aufzubereiten.

Zu diesem Zweck kann das Framework Reports in drei verschiedenen Formaten erstellen: Als reine Textdatei, als Markdown-Datei sowie als HTML-Datei.

Diese Reports sind darauf ausgelegt, nicht mehr den Fortschritt oder Informationen zum Programmablauf innerhalb des Frameworks zu vermitteln, sondern in einer übersichtlichen Form auf mögliche Probleme innerhalb der analysierten Dateien hinzuweisen.

Zu diesem Zweck wird eine private Methode, `generate_report`, innerhalb der `Report`-Klasse eingesetzt, welche eine generische Aufbereitung der Analyseresultate vornimmt.

Hierfür werden Funktionen eingesetzt, die passende Textstücke zurückgeben und wo nötig vorher die benötigten Informationen aus den gespeicherten Ergebnissen extrahieren.

Die Methode `generate_report` formatiert weiterhin sämtliche Ausgaben, indem vor und nach jedem Textbaustein ein Markup eingefügt wird und die Textbausteine mit einer übergebenen Funktion formatiert werden.

Hierdurch ist es möglich, fast beliebige Markup-Formate zu implementieren, indem lediglich das Markup angepasst und eine passende Formatierungsfunktion implementiert bzw. angegeben wird.

Damit der Report weiß, welche Ergebnisse er wo ausgeben muss, müssen ihm die Analyseergebnisse als Liste, die maximale zyklomatische Komplexität, die maximale Entfernung zwischen der Senke und der Absicherung sowie der Ausgabestrom mitgegeben werden.

3.12.1 Erstellung eines Reports als Textdatei

Die Erstellung des Reports als Textdatei ist die einfachste Variante, da es hierbei lediglich notwendig ist, die `generate_report`-Funktion mit den Standardparametern aufzurufen, wodurch eine Textdatei ohne jegliche Auszeichnungen erstellt wird.

3.12.2 Erstellung eines Reports als Markdown-Datei

Für die Erstellung des Markdown-Reports müssen zuerst die Markupinformationen angepasst werden, indem die entsprechenden Einträge in der Klassenvariable `markup` überschrieben werden.

Anschließend wird auch hier wieder die `generate_report`-Funktion genutzt, welche die passende Ausgabe generiert.

3.12.3 Erstellung eines Reports als HTML-Datei

Für die Erstellung des Reports als HTML-Datei müssen ebenfalls die Markupinformationen angepasst werden.

Um allerdings hierbei noch eine weitere Formatierung nach Corporate Identity Standards durchzuführen, werden zusätzlich die Datei `style.css` und `logo.png` eingebunden, mit welchen Endanwender den Report an ein für sie passendes Design anpassen können.

Weiterhin wird das JavaScript `customize.js` eingebunden, mit welchem weitere Anpassungen möglich sind.

Um zu verhindern, dass durch die Nutzung von Sonderzeichen in den analysierten Dateien Probleme bei der Anzeige auftreten, werden sämtliche Ausgaben mittels `html.escape` formatiert, um Sonderzeichen zu filtern.

4. Test des Frameworks

Um die Praxistauglichkeit des Frameworks zu demonstrieren, wird im Folgenden gezeigt, wie man mittels Docker von beliebigen Systemen aus das Framework benutzen kann, selbst wenn das eigene System nicht die notwendigen Voraussetzungen – installiertes Python 3.7 und libmagic – erfüllt.

Anschließend wird anhand eines Beispielreports gezeigt, welche Arten von Schwachstellen mit dem Framework gefunden werden können und deren Aufkommen im Quelltext stichprobenartig nachvollzogen.

4.1 Test auf verschiedenen Systemen mittels Docker

Bei Docker handelt es sich um eine freie Software zur Containervirtualisierung für Linux, Windows und MacOS.[\[58\]](#)

Der große Vorteil von Docker besteht darin, dass es hiermit möglich ist, Anwendungen, Konfigurationen und Abhängigkeiten gebündelt in einem Container auszuliefern, wodurch keine direkte Anpassung des Hostsystems mehr notwendig ist, um Software zu installieren.

Stattdessen wird durch das Dockersystem eine vom Hostsystem isolierte Umgebung mittels CGroups³⁶, Namespaces und UnionFS³⁷ geschaffen. In Windows-Systemen, wo diese Technologien nicht in dieser Form existieren, wird mittels Hyper-V oder VirtualBox eine virtuelle Linux-Maschine erstellt, in welcher die Umgebung wie beschrieben eingerichtet wird.[\[59\]](#)

Docker baut seine Container mittels einer Beschreibung im sogenannten »Dockerfile« auf.

Für dieses Framework wird das folgende Dockerfile verwendet:

Listing 4.1: Dockerfile für das entwickelte Framework

```
1 FROM python:3.7-alpine
2 MAINTAINER Jonas A. Wendorf <jonas_wendorf@mail.de>
3 RUN apk add libmagic
4 COPY Projekt/ /app
5 WORKDIR /app
6 RUN pip install -r requirements.txt
7 RUN pipenv install --ignore-pipfile
8 ENTRYPOINT ["pipenv", "run", "python", "main.py"]
```

³⁶Control Groups

³⁷Unification File System

Die erste Zeile gibt hierbei an, dass das neue Image basierend auf einem Linux Alpine Image mit vorinstalliertem Python 3.7 gebaut werden soll.

Linux Alpine hat hierbei den Vorteil, dass es sehr klein ist und somit auch die Dateigröße des resultierenden Images geringer hält, als dies mit größeren Basisimages wie Windows Server Core oder Debian Stretch, den anderen beiden offiziell im Docker Hub verfügbaren Basisimages für Python 3.7, möglich wäre.

Auf der anderen Seite ist Linux Alpine so minimalistisch, dass in der dritten Zeile libmagic als Abhängigkeit nachinstalliert werden muss, da dieses standardmäßig nicht installiert ist.

Die darauffolgende Zeile gibt an, dass das Projekt im Verzeichnis **Projekt** liegt und nach **/app** im Image kopiert werden soll.

Dieses Verzeichnis wird dann auch als Arbeitsverzeichnis angegeben und die Installation der Python-Pakete wird eingeleitet.

Der in der letzten Zeile angegebene Entrypoint gibt an, dass beim Start des Containers direkt das Framework gestartet werden sollte, damit der Benutzer nicht erst im Container nach dem passenden Pfad suchen muss.

Zum Bauen des Images kann anschließend auf einem Betriebssystem mit installiertem Docker der Befehl `docker build . -t taintalyzing` eingegeben werden, um mit dem Dockerfile im aktuellen Verzeichnis ein Image zu erstellen und dieses mit dem Tag »taintalyzing« zu versehen.

Anschließend kann das Dockerimage zum Beispiel mittels `docker run --rm -v "$(pwd)/appdata":/appdata taintalyzing` gestartet werden, um den Container nach der Beendigung automatisch aufzuräumen und während der Laufzeit das Unterverzeichnis **appdata** im aktuellen Verzeichnis in das Verzeichnis **/appdata** im Container weiterzureichen.

Von hier aus kann das Framework dann wie bei einer nativen Installation auf dem eigenen Betriebssystem benutzt werden mit dem einzigen Unterschied, dass nur Dateien im Verzeichnis **/appdata** auf dem Zielsystem untersucht werden können.

Wenn der Benutzer einen Report im HTML-Format erstellt, sollte außerdem darauf geachtet werden, dass die Dateien **customize.js**, **logo.png** und **style.css** ebenfalls aus dem Projektordner in den Ordner mit dem gespeicherten Report kopiert werden müssen.

Für das ältere Docker Toolbox unter Windows ist außerdem wichtig zu beachten, dass standardmäßig nur Unterverzeichnisse von **C:\Users** in Container weitergegeben werden können. Weitere Pfade müssen manuell in VirtualBox für die virtuelle Maschine freigegeben werden.^[60]

4.2 Beispielreport und Auswertung

Für den folgenden Beispielreport wurde der Terminaldateimanager cfiles von Manan Singh mit Stand vom 17. Januar 2019 genutzt (Commit [a4e8ba74f2662f2df5a45deb3a4377936d5141fb](#)).[\[61\]](#)

Das Projekt wurde ausgewählt, da es zum einen in C geschrieben ist und einige der in dieser Sprache typischsten Sicherheitslücken aufweist, zum anderen, weil die besagten Sicherheitslücken mittlerweile korrigiert wurden und somit im Rahmen dieser Masterarbeit keine Zero-Day-Schwachstellen veröffentlicht werden, welche Benutzer des Projekts in Gefahr bringen könnten.

Für die Auswertung wurde die folgende Regel zum Suchen nach Quellen genutzt:

Listing 4.2: Regel für die Suche nach Quellen

```
1 ---
2 null:
3     Methods:
4     - Methodname: getenv
5     Parameters: [null]
6     Comment: Reads environment variables
```

Weiterhin wurden folgende Regeln eingesetzt, um Senken zu finden:

Listing 4.3: Regeln für die Suche nach Senken

```
1 ---
2 null:
3     Methods:
4     - Methodname: sprintf
5     Parameters: [null, null, $TAINT]
6     Comment: No check for destination buffer size, use snprintf
           instead.
7     - Methodname: strcpy
8     Parameters: [null, $TAINT]
9     Comment: No check for destination buffer size, use strncpy
           instead.
10    - Methodname: system
11    Parameters: [$TAINT]
12    Comment: Executes arbitrary system commands.
```

Insgesamt wurden bereits mit nur diesen Regeln 24 Verschmutzungen und 31 Senken gefunden. Weitere Schwachstellen existieren im Code, dieses Beispiel soll allerdings zeigen, wie typisch es für Entwickler ist, einen Fehler durchgängig zu machen, und wenn bei der Korrektur nur einer dieser Fehler übersehen wird, kann es zu fatalen Konsequenzen führen.

Im Folgenden sind daher stichprobenartig einige der Befunde zusammen mit einem Vergleich des Quelltextes aufgeführt.

Der vollständige Report im Reintext- und HTML-Format befindet sich in den Anlagen.

»Analysis results for method “init” (lines 134 to 171).

The following taints were detected:

In line 145 a call with potentially user controlled input is made to `sprintf`.

The following comment is linked to this sink: No check for destination buffer size, use `snprintf` instead.

No sanitizer detected.

Severity level: 100%.«

Im Code wird in Zeile 145 folgender Aufruf benutzt:

```
sprintf(editor, "%s", getenv("EDITOR"));
```

Wie der Kommentar aus dem Report bereits andeutet, findet bei `sprintf` keine Prüfung der Größe des Zielpuffers statt, entsprechend könnte ein Angreifer die Variable `editor`, welche in Zeile 53 als `char editor[20]` deklariert wird, überfluten, indem er einen Eintrag mit mehr als 20 Zeichen verwendet.

Die Senke wird dadurch zur Verschmutzung, dass im gleichen Aufruf zu `sprintf` ein weiterer Aufruf zu `getenv("EDITOR")` enthalten ist, wodurch die Umgebungsvariable `EDITOR` eingelesen wird.

Ein Angreifer, der Zugriff auf die Umgebungsvariablen hat, kann diese Variable entsprechend abändern und anschließend die Kontrolle über das System übernehmen.

Ein ähnliches Problem befindet sich in Zeile 491.

Hier meldet der Report

»Analysis results for method “getImgPreview” (lines 473 to 521).

The following taints were detected:

In line 491 a call with potentially user controlled input is made to `sprintf`.

The following comment is linked to this sink: No check for destination buffer size, use `snprintf` instead.

No sanitizer detected.

Severity level: 100%.«

Im Code wird an dieser Stelle die folgende Zeile genutzt:

```
sprintf(getdimensions_command,"echo -e \'5;%s\'|/usr/lib/w3m/w3mimgdisplay",filepath);
```

Hier wird in den in Zeile 484 deklarierten Puffer `char getdimensions_command[250]` die Eingabe von `filepath` geschrieben, ohne die Größe von `getdimensions_command` vorher zu prüfen.

Die Senke wird in diesem Fall zur Verschmutzung, weil die Variablenverfolgung ergibt, dass `filepath` in Zeile 473 als Parameter an die Methode übergeben wird und somit potenziell benutzerkontrolliert ist.

Ein interessanter Fall ergibt sich in Zeile 519, in der Methode `getImgPreview`, wo `system` genutzt wird, um einen Systembefehl auszuführen:

```
system(imgdisplay_command);
```

Diese Stelle ist dadurch interessant, dass in der vorherigen Zeile die in diesem Aufruf verwendete Variable `imgdisplay_command` durch `sprintf` basierend auf dem vom Benutzer kontrollierten `filepath` befüllt wird:

```
sprintf(imgdisplay_command, "echo -e '0;1;%d;%d;%d;%d;;;;;%s\n4;\n3;' | /usr/lib/w3m/w3mimgdisplay", maxx+maxx/5, 8, width, height, filepath);
```

Der Aufruf zu `system` wird allerdings nicht als Verschmutzung erkannt, da `sprintf` in den Regeln nicht als Quelle markiert wurde.

Dies demonstriert die zuvor beschriebene Notwendigkeit, auch Senken aufzunehmen, für die keine Quelle gefunden werden konnte, da es sein kann, dass lediglich eine entsprechende Regel für die Quellen fehlt.

Die entsprechende Ausgabe im Report ist daher: »In line 519 a call without any detected user controlled input is made to system.

The following comment is linked to this sink: Executes arbitrary system commands.

Severity level: 50%.«

Auch die Verfolgung von verwundbaren Methodenaufrufen lässt sich an diesem Projekt gut nachvollziehen.

So wird in der Methode `main` in Zeile 1061 die Methode `getPreview` aufgerufen, welche wiederum in Zeile 616 `getArchivePreview` aufruft.

Da in `getArchivePreview` eine Verschmutzung erkannt wurde, allerdings keine benutzerkontrollierte Eingabe für den Aufruf von `getPreview` in der `main`-Methode erkannt wurde, wird dieser Aufruf im Report lediglich als Senke und nicht als Verschmutzung ausgegeben: »In line 1061 a call without any detected user controlled input is made to getPreview.

The following comment is linked to this sink: Calls getArchivePreview from None.

Severity level: 50%.«

Die Ausgaben, dass eine Methode »from None« aufgerufen wird, bedeutet dabei jeweils, dass keine Objektzugehörigkeit erkannt wurde.

5. Evaluierung

Im Folgenden werden die in dieser Masterarbeit erreichten Ergebnisse zusammengefasst, Empfehlungen für den Einsatz des entwickelten Frameworks vorgestellt und es wird ein Ausblick gegeben auf mögliche weitere Entwicklungen des Frameworks, die aufgrund der zeitlichen Begrenzung und dem Universalitätsgedanken nicht behandelt werden konnten.

5.1 Erreichte Ergebnisse

In dieser Ausarbeitung wurde ein Framework erstellt, welches zur statischen Analyse von Quelltexten dient, um Sicherheitslücken zu entdecken.

Das Besondere an diesem Framework ist, dass versucht wurde, es möglichst universell einsetzbar zu gestalten, sodass es späteren Anwendern möglichst leicht sein sollte, das Framework nicht nur mit eigenen Regeln, sondern auch mit eigenen Grammatiken für gänzlich neue Programmiersprachen zu erweitern.

Als Beispiele wurden drei Module für Programmiersprachen implementiert, die jeweils aus den Grammatiken zur Beschreibung der Syntax sowie aus den Regeln zur Beschreibung der Quellen, Senken und Absicherungen bestehen.

Die Grammatik für C ist hierbei die ausgereifteste, mit der die meisten Syntaxelemente gut erkannt werden können.

Bei der Grammatik für PHP soll vor allem gezeigt werden, wie Grammatiken für objekt-orientierte Programmiersprachen erstellt werden können.

Die Grammatik für Python hat als Besonderheit, dass in Python nicht geschweifte Klammern, sondern Einrückungen zur Markierung von Codeblöcken benutzt werden.

Mit allen drei Grammatiken ist es möglich, nicht nur Schwachstellen in Quelltexten aufzudecken, sondern diese Schwachstellen auch über mehrere Funktionsaufrufe hinweg zu verfolgen und mit anderen Funktionen zu verknüpfen, wie jeweils mit Testfällen in der Datei `test.py` nachgewiesen wurde.

Dies zeigt einen weiteren Vorteil des Frameworks: Es ist nicht notwendig, vollständige Grammatiken für eine Sprache zu erstellen, die jedes einzelne Konstrukt genau verstehen. Stattdessen lassen sich unvollständige und damit auch robustere Grammatiken erstellen, mittels derer bereits einige Sicherheitslücken gefunden werden können.

Ob die Benutzer letztlich mehr Wert auf Genauigkeit bei der Erstellung der Grammatiken legen, was mit höherem Arbeitsaufwand und geringerer Robustheit gegenüber Änderungen an der Syntax der Programmiersprache einherginge, bleibt ihnen überlassen.

Es ist für jedes Modul möglich, mittels einfacher Regeln Funktionen zu beschreiben, welche als Benutzereingabe, potenziell verwundbare Funktion oder als Absicherung einer verwundbaren Funktion dienen.

Im Anschluss untersucht das Framework dann selbstständig die angegebenen Pfade auf diese verwundbaren Funktionen und gibt an, ob diese mit potenziell benutzerkontrollierten Eingaben aufgerufen werden, was zu einer Kompromittierung des Systems führen könnte.

Durch die Ausgabe des Reports in verschiedenen Formaten ist eine einfache Weiterverarbeitung und Anpassung möglich, was besonders den Einsatz im Geschäftsumfeld erleichtern sollte.

5.2 Empfehlungen

Das erstellte Framework kann zwar dabei helfen, Sicherheitslücken in Quelltexten zu finden, bietet aber keine absolute Sicherheit, dass alle Schwachstellen entdeckt werden.

Dies liegt zum einen daran, dass die Schwachstellensuche stark von der Anzahl und Qualität der genutzten Regelsätze abhängt, zum anderen aber auch, weil es mittels statischer Analyse nicht möglich ist, in jedem Fall alle Schwachstellen in einem Quelltext aufzudecken, wie bereits in Abschnitt 2.9 auf Seite 27 erläutert wurde.

Stattdessen sollten Anwender zusätzlich immer auch eine manuelle Analyse durchführen und die Resultate des Frameworks manuell prüfen.

Es kann jedoch sehr hilfreich sein, automatisierte Sicherheitsscans, wie sie dieses Framework bietet, in den Entwicklungsprozess mit einzubeziehen, um Schwachstellen möglichst frühzeitig und kostensparend zu entdecken und zu beheben.

Da das Framework auch in einer Docker-Umgebung läuft, sollte es immer möglich sein, das Framework auf Entwicklungsservern einzusetzen.

5.3 Ausblick

Da das Framework theoretisch beliebige Programmiersprachen unterstützt, sollten im nächsten Schritt die Programmiersprachen ermittelt werden, die in der Praxis zum einen am häufigsten eingesetzt werden und zum anderen am meisten von einem automatisierten Sicherheitsscan profitieren.

Anschließend sollten Grammatiken und Regelsätze für diese Programmiersprachen erstellt werden.

Nach Möglichkeit sollten diese Grammatiken und Regeln in einem zentralen Repository gespeichert werden, sodass später nicht mehr jeder Anwender seine eigenen Grammatiken und Regeln schreiben muss, sondern auf der Arbeit von anderen aufbauen kann, wodurch sich die Qualität der Analyse stetig verbessert.

Die statische Analyse alleine kann nicht sämtliche Sicherheitslücken in einem System finden, allerdings ist es hiermit möglich, auch selten manuell geprüfte Codezweige abzudecken, sodass sie nach Möglichkeit immer zusätzlich zur dynamischen Analyse eingesetzt werden sollte.

Mit dem hier vorgestellten Framework sind Benutzer somit in der Lage, bereits mit wenig Aufwand ihre Programme sicherer zu gestalten.

Literatur

- [1] G. Anderson, A. Weaver, C. Neill und J. Paz. (24. Aug. 2018). OWASP DefectDojo Project [Online]. Verfügbar: https://www.owasp.org/index.php/OWASP_DefectDojo_Project
- [2] S. Micheelsen und B. Thalmann, »PyT,« Magisterarb., Aalborg University, 31. Mai 2016. [Online]. Verfügbar: <http://projekter.aau.dk/projekter/files/239563289/final.pdf> (besucht am 15. 10. 2018).
- [3] A. Ronacher. (2. Mai 2018). Flask [Online]. Verfügbar: <http://flask.pocoo.org/>
- [4] Django Software Foundation. (1. Okt. 2018). The Web framework for perfectionists with deadlines | Django [Online]. Verfügbar: <https://www.djangoproject.com/>
- [5] Python Software Foundation. (2. Aug. 2018). ast — Abstract Syntax Trees [Online]. Verfügbar: <https://docs.python.org/3.5/library/ast.html>
- [6] Secure Software Inc. (31. Dez. 2013). RATS [Online]. Verfügbar: <https://code.google.com/archive/p/rough-auditing-tool-for-security/>
- [7] M. Hicks, »Symbolic Execution for finding bugs,« 15. Mai 2013, [Online]. Verfügbar: <https://www.cs.umd.edu/%7Emwh/se-tutorial/symbolic-exec.pdf> (besucht am 30. 10. 2018).
- [8] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich und M. F. Kaashoek, »Undefined Behavior: What Happened to My Code?« In *Proceedings of the Asia-Pacific Workshop on Systems*, Ser. APSYS '12, Seoul, Republic of Korea: ACM, 2012, 9:1–9:7, ISBN: 978-1-4503-1669-9. DOI: [10.1145/2349896.2349905](https://doi.org/10.1145/2349896.2349905). [Online]. Verfügbar: <http://doi.acm.org/10.1145/2349896.2349905>.
- [9] A. Gaynor. (2013). So you want to write an interpreter? [Online]. Verfügbar: <https://youtu.be/LCslqgM48D4?t=316>
- [10] D. Watson, *A Practical Approach to Compiler Construction*. Springer International Publishing, 2017. DOI: [10.1007/978-3-319-52789-5](https://doi.org/10.1007/978-3-319-52789-5). [Online]. Verfügbar: <https://doi.org/10.1007/978-3-319-52789-5>.
- [11] A. van der Stock, N. Smithline und T. Gigler, »OWASP Top 10 - 2017,« The OWASP Foundation, Techn. Ber., 20. Nov. 2017. [Online]. Verfügbar: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf (besucht am 17. 10. 2018).
- [12] E. J. Schwartz, T. Avgerinos und D. Brumley, »All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask),« in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, Ser. SP '10, Washington, DC, USA: IEEE Computer Society, 2010, S. 317–331, ISBN: 978-0-7695-4035-1. DOI: [10.1109/SP.2010.26](https://doi.org/10.1109/SP.2010.26). [Online]. Verfügbar: <http://dx.doi.org/10.1109/SP.2010.26>.

- [13] M. Andreko. (17. Dez. 2011). Exploit Exercises - Protostar Stack 5 [Online]. Verfügbar: <https://www.mattandreko.com/2011/12/17/exploit-exercises-protostar-stack-5/>
- [14] V. Kuznetsov, J. Kinder, S. Bucur und G. Candea, »Efficient State Merging in Symbolic Execution,« in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ser. PLDI '12, Beijing, China: ACM, 2012, S. 193–204, ISBN: 978-1-4503-1205-9. DOI: [10.1145/2254064.2254088](https://doi.org/10.1145/2254064.2254088). [Online]. Verfügbar: <http://doi.acm.org/10.1145/2254064.2254088>.
- [15] D. Kuske, »Berechenbarkeit und Komplexität,« 17. Jan. 2011, [Online]. Verfügbar: https://www.tu-ilmenau.de/fileadmin/public/iti/Lehre/BuK/WS10_11/buk_vorlesung_09.pdf (besucht am 19.10.2018).
- [16] M. I. Schwartzbach, »Lecture Notes on Static Analysis,« 18. Dez. 2007, [Online]. Verfügbar: http://lara.epfl.ch/w/_media/sav08:schwartzbach.pdf (besucht am 19.10.2018).
- [17] C. M. Ringel. (9. Jan. 2005). Relation, Halbordnung, Verband [Online]. Verfügbar: <https://www.math.uni-bielefeld.de/~sek/linalg/leit/poset.htm>
- [18] E. Prisner. (Apr. 2000). Verbandsordnungen/Verbände [Online]. Verfügbar: <https://www.math.tu-cottbus.de/INSTITUT/lsgdi/DM/Unix/Verband.html>
- [19] M. Kulkarni, »Control flow graphs and loop optimizations,« 14. Apr. 2011, [Online]. Verfügbar: <https://engineering.purdue.edu/~milind/ece573/2011spring/lecture-11.pdf> (besucht am 25.10.2018).
- [20] Free Software Foundation, Inc. (22. Juli 2009). Options Controlling the Kind of Output [Online]. Verfügbar: <https://gcc.gnu.org/onlinedocs/gcc-4.4.1/gcc/Overall-Options.html#index-file-name-suffix-71>
- [21] FILEExt. (15. Aug. 2018). File Extension B [Online]. Verfügbar: <http://filext.com/file-extension/B>
- [22] G. van Rossum. (8. Nov. 2018). What's New In Python 3.0 [Online]. Verfügbar: <https://docs.python.org/3/whatsnew/3.0.html#>
- [23] Python Software Foundation. (2. Aug. 2018). 20.1. html — HyperText Markup Language support [Online]. Verfügbar: <https://docs.python.org/3.5/library/html.html#html.escape>
- [24] T. Janca. (10. Aug. 2018). XSS (Cross Site Scripting) Prevention Cheat Sheet [Online]. Verfügbar: [https://www.owasp.org/index.php?title=XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet#XSS_Prevention_Rules_Summary](https://www.owasp.org/index.php?title=XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#XSS_Prevention_Rules_Summary)
- [25] The PHP Group. (13. Nov. 2018). PHP: assert - Manual [Online]. Verfügbar: <https://secure.php.net/manual/en/function.assert.php>
- [26] R. Chandel. (20. Feb. 2017). RCE with LFI and SSH Log Poisoning [Online]. Verfügbar: <http://www.hackingarticles.in/rce-with-lfi-and-ssh-log-poisoning/>

- [27] R. Seacord. (28. Apr. 2015). Input Validation and Data Sanitization [Online]. Verfügbar: <https://wiki.sei.cmu.edu/confluence/display/java/Input+Validation+and+Data+Sanitization>
- [28] J. Shore, »Fail Fast,« *IEEE Software*, Jg. 21, Nr. 05, S. 21–25, Sep. 2004. DOI: [10.1109/ms.2004.1331296](https://doi.org/10.1109/ms.2004.1331296). [Online]. Verfügbar: <https://doi.org/10.1109/ms.2004.1331296>.
- [29] T. Bray, »The JavaScript Object Notation (JSON) Data Interchange Format,« Textuality, Techn. Ber., Dez. 2017, S. 6, 16 S. [Online]. Verfügbar: <https://tools.ietf.org/html/rfc8259#section-3> (besucht am 20. 11. 2018).
- [30] B. Ippolito. (5. Dez. 2005). Remote JSON - JSONP [Online]. Verfügbar: <https://bob.ippoli.to/javascript/mochikit/ajax/archives/2005/12/05/remote-json-jsonp>
- [31] O. Ben-Kiki, C. Evans und I. döt Net. (1. Okt. 2009). YAML Ain't Markup Language (YAML™) Version 1.2 [Online]. Verfügbar: <http://yaml.org/spec/1.2/spec.html#id2708649>
- [32] I. döt Net und K. Simonov. (17. Juli 2018). PyYAML Documentation [Online]. Verfügbar: <https://pyyaml.org/wiki/PyYAMLDocumentation#loading-yaml>
- [33] R Interview Bubble. (5. Apr. 2018). Billion laughs attack (YAML Bomb) [Online]. Verfügbar: <https://interviewbubble.com/billion-laughs-attack-yaml-bomb/>
- [34] guyskk. (25. Nov. 2016). Simple YAML [Online]. Verfügbar: <https://github.com/guyskk/simple-yaml>
- [35] B. Sullivan. (Nov. 2009). Security Briefs - XML Denial of Service Attacks and Defenses [Online]. Verfügbar: <https://msdn.microsoft.com/en-us/magazine/ee335713.aspx>
- [36] N. Smithline. (14. Okt. 2017). XML External Entity (XXE) Processing [Online]. Verfügbar: [https://www.owasp.org/index.php/XML_External_Entity_\(XXE\)_Processing](https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing)
- [37] N. Ferguson und B. Schneier, »A Cryptographic Evaluation of IPsec,« Counterpane Internet Security, Inc., Techn. Ber., 5. Dez. 1999, S. 2, 28 S. (besucht am 09.09.2017).
- [38] A. H. Watson, T. J. McCabe und D. R. Wallace, »Special Publication 500-235, Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric,« in *U.S. Department of Commerce/National Institute of Standards and Technology*, 1996. [Online]. Verfügbar: <http://www.mccabe.com/pdf/nist235r.pdf>.
- [39] B. Hummel. (21. Mai 2014). McCabe's Cyclomatic Complexity and Why We Don't Use It [Online]. Verfügbar: <https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/>

- [40] Academic. (30. Apr. 2015). Betti-Zahl [Online]. Verfügbar: <http://deacademic.com/dic.nsf/dewiki/165735>
- [41] ProjectCodeMeter. (23. Aug. 2014). Cyclomatic complexity [Online]. Verfügbar: http://www.projectcodemeter.com/cost_estimation/help/GL_cyclomatic.htm
- [42] I. Bicking. (5. Okt. 2018). Virtualenv [Online]. Verfügbar: <https://virtualenv.pypa.io/en/latest/>
- [43] Python Software Foundation. (2. Okt. 2018). Installing Packages [Online]. Verfügbar: <https://packaging.python.org/tutorials/installing-packages/#creating-virtual-environments>
- [44] K. Reitz. (2. Dez. 2018). Pipenv: Python Dev Workflow for Humans [Online]. Verfügbar: <https://pipenv.readthedocs.io/en/latest/>
- [45] G. van Rossum, B. Warsaw und N. Coghlan. (1. Aug. 2013). PEP 8 – Style Guide for Python Code [Online]. Verfügbar: <https://www.python.org/dev/peps/pep-0008/>
- [46] numpydoc maintainers. (16. Juni 2018). numpydoc docstring guide [Online]. Verfügbar: <https://numpydoc.readthedocs.io/en/latest/format.html>
- [47] G. van Rossum, J. Lehtosalo und Ł. Langa. (29. Sep. 2014). PEP 484 – Type Hints [Online]. Verfügbar: <https://www.python.org/dev/peps/pep-0484/>
- [48] I. S. Cordasco. (22. Nov. 2018). Flake8: Your Tool For Style Guide Enforcement [Online]. Verfügbar: <http://flake8.pycqa.org/en/latest/>
- [49] PyCQA. (10. Apr. 2018). Pylint [Online]. Verfügbar: <https://www.pylint.org/>
- [50] The Open Group. (4. Dez. 2018). Utility Conventions [Online]. Verfügbar: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html#tag_12_01
- [51] V. Keleshev. (24. Aug. 2018). docopt creates beautiful command-line interfaces [Online]. Verfügbar: <https://github.com/docopt/docopt>
- [52] —, (2. Dez. 2018). Schema validation just got Pythonic [Online]. Verfügbar: <https://github.com/keleshev/schema>
- [53] P. McGuire. (23. Nov. 2018). PyParsing – A Python Parsing Module [Online]. Verfügbar: <https://github.com/pyparsing/pyparsing>
- [54] —, (Aug. 2018). Using the pyparsing module: Usage notes [Online]. Verfügbar: <https://pyparsing-docs.readthedocs.io/en/latest/HowToUsePyparsing.html#usage-notes>
- [55] D. Beazley. (7. Sep. 2018). PLY (Python Lex-Yacc) [Online]. Verfügbar: <http://www.dabeaz.com/ply/>
- [56] E. Rose. (12. März 2012). Parsing Horrible Things with Python [Online]. Verfügbar: <https://youtu.be/tCUdeLIj4hE?t=906>

- [57] M. Hevery. (17. Aug. 2008). Singletons are Pathological Liars [Online]. Verfügbar: <https://testing.googleblog.com/2008/08/by-miko-hevery-so-you-join-new-project.html>
- [58] Docker Inc. (28. Feb. 2019). Docker for Developers [Online]. Verfügbar: <https://www.docker.com/get-started>
- [59] —, (1. März 2019). Docker overview [Online]. Verfügbar: <https://docs.docker.com/engine/docker-overview/>
- [60] D. Procida. (1. März 2019). How to use a directory outside C:\Users with Docker Toolbox/Docker for Windows [Online]. Verfügbar: <https://support.divio.com/local-development/docker/how-to-use-a-directory-outside-cusers-with-docker-toolboxdocker-for-windows>
- [61] M. Singh. (17. Jan. 2019). cfiles [Online]. Verfügbar: <https://github.com/mananapr/cfiles/commit/a4e8ba74f2662f2df5a45deb3a4377936d5141fb>