**Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering**

Erstellung und Verfeinerung benutzerdefinierter Prüfregeln für statische Codeanalyse

Bachelorarbeit

im Studiengang Informatik

von

Pascal Elster

**Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider
Betreuer: Dipl.-Inform. Stefan Gärtner**

Hannover, 15.08.2014

Zusammenfassung

Viele potentielle Fehler in Programmcode können durch statische Analysen gefunden werden. Doch das Erstellen neuer Regeln ist aufwändig und bedarf Fachwissen. In dieser Arbeit wird ein Verfahren vorgestellt, um neue Regeln für statische Code-Analysen direkt aus annotiertem Beispiel-Code abzuleiten. Die dazu entwickelte Annotationssprache erlaubt es, die Regelerstellung ohne Vorwissen über die Funktionsweise statischer Analyseprogramme zu steuern. In den erstellten Regeln werden mittels Data-Mining-Verfahren Gemeinsamkeiten und Unterschiede gesucht, mit denen eine Verfeinerung und Vereinigung der Regeln durchgeführt wird. Es wurde ein Prototyp entwickelt, der die Methode für die Java-Programmiersprache und ein Open-Source-Analyse-Framework umsetzt. Dieser wurde anhand einer Test-Suite evaluiert. Die Ergebnisse zeigten, dass das Verfahren plausibel einsetzbar ist.

Abstract

Many potential errors in program code can be found through static program analysis. However, developing new rules is complicated and requires expertise. This work presents a method to derive rules directly from annotated sample code. The annotation language that was developed for this purpose allows the user to control the rule creation without requiring knowledge about the functionality of analysis programs. The derived rules are analyzed with data mining techniques to extract their similarities and differences, which are then used to refine and combine the rules. A prototype implementation targetting the Java programming language and an open source analysis framework was developed. It was evaluated with a test suite, and the results showed that the concept works.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation 1
1.2. Ziele der Arbeit 2
1.3. Gliederung 2

2. Grundlagen **3**
2.1. Statische Code-Analyse 3
2.1.1. Tainted Object Propagation 5
2.2. Association-Rule-Mining 6
2.3. Verwandte Arbeiten 7

3. Konzepte **9**
3.1. Anforderungen und Überblick 9
3.2. Annotationssprache 12
3.3. Regelerstellung 14
3.3.1. Bestimmung der Abhängigkeiten 15
3.3.2. Strukturelle Vorschriften 20
3.3.3. Datenfluss-Vorschriften 21
3.4. Regelverfeinerung 24
3.5. Nutzung der Vorschriften in der Analyse 28

4. Prototypische Umsetzung **34**
4.1. Analyseprogramme 34
4.1.1. FindBugs 35
4.1.2. PMD 36
4.1.3. Wahl des Frameworks 36
4.2. Einlesen des Source-Codes 37
4.3. Regelerstellung 38
4.4. Regelverfeinerung 39
4.5. Speicherung der Vorschriften 40
4.6. Integration in Analyse-Framework 41

5. Evaluation **44**
5.1. Bewertungsmaß 44
5.2. Testaufbau 45
5.3. Ergebnisse 46
5.4. Diskussion 49

6. Fazit **50**
6.1. Ausblick 50

A. Inhalt der CD **52**

1. Einleitung

1.1. Motivation

Um die Funktionsfähigkeit von Software zu gewährleisten gibt es eine Vielzahl von Testverfahren. Diese sind meist mit viel Aufwand verbunden, und im Allgemeinen muss im Voraus klar sein, was man testen möchte. Gerade mögliche Sicherheitslücken können dabei vergessen werden. Allerdings folgt eine Vielzahl von möglichen Problemen auch klaren Mustern: Eine SQL-Injektion ist im Allgemeinen beispielsweise nur dann möglich, wenn SQL-Querys aus Nutzereingaben erstellt werden, ohne, dass diese vorher ausreichend überprüft wurden. Viele weitere Sicherheitsprobleme folgen ähnlichen Mustern. Für diese Art Problem eignet sich die statische Code-Analyse. Diese kann man als Softwaretestverfahren den Glass-Box-Tests zuordnen: der Test findet direkt am Source-Code des Programms statt. Insbesondere wird bei der statischen Analyse der Code nicht ausgeführt [1].

Es gibt neben den Compilern selbst zahlreiche Werkzeuge, die statische Analysen durchführen und um neue Regeln erweitert werden können. Eines haben sie jedoch gemeinsam: Die Formulierung eigener Regeln ist aufwändig und bedarf einiger Fachkenntnisse. Da dies jedoch Voraussetzung für die effektive Nutzung der statischen Analyse ist, besteht die Gefahr, dass dieses nützliche Testverfahren von Software-Entwicklern vernachlässigt wird.

Im Allgemeinen müssen während der Erstellung einer Regel Code-Beispiele formuliert werden, um die implementierte Regel zu testen. Ein fehlerhaftes Code-Beispiel kann auch Ausgangspunkt für eine neue Regel sein. Diese Beispiele sind somit essentieller Teil der Entwicklung einer neuen Regel [2]. Deshalb bietet es sich an, die Regeln direkt aus diesen Beispielen abzuleiten. Wenn sie sich dazu eignen, die erstellte Regel zu testen, dann sollten dort auch die benötigten Informationen zur Definition des Problems enthalten sein. So ein Vorgehen würde es Software-Entwicklern erleichtern, die statische Analyse effizient als Testverfahren einzusetzen, da die komplizierte Hürde der Regelerstellung größtenteils wegfällt. Aber auch Sicherheitsexperten können davon profitieren: Wenn Regeln zur Erkennung von Sicherheitsproblemen direkt aus fehlerhaften Beispielen abgeleitet werden können, verringert sich der Zeitaufwand für diesen Schritt erheblich. Dadurch können sie in der selben Zeit potentiell deutlich mehr erreichen, als wenn sie die Regeln manuell formulieren müssen.

1.2. Ziele der Arbeit

Ziel dieser Arbeit ist es, das beschriebene Verfahren umzusetzen. Es soll eine Methode entwickelt werden, mit der Regeln für die statische Code-Analyse aus Beispielen für eine fehlerhafte Ausprägung abgeleitet werden können. Ein besonderer Fokus soll dabei auf Regeln liegen, die zur Erkennung von Sicherheitsrisiken wie der SQL-Injektion genutzt werden können. Diese haben oftmals verschiedene Ausprägungen, weshalb es darüber hinaus möglich sein soll, eine Regel durch weitere Beispiele zu erweitern und verfeinern. Beides soll vom Benutzer keine umfassenden Kenntnisse über die Thematik erfordern. Die entstehenden Regeln sollen schließlich in ein bestehendes Statisches-Analyse-Framework integriert werden.

1.3. Gliederung

Die Arbeit gliedert sich in 6 Kapitel. Kapitel 1 beschreibt die Ziele und die Motivation der Arbeit. In Kapitel 2 werden zum Verständnis der Arbeit notwendige Grundlagen im Gebiet der statischen Analyse und des Data-Minings erklärt. Kapitel 3 stellt den Hauptteil der Arbeit dar, hier werden die erarbeiteten Konzepte vorgestellt und erläutert. Dazu werden zunächst die Anforderungen erläutert und ein Überblick über die zu lösenden Teilprobleme gegeben. Diese werden schließlich in den folgenden Abschnitten im Detail vorgestellt. In Kapitel 4 werden zunächst erweiterbare Frameworks zur statischen Code-Analyse verglichen und das geeignetste für die Realisierung der erarbeiteten Konzepte ausgewählt. Anschließend werden anhand eines entwickelten Prototypen Implementierungsdetails erläutert. Der Prototyp wird in Kapitel 5 genutzt, um die Funktionalität der erarbeiteten Konzepte anhand einer Open-Source-Test-Suite zu evaluieren. In Kapitel 6 folgt schließlich das Fazit und ein Ausblick auf mögliche Weiterentwicklungen.

2. Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen erläutert, die im Rest der Arbeit Verwendung finden. Anschließend werden einige Arbeiten diskutiert, in denen relevante Konzepte behandelt wurden.

2.1. Statische Code-Analyse

Statische Analyse bezeichnet das Analysieren von Code ohne ihn auszuführen [1]. Ein primärer Einsatzzweck ist in Compilern, wo sie zur Optimierung oder Fehlerkennung genutzt wird. Mit ihr können eine Vielzahl Probleme erkannt werden, ohne, dass spezielle Sollwerte definiert werden müssen. Der große Vorteil liegt dabei in der Automatisierung und einfachen Wiederholbarkeit. Meistens wird die Analyse direkt am Programmcode durchgeführt, in einigen Fällen jedoch auch mit dem kompilierten Code. Im Folgenden wird nur auf die Analyse am Programmcode eingegangen. Der Programmcode besteht im Wesentlichen aus zwei Elementen: Statements und Ausdrücke. Statements sind die eigentlichen Befehle, wie While-Schleifen oder If-Abfragen. Ausdrücke sind Teil eines Statements, wie etwa Bedingungen der Form `a != null`.

Analysen werden im Allgemeinen an einem *Abstract Syntax Tree*, kurz *AST*, durchgeführt [3]. Dieser entspricht einer standardisierten Version des Source-Codes und wird aus dem Parse-Tree erzeugt, der durch die Abarbeitung der Grammatik der Programmiersprache entsteht. Die primären Unterschiede zwischen den beiden Repräsentationen liegen darin, dass in einem Parse-Tree jeder interne Knoten eine Produktion der Grammatik repräsentiert, während interne Knoten eines ASTs Programmkonstrukte darstellen [4]. Abbildung 2.1 zeigt, wie ein AST aussehen könnte. Das dritte Kind des Wurzelknotens entspricht dabei dem Else-Zweig des If-Statements, das in diesem Fall leer ist. Insbesondere komplexe Analysen werden durch den AST vereinfacht.

Es gibt eine Vielzahl von Techniken, die in der statischen Analyse Anwendung finden. Im simpelsten Fall wird nur der AST betrachtet, etwa um die Verwendung einer bestimmten Funktion als Fehler anzumerken. Komplexe Analysen bilden Modelle, um nachzuvollziehen, wie das Programm ausgeführt wird. Kontrollflussgraphen werden verwendet, um die möglichen Kontrollflüsse durch eine Funktion zu modellieren. Mit dem Kontrollflussgraphen können etwa Schleifen erkannt werden (auch nicht-triviale, die über `goto` zustande kommen). Eine Datenfluss-Analyse wird genutzt, um zu bestimmen, wie Daten durch das Programm oder die Funktion gereicht werden. Dieses Wissen kann dazu genutzt werden, mögliche Wertebereiche für die Variablen zu bestimmen. Damit lassen sich *Constraints* (Einschränkungen) formulieren, mit denen beispielsweise Gleichungen oder Bedingungen teilweise ausgewertet werden können.

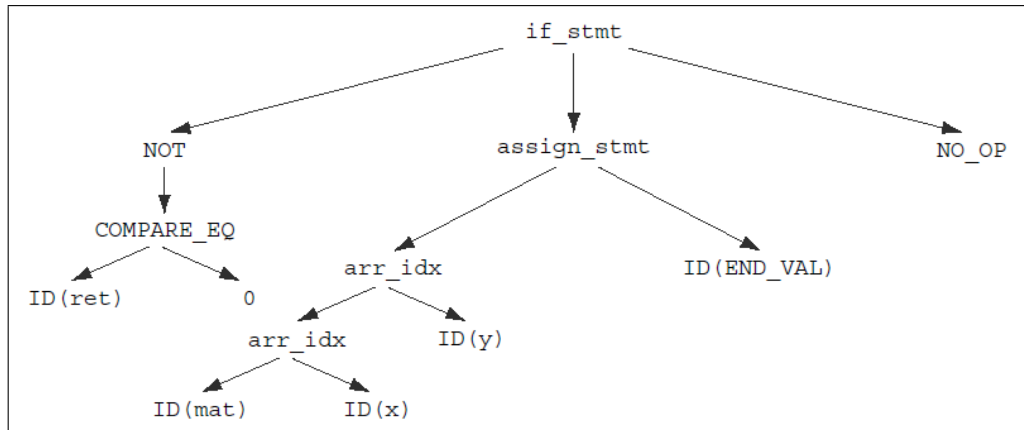

Abbildung 2.1.: AST eines simplen C-Programms (aus [3], Seite 75)

Eine Analyse ist intraprozedural falls sie nur innerhalb einer Funktion stattfindet, und nicht über ihre Grenzen hinaus. Das Gegenstück dazu ist die interprozedurale Analyse, welche auch über mehrere Funktionen analysiert.

Eines der ersten Programme, mit denen sich statische Analysen zur Fehlererkennung durchführen ließen, war lint für C-Programme [5]. Seitdem wurden zahlreiche weitere Programme für die verschiedensten Sprachen entwickelt. Dazu zählen etwa FindBugs [6] und PMD [7] für Java und CLang [8] für C++ und Objective-C. Ein Teil von ihnen ist durch benutzerdefinierte Regeln erweiterbar. Mit diesen Regeln können eigene Tests eingebaut werden, um Fehler zu finden, die das Programm nicht von selbst unterstützt. Im Allgemeinen ist dafür ein nicht-trivialer Programmieraufwand nötig. Die erweiterbaren Programme werden im Folgenden als Analyse-Frameworks bezeichnet. Ein Fehler, der durch eine statische Analyse gefunden wird, lässt sich einem von vier Typen zuordnen.

- Ein *True Positive (TP)* ist ein Fehler, der korrekt als solcher erkannt wurde
- Ein *False Positive (FP)* ist eine legitime Verwendung, die inkorrekt als Fehler erkannt wurde
- Ein *False Negative (FN)* ist ein Fehler, der nicht als solcher erkannt wurde
- Ein *True Negative (TN)* ist eine legitime Verwendung die nicht als Fehler erkannt wurde

Eine vollständig korrekte statische Analyse ist nicht möglich [9], wie sich durch Reduktion auf das Halteproblem zeigen lässt. Könnte ein Programm in jedem Fall korrekt entscheiden, ob eine Fehlerverletzung vorliegt, so ließe sich dieses Programm nutzen, um das Halteproblem zu lösen. Dies ist aber bewiesenermaßen unentscheidbar. Deshalb ist immer ein Kompromiss zwischen FPs und FNs nötig. Ein Programm, was alle möglichen Verstöße ausgibt, wird im Allgemeinen eine hohe Anzahl FPs verursachen, was den Nutzen einschränkt, da jeder einzelne Fund verifiziert werden muss. Wenn jedoch zu viele Verstöße ausgefiltert werden kann es zu FNs kommen, ein Fehler der nicht gefunden wurde. Dies ist potentiell noch schädlicher, als eine Vielzahl FPs, lässt

sich jedoch nicht immer ausschließen.

2.1.1. Tainted Object Propagation

Ein wichtiges Teilgebiet der statischen Analyse ist das *Tainted Object Propagation Problem*, welches unter der Datenfluss-Analyse einzuordnen ist. In [10] werden folgende Bestandteile des Problems definiert:

- Eine Menge von Source-Deskriptoren, die angeben, auf welche Art User-Daten in das Programm gelangen können
- Eine Menge von Sink-Deskriptoren, die unsichere Wege angeben, auf die Daten von einem Programm benutzt werden können
- Eine Menge von Derivation-Deskriptoren, die angeben, wie Daten durch das Programm propagiert werden können

Alle Daten, die von außerhalb des Programms kommen sind nicht vertrauenswürdig und könnten potentiell manipuliert worden sein. Ein Objekt, was aus einem Source-Deskriptor abgeleitet wird, gilt deshalb als *tainted* („*verdorben*“). Das Gegenteil davon ist eine vertrauenswürdige Variable, die als *untainted* gilt. Die Derivation-Deskriptoren definieren Funktionen, mit denen dieser Status auf andere Variablen übertragen werden kann. Dies wird auch als Propagierung bezeichnet. Eine Sicherheitsverletzung liegt nun vor, falls durch beliebig häufige Anwendung von Derivation-Deskriptoren das Ergebnis eines Source-Deskriptors bis zu einer Verwendung an einem Sink-Deskriptor propagiert werden kann. Anders ausgedrückt: Falls es einen Fluss von einer Source zu einem Sink gibt, auf dem der tainted-Zustand propagiert wird. Das Gegenstück eines Derivation-Deskriptors ist ein Sanitizer. Er setzt eine Variable explizit auf untainted.

Eine sehr relevante Ausprägung dieses Problems ist die sogenannte *SQL-Injektion*. Sie ist möglich, falls ein Programm SQL-Querys zusammensetzt, ohne die Benutzereingaben ausreichend zu verifizieren. Zum Zeitpunkt der Verfassung war SQL-Injektion als gefährlichster Software-Fehler in der Common Weakness Enumeration gelistet [11]. Listing 2.1 zeigt ein simples Beispiel, mit dem eine SQL-Injektion möglich ist. Zunächst wird ein String in die Variable `name` eingelesen, und mit diesem ein SQL-Query formuliert. Dies geschieht über eine einfache String-Konkatenation. Das entstehende Query wird schließlich ausgeführt. Der Fehler liegt nun darin, dass die Eingaben ungeprüft und unverändert in das Query eingebaut werden. `name` könnte beispielsweise den Wert `"Tom' OR 'x'='x"` haben. Eingesetzt in das SQL-Query ergibt dies `"... WHERE name='Tom' OR 'x'='x'"`, eine WHERE-Klausel, die unabhängig vom Namen immer erfüllt ist. Somit ist es möglich, potentiell kritische Daten auszulesen, wenn man die Form des Querys erraten kann.

```
1 String name = getUserData();
2 String query = "SELECT bankAccount FROM clients WHERE name='"
     + name +"'";
3 Set result = executeQuery(query);
```

Listing 2.1: Beispiel für SQL-Injektion

2.2. Association-Rule-Mining

Im späteren Verlauf der Arbeit wird es nötig sein, aus einer Menge von Beispielen zu lernen, indem Gemeinsamkeiten und Unterschiede der Beispiele extrahiert werden. Dazu wird das *Association Rule Mining* benutzt, welches ein Teilgebiet des *Data-Minings* ist. Nach [12] bezeichnet Data-Mining *„The process of nontrivial extraction of implicit, previously unknown and potentially useful information from data“*, d.h. den Prozess der Extrahierung potentiell nützlicher Information, die jedoch bis dahin unbekannt sind und nur implizit in der Datenmenge vorliegen. Eine der frühesten Anwendungen des Data-Minings war das Analysieren von Kaufverhalten, der sogenannten *Shopping Basket Analysis* [13]. Es wurde dazu genutzt, Artikel zu finden, die von Kunden oft zusammen gekauft werden. Mit diesen Informationen wurde dann das Sortiment und die Anordnung der Artikel im Laden optimiert. Das Vorgehen wird im Folgenden erklärt [14].

Association-Rule-Mining arbeitet mit sogenannten *Items*, die im allgemeinen numerische Werte sind. Sei $I = \{i_1, i_2, ..., i_n\}$ eine Menge solcher Items. Ein *Itemset* ist eine nicht-leere Teilmenge von I. Eine *Transaktion* ist ein Itemset, aus dem Informationen gewonnen werden sollen. In obigem Beispiel wäre jeder Einkauf eines Kunden eine Transaktion. Eine Menge von Transaktionen wird als *Transaktionsdatenbank* bezeichnet. Ein Itemset X ist in einer Transaktion *enthalten*, falls es eine Teilmenge der Transaktion ist. Es ist wichtig zu wissen, in wie vielen Transaktionen ein Itemset enthalten ist. Dieser Wert wird als *Support* eines Itemsets X bezeichnet und mit $sup(X)$ angegeben.

Nun ist es interessant zu wissen, welche Items oft zusammen auftreten. Das Finden solcher *Frequent Itemsets* wird als *Frequent Itemset Mining* bezeichnet. Gesteuert wird das Vorgehen über einen Parameter, der den minimalen Support angibt, den ein Frequent Itemset aufweisen muss.

Als *Association Rule* bezeichnet man eine Implikation $X \Rightarrow Y$ zwischen zwei Itemsets X und Y, die beide echte, nicht-leere Teilmengen von I sind. Sie besteht aus der Antezedens X und der Konsequenz Y. Weiterhin wird gefordert, dass X und Y disjunkt sind.

Der Support einer Regel, $sup(X \Rightarrow Y)$, ist definiert als $sup(X \cup Y)$ und gibt an, in wie vielen Transaktion die Regel gültig ist. Die *Confidence* einer Regel gibt an, wie viele Transaktionen, die X enthalten, auch Y enthalten. Anders ausgedrückt gibt die Confidence die Wahrscheinlichkeit an, dass die Implikation wahr ist, falls die Antezedens in

der Transaktion enthalten ist. Sie kann als Maß für die Signifikanz der Regel genutzt werden. Formal ausgedrückt gilt:

$$conf(X \Rightarrow Y) = \frac{sup(X \cup Y)}{sup(X)}$$

Das Finden aller geltenden Association Rules in einer Transaktionsdatenbank wird als Association Rule Mining bezeichnet. Als Eingabe werden vorher berechnete Frequent Itemsets genutzt. Gesteuert wird es über einen Parameter, der die minimale Confidence einer Regel angibt.

Die Association Rules enthalten das vorher beschriebene implizite Wissen der Datenmenge. Ein häufiges Problem ist, dass die Regelmenge sehr groß ist und viele triviale Regeln enthält. Das verringert die Effizienz der Algorithmen und erfordert einen großen Benutzeraufwand. In [15] wurde die Nutzung *geschlossener* (*closed*) Itemsets und Association Rules eingeführt. Ein Itemset X ist closed, falls kein Itemset X' existiert sodass $X \subset X'$ und $sup(X) = sup(X')$. Ein *Frequent Closed Itemset* ist ein Closed Itemset, welches die Anforderungen eines Frequent Itemsets erfüllt. Für Association Rules auf Frequent Closed Itemsets muss zusätzlich gelten, dass sowohl X als auch $X \cup Y$ Frequent Closed Itemsets sind und, dass kein Frequent Closed Itemset Z existiert, für das $X \subset Z \subset (X \cup Y)$ gilt [14]. Das Finden von Closed Association Rules wird als *Closed Association Rule Mining* bezeichnet.

Durch die zusätzlichen Anforderungen wird die Menge an Association Rules drastisch reduziert. Dies geschieht jedoch nicht auf Kosten der Genauigkeit, da sich alle Regeln, die mit diesem Vorgehen wegfallen, in trivialer Weise aus den erstellten Regeln ableiten lassen [14][15]. Das Closed Association Rule Mining ist besonders effizient, wenn es eine große Korrelation in den Daten gibt [15].

2.3. Verwandte Arbeiten

In diesem Abschnitt werden einige Arbeiten diskutiert, die für diese Arbeit relevante Konzepte vorgestellt haben, welche über die besprochenen Grundlagen hinausgehen. Sie befassen sich nicht mit der Ableitung von Regeln aus Beispielcode, jedoch sind Teile der folgenden Konzepte in diese Arbeit mit eingeflossen.

Ein interessanter Ansatz wird von PQL (*Program Query Language* [16]) verfolgt. PQL erlaubt es, Source-Code-Abfragen in einer Java-ähnlichen Syntax zu formulieren. Diese Abfrage führt es auf ein gegebenes Programm aus, um so fehlerhaften Code zu finden, welcher der Abfrage entspricht. Zu Beginn der Abfrage müssen die verwendeten Variablen deklariert werden. Innerhalb der Abfrage können Platzhalter verwendet werden, um an einer Stelle beliebige Werte zu erlauben. Es erlaubt Unterabfragen zu definieren. Diese können etwa dazu genutzt werden, um alle möglichen Propagierungen als einzelne Operation auszudrücken. Während der Analyse wird eine Belegung für die Variablen gesucht, mit der sich die ganze Abfrage erfüllen lässt. Die Abfragen werden mittels bddbddb (*BDD-Based Deductive DataBase* [17]) in ein Datalog-Programm übersetzt. Datalog ist eine logische Programmiersprache für Datenbankab-

fragen, ähnlich Prolog. bddbddb liest zunächst den Bytecode des zu analysierenden Programms ein und bildet eine Faktenmenge. Es wird nun versucht, eine Reihe von Fakten zu finden, mit denen die übersetzte Anfrage erfüllt werden kann. Die Ergebnisse dieser statischen Analyse sind allerdings flow-insensitiv, d.h. die Reihenfolge, in der die Befehle vom Programm ausgeführt werden, muss nicht der Reihenfolge in der ursprünglichen PQL-Abfrage entsprechen. Um hier Abhilfe zu schaffen unterstützt PQL noch eine dynamische Analyse. Hier werden die Ergebnisse der statischen Analyse genutzt, um den Bytecode des Programms um weitere Abfragen zu ergänzen. Falls nun während der Laufzeit eine Regelverletzung gefunden wird, wird eine benutzerdefinierte Aktion ausgeführt. PQL verbindet also die statische mit der dynamischen Analyse, indem die Ergebnisse der statischen Analyse genutzt werden, um Überprüfungen für die dynamische Analyse zu erstellen.

PR-Miner [18] setzt Closed Association Rules zur Analyse von Programmcode ein. Ziel war allerdings nicht, auf Basis externer Regeln zu prüfen, sondern Verletzungen impliziter Regeln innerhalb eines Programms zu finden. Implizite Regeln definiert es als gängige Konventionen, die innerhalb eines Programms genutzt werden, die jedoch von den Entwicklern nicht dokumentiert werden. Dieses Wissen wäre etwa für neue Entwickler überaus hilfreich. Darüber hinaus versucht PR-Miner, Verletzungen dieser Regeln innerhalb des Codes zu finden. Zunächst bildet es die einzelnen Funktionen des Codes auf Transaktionen ab. Dazu wird jedes Statement einer Funktion in einen Hash umgerechnet. Anschließend bildet es die Closed Association Rules dieser Transaktionsmenge. Aus den Ergebnissen werden die Regeln entfernt, die nur eine geringe Confidence haben. Bei solchen Regeln ist es unwahrscheinlich, dass sie eine Konvention innerhalb des Programms darstellen. Verletzungen werden nun daran erkannt, dass die Antezedens einer Regel einen höheren Support als die Konsequenz hat.

Splint [19] ist ein Analyse-Programm für C. Es wird durch Annotationen innerhalb des Codes gesteuert. Da C von sich aus keine Annotationen unterstützt, werden diese in Kommentarblöcke eingebettet. Der Code muss also auf die Anwendung von Splint ausgelegt sein. Wenn eine Variable etwa mit `/*@notnull@*/` annotiert ist, weiß Splint, dass diese Variable nie den Wert `NULL` haben sollte. Es gibt deshalb einen Fehler aus, falls es eine Verwendung erkennt, in der sie diesen Wert haben könnte. Annotationen können Attribute an eine Variable zuzuweisen, etwa, ob sie tainted ist. Splint kann darüber hinaus um benutzerdefinierte Annotationen erweitert werden. Es kann außerdem definiert werden, wie verschiedene Kombinationen von Attributen propagiert werden. Beispielsweise könnte gelten, dass die Kombination von tainted und untainted wieder tainted ergibt. Splint wird als iterativer Prozess beschrieben: Zunächst wird es ausgeführt, um Warnungen zu generieren. Anschließend wird entweder der Code angepasst oder um Annotationen ergänzt. Mit diesen neuen Erkenntnissen wird Splint nochmals ausgeführt, und der Prozess wiederholt, bis keine Warnungen mehr vorliegen.

3. Konzepte

In diesem Kapitel werden die erarbeiteten theoretischen Konzepte vorgestellt. Es ist wie folgt aufgeteilt: Zunächst werden in Abschnitt 3.1 die Anforderungen an die zu entwickelnde Methode geklärt und ein grober Überblick über die Umsetzung gegeben. Dort werden auch die einzelnen Teilaspekte erklärt, die für die Methode zu betrachten sind. Diese werden anschließend in jeweils einem Abschnitt behandelt.

3.1. Anforderungen und Überblick

Zunächst gilt es, die genauen Anforderungen zu formulieren, welche die zu entwickelnde Methode erfüllen muss. Diese Anforderungen basieren auf der Annahme, dass die Methode später von Software-Entwicklern eingesetzt wird, die wenig bis keine Erfahrung mit der Funktionsweise statischer Analysen haben.

- Die Regeln sollen direkt aus Codebeispielen abgeleitet werden und mit einem Analyse-Framework nutzbar sein
- Es soll vom Benutzer kein Fachwissen über Methoden der statischen Analyse gefordert werden. Ebenso soll die Methode leicht erlernbar sein
- Die Methode soll geeignet sein, um Regeln für wichtige Sicherheitsprobleme wie SQL-Injektion zu formulieren
- Es soll möglich sein, bestehende Regeln zu verfeinern

Natürlich ist es nicht ohne weiteres möglich, aus einem Codestück eine Regel zu erstellen. Meist werden sich darin irrelevante Teilstücke finden, und die Bedeutung der einzelnen Teile kann nicht automatisiert abgeleitet werden. Um eine zuverlässige Regelerstellung zu gewährleisten sind noch weitere Informationen nötig, die der User zusätzlich zum Code beisteuern muss. Damit das so einfach und intuitiv wie möglich bleibt werden dafür Annotationen verwendet. Annotationen sind ein Feature verschiedener objektorientierter Programmiersprachen, unter anderem von Java und C# (dort werden sie als Attribute bezeichnet). Sie erlauben es, im Quelltext Meta-Informationen zu hinterlegen, die von anderen Programmen oder während der Laufzeit ausgelesen werden können. Ein gängiges Beispiel ist die `@Override`-Annotation, die häufig in Java-Code verwendet wird. Mit ihr kann man angeben, dass eine Methode zwingend eine vererbte Funktion überschreiben muss, sie also in einer der Oberklassen existieren muss.

Zunächst gilt es also, eine passende Annotationssprache zu konzipieren. Dies geschieht in Abschnitt 3.2. Unter Nutzung dieser Annotationssprache soll es dann mög-

lich sein, die notwendigen Zusatzinformationen für die Erstellung neuer Regeln abzuleiten. Wie sich zeigen wird sind dazu teils mehrere Teilregeln nötig, die im Folgenden zur Unterscheidung von der Oberregel *Vorschriften* genannt werden. Die detaillierte Erklärung erfolgt in Abschnitt 3.3. Nun soll es auch möglich sein, diese Regeln weiter zu verfeinern. Da die Regeln direkt aus Beispielcode abgeleitet werden sind sie zunächst sehr speziell, oftmals ist jedoch eine Verallgemeinerung möglich. Beispielsweise gibt es viele verschiedene Klassen, mit denen man Daten aus einer Datei lesen kann. Sie machen im Grunde alle das gleiche, nur auf andere Art und Weise. Es wäre also ausreichend, einfach eine allgemeine Dateiklasse zu referenzieren. Eine genaue Erläuterung der Problematik und des Vorgehens dazu folgt in Abschnitt 3.4. Zuletzt müssen die gewonnenen Regeln natürlich auch in ein Analyse-Framework integriert werden, wofür Abschnitt 3.5 ein einfaches Analyse-Verfahren vorschlägt.

Im Rahmen dieser Arbeit werden zwei verschiedene Regeltypen unterstützt, strukturelle Regeln und Datenfluss-Regeln. Mit strukturellen Regeln wird versucht, den Regelcode innerhalb des Syntax-Baums des zu analysierenden Codes zu finden. In Datenfluss-Regeln hingegen wird nicht der Code als ganzes gesucht. Stattdessen werden Teilregeln, Vorschriften, extrahiert, die Sources, Sinks und Sanitizer beschreiben. Anders als in [10] werden keine Propagierungen extrahiert, da ein FP durch eine ungewünschte Propagierung einem FN durch eine vergessenen Propagierung vorgezogen wird. Propagierungen finden implizit statt. Dabei werden neben den Elementen selbst auch ihre Abhängigkeiten extrahiert. So muss ein Sink etwa erst erstellt werden, bevor er sicherheitsrelevant ist. Dadurch werden FPs vermieden, die durch die reine Konzentration auf Namensgleichheit der Methoden entstehen können. In der Analyse werden alle extrahierten Vorschriften parallel gesucht und genutzt, um den Status der verwendeten Variablen zu bestimmen. Ein Code-Beispiel für Datenfluss-Regeln dient also nur dazu, Wissen über diese drei Elemente zu sammeln. Wie genau sie im Code zusammenspielen ist nicht relevant.

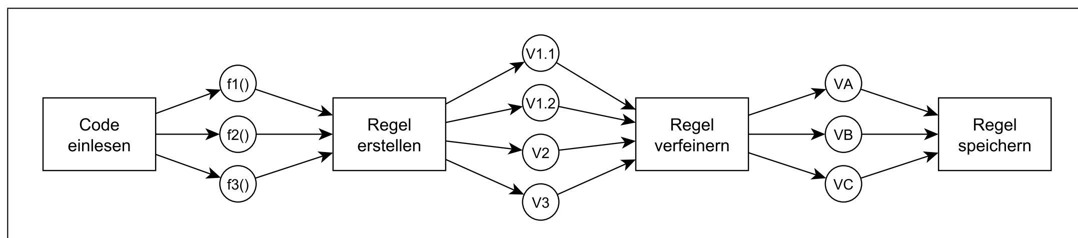

Abbildung 3.1.: Schritte der Regelableitung

Abbildung 3.1 zeigt das geplante Vorgehen. Zunächst wird der Code eingelesen. Aus den einzelnen Funktionen werden die Vorschriften erstellt. Diese Vorschriften werden verallgemeinert, wobei im günstigsten Fall einige Vorschriften wegfallen. Das Ergebnis wird schließlich gespeichert, damit es später in einem Analyse-Framework genutzt werden kann.

Es wurden einige Einschränkungen bei den folgenden Methoden gemacht, welche in den einzelnen Abschnitten noch weiter erläutert werden:

```
String data = "";

Socket socket = new Socket("host.example.org", 39544);
InputStreamReader input = new InputStreamReader(socket.
  getInputStream(), "UTF-8");
BufferedReader reader = new BufferedReader(input);

data = reader.readLine();

Connection conection = IO.getDBConnection();
Statement statement = conection.createStatement();

String query = "insert into users (status) values "
  + "('updated') where name='"+data+"'";
Boolean result = statement.execute(query);

statement.close();
```

Listing 3.1: Beispiel-Code, aus dem eine Regel abgeleitet werden soll (aus der Juliet-Test-Suite [20], leicht angepasst)

- Es wird rein intraprozedural gearbeitet, weder die Regelerstellung noch die Analyse können Informationen zwischen Funktionen austauschen. Dadurch verringert sich die Komplexität des Vorgehens erheblich
- Bei der Regelverfeinerung wird nur ein einzelner Fluss durch die Funktion betrachtet, d.h. dass im Allgemeinen ein if/then/else-Konstrukt nicht unterstützt wird. Auch dies wird primär durch die Komplexität verursacht
- Es wird davon ausgegangen, dass Sources und Sanitizer immer über den Rückgabewert arbeiten. Dies dient der einfacheren Benutzung der Annotationssprache

Bevor die Ansätze vorgestellt werden, soll an dieser Stelle ein etwas umfangreicheres Beispiel erläutert werden. Es wird innerhalb dieses Kapitels benutzt, um die Konzepte zu demonstrieren. Listing 3.1 zeigt ein Code-Beispiel, in dem eine SQL-Injektion möglich ist. Zeile 8 stellt die Source dar. Daten werden von einem Socket gelesen, d.h. aus dem Netzwerk. Da man dieser Quelle nicht vertrauen kann ist die Variable `data` als tainted anzusehen. Dieser Zustand wird in Zeile 13 in die Variable `query` propagiert. In Zeile 15 wird ein SQL-Query ausgeführt, dieses Statement ist also ein Sink. Das Query ist Ergebnis einer String-Konkatenation in der `data` verwendet wurde. Da `data` vorher nicht überprüft wurde stellt dieser Code ein Sicherheitsrisiko dar.

Annotation	Typ	Beschreibung
`@observe`	Allgemein	Hebt für die Regel relevante Variable hervor
`@dirty`	Datenfluss	Markiert Statement als Source
`@clean`	Datenfluss	Markiert Statement als Sanitizer
`@critical`	Datenfluss	Markiert Statement als Sink
`@required`	Struktur	Das Statement muss vorkommen, falls die Befehlsfolge bis hierhin gefunden wurde
`@match`	Allgemein	Konstante Parameter müssen übereinstimmen
`@redundant`	Allgemein	Statement ist für die Regel nicht relevant

Tabelle 3.1.: Verwendete Annotationen

3.2. Annotationssprache

Im Folgenden werden die verwendeten Annotationen nacheinander vorgestellt und begründet. Außerdem wird erläutert, welche Bedeutung sie für die Regelerstellung und Analyse haben. Um den Punkt der einfachen Erlernbarkeit zu erfüllen wurde Wert darauf gelegt, den Umfang der Annotationssprache möglichst klein zu halten. Eine zu umfangreiche Annotationssprache würde diesem Ziel widersprechen. Ebenso wurde vermieden, Fachbegriffe für die Annotationsnamen zu verwenden, da nicht zu erwarten ist, dass ein Benutzer diese dem Einsatzzweck zuordnen kann.

Ein unannotiertes Code-Beispiel hat zunächst keinerlei Funktion. Die Regelerstellung wird strikt über die verwendeten Annotationen gesteuert, sowie die Abhängigkeiten, die sich dadurch ergeben. Der Grundsatz lautet deshalb, dass der Benutzer explizit sein Vorhaben angeben muss und keine impliziten Annahmen gemacht werden, die sich nicht direkt aus den Annotationen ableiten lassen.

Die Annotationen gliedern sich in drei Gruppen: Allgemein, Struktur und Datenfluss. Allgemeine Annotationen können mit allen Regeltypen verwendet werden, strukturelle Annotationen nur mit strukturellen Regeln und Datenfluss-Annotationen nur mit Datenfluss-Regeln. Die verwendeten Annotationen und ihre Zugehörigkeiten sind in Tabelle 3.1 dargestellt.

Allgemeine Annotationen

Um Variablen als relevant hervorzuheben wird `@observe` benutzt. Einige der folgenden Annotationen implizieren diesen Status auf die verwendeten Variablen. Ebenso werden automatisch Variablen observiert, von denen andere observierte Variablen abhängen. Die Details dazu werden in Abschnitt 3.3.1 erläutert. Variablen, die nicht relevant sind, werden während der Regelerstellung nicht berücksichtigt und demnach entfernt. Dadurch ist es nicht erforderlich, dass der Benutzer irrelevanten Code entfernt, was unter Umständen ein großer Aufwand und fehleranfällig wäre.

Alle Statements, die Referenzen zu relevanten Variablen enthalten, werden beibehalten. Sollte ein solches Statement jedoch für die Erkennung nicht notwendig sein, aber

Parameter	Funktion
full	Die Zeichenkette muss exakt gefunden werden (Standard)
contains	Die Zeichenkette muss enthalten sein
starts	Die Zeichenkette muss am Beginn gefunden werden

Tabelle 3.2.: Parameter von `@match`

```
1 InputStream fis = new FileInputStream("userfile.txt");
2 BufferedReader br = new BufferedReader(
3   new InputStreamReader(fis,Charset.forName("UTF-8")));
4
5 @required
6 br.close();
```

Listing 3.2: Beispiel für `@required`

der Benutzer es aus irgendwelchen Gründen nicht entfernen wollen, so kann es mit `@redundant` annotiert werden. Statements, die mit `@redundant` annotiert sind, werden bei der Regelerstellung nicht berücksichtigt.

Konstante Parameter (sog. *Literals*) werden standardmäßig ignoriert. Das lässt sich damit begründen, dass es zu einschränkend wäre, nur nach einem bestimmten Literal zu suchen. Der Socket in Listing 3.1 wäre beispielsweise auch mit anderen Adressen und Ports verdächtig. Die Relevanz der gegebenen Literals kann mit der `@match`-Annotation für beliebige Statements angegeben werden. `@match` kann darüber hinaus parametrisiert werden. Die Parameter sind nur für String-Literals relevant, für die sie steuern, auf welche Art der Text während der Analyse verglichen werden soll. Tabelle 3.2 enthält eine Auflistung der möglichen Parameter. `@match` impliziert `@observe` auf das Statement und die oberste vom Statement verwendete Variable.

Strukturelle Annotationen

Die Annotation `@required` ist nur für strukturelle Regeln erlaubt. Normalerweise erkennen strukturelle Regeln einen Defekt wenn der gesamte Regelcode gefunden wurde. Falls die Regel eine `@required`-Annotation enthält, wird dieses Verhalten verändert. Wird in diesem Fall die Codesequenz vor dem mit `@required` annotiertem Statement gefunden, dann muss vor Funktionsende zwingend das annotierte Statement folgen, sonst liegt ein Defekt vor. In diesem Fall stellt der Regelcode also die korrekte Benutzung dar. Hiermit lassen sich Abhängigkeiten modellieren, die nicht mit Datenfluss-Regeln möglich sind. Listing 3.2 zeigt eine beispielhafte Verwendung dieser Annotation: Wenn ein `BufferedReader` erzeugt wird, dann muss er auch wieder geschlossen werden, sonst entsteht ein Ressource-Leak. `@required` impliziert `@observe` auf das Statement und die oberste vom Statement verwendete Variable.

Datenfluss-Annotationen

Mit `@dirty` kann in einer Datenfluss-Regel eine Quelle unvertraulicher Daten hervorgehoben werden. Es muss an einer Zuweisung benutzt werden. Die Abhängigkeiten des Statements werden automatisch ermittelt. Wenn die Befehlsfolge während der Analyse gefunden wird, dann wird das Ziel der Zuweisung auf tainted gesetzt.

Ein Sanitizer wird mit `@clean` hervorgehoben. Wie bei `@dirty` werden die Abhängigkeiten automatisch ermittelt. Ebenso ist auch `@clean` nur an Zuweisungen erlaubt. Während der Analyse wird das Ziel der Zuweisung auf untainted gesetzt. Dabei haben Sanitizer eine höhere Priorität als Sources und Propagierungen.

Sinks werden mit `@critical` annotiert. Falls während der Analyse eine Variable benutzt wird, die vorher auf tainted gesetzt wurde, so wird ein Defekt ausgegeben.

Mit der beschriebenen Annotationssprache lässt sich eine Vielzahl von sicherheitsrelevanten Regeln erstellen, wie später in Kapitel 5 demonstriert wird. Die Sprache ist zudem kompakt und verwendet simple, aber intuitive Namen für die Annotationen, weshalb sie die gesetzten Ziele erfüllt.

Beispiel

Zuletzt soll der in Abschnitt 3.1 beschriebene Beispielcode in eine annotierte Version überführt werden. Das Ergebnis davon ist in Listing 3.3 zu sehen, worauf sich auch die folgenden Zeilennummern beziehen. Zeile 10 ist wie beschrieben eine Source, sodass sie mit `@dirty` annotiert werden muss. Zeile 19 stellt einen Sink dar, welcher mit `@critical` markiert wird. Die Propagierung in Zeile 15 muss nicht extra annotiert werden, da sie sich implizit aus dem Code ergibt. Wäre die Variable `query` jedoch unabhängig vom Status der Variable `data` untainted, dann müsste die Zeile mit `@clean` annotiert werden. In Zeile 2 wird `data` ein Wert zugewiesen. Dieser wird jedoch nie benutzt, wodurch das Statement überflüssig ist. Es kann somit mit `@redundant` annotiert werden, und wird in der Regel nicht weiter beachtet. Diese drei Annotationen reichen aus, um den Code in eine funktionierende Regel zu überführen.

3.3. Regelerstellung

Die Eingabe der Regelerstellung ist eine geeignete Repräsentation des Beispielcodes in Form einer Baumstruktur. Die Erstellung erfolgt in mehreren Schritten. Zuerst werden in Abschnitt 3.3.1 die Abhängigkeiten berechnet, die sich aus dem Code ergeben. In den Abschnitten 3.3.2 und 3.3.3 folgt schließlich die eigentliche Erstellung der strukturellen bzw. Datenfluss-Vorschriften. Das Ergebnis der Regelerstellung ist eine Regel, die aus einer Menge von Vorschriften besteht.

```
1  @redundant
2  String data = "";
3
4  Socket socket = new Socket("host.example.org", 39544);
5  InputStreamReader input = new InputStreamReader(socket.
6    getInputStream(), "UTF-8");
7  BufferedReader reader = new BufferedReader(input);
8
9  @dirty
10 data = reader.readLine();
11
12 Connection conection = IO.getDBConnection();
13 Statement statement = conection.createStatement();
14
15 String query = "insert into users (status) values "
16   + "('updated') where name='"+data+"'";
17
18 @critical
19 Boolean result = statement.execute(query);
20
21 statement.close();
```

Listing 3.3: Annotierte Version von Listing 3.1

3.3.1. Bestimmung der Abhängigkeiten

Für die Regelerstellung ist es notwendig, die Abhängigkeiten innerhalb des Eingabecodes zu bestimmen. Diese werden dazu genutzt, um die Statements zu berechnen, die zu einer Vorschrift gehören. In diesem Abschnitt werden Verfahren vorgestellt, mit denen sie sich im nötigen Umfang berechnen lassen. Es gibt zwei verschiedene Abhängigkeiten, die bestimmt werden müssen: Als erstes wird bestimmt, in welcher Reihenfolge die Statements ausgeführt werden können. Danach folgen die Abhängigkeiten der Variablen untereinander.

Vorgängergraph

Der Vorgängergraph wird später dazu genutzt, um zu bestimmen, welche Statements vor einem gegebenen Statement ausgeführt werden können. Zum Beispiel wird vor einem Else-Block der zur selben Abfrage zugehörige If-Block nicht ausgeführt, und hat deshalb für den Else-Block keine Relevanz. Er wird wie folgt konstruiert: Es wird ein gerichteter Graph erstellt, und für jedes Statement ein Knoten in diesem Graph. Danach wird der Code sequentiell durchgegangen und jedes Statement mit allen Statements verbunden, die direkt vor ihm ausgeführt werden können. Sprünge zurück zum Start von Schleifen werden dabei nicht berücksichtigt, wodurch keine Zyklen entstehen kön-

nen. Der Vorgängergraph ähnelt dem Kontrollflussgraphen, die Unterschiede liegen primär darin, dass er schleifen- und zyklenfrei ist, sowie der Richtung der Kanten. Algorithmus 1 beschreibt das genaue Vorgehen.

Algorithmus 1 : Bestimmung des Vorgängergraphen

Input : Liste $statements$ der im Code vorkommenden Statements
Result : Vorgängergraph $G = (V, E)$

Erstelle Graphen G und füge für jedes $s \in statements$ einen Knoten mit Namen s hinzu.
foreach $s \in statements$ **do**
  Sei $next$ die Menge der Statements, die direkt nach s ausgeführt werden können.
  foreach $n \in next$ **do**
    if *n ist kein Vorgänger von s* **then**
      Erstelle Kante von n nach s.
    end
  end
end

return G

Definition 3.1 *Ein Statement s ist Vorgänger eines Statements t gdw. im Vorgängergraph ein Pfad von t nach s existiert oder falls $t = s$. Das Statement s ist direkter Vorgänger von t gdw. im Vorgängergraph eine Kante von t nach s existiert. Entsprechend ist t Nachfolger von s falls s ein Vorgänger von t ist und $s \neq t$.*

Falls ein Statement s aus dem Graphen entfernt wird, werden die eingehenden Kanten mit den Zielknoten der ausgehenden Kanten verbunden und umgekehrt. Der Fall $t = s$ dient der einfacheren Behandlung in späteren Abschnitten.

Um die Folgestatements zu berechnen ist eine rudimentäre Kontrollfluss-Analyse des Codes notwendig. Die genaue Berechnung ist vom Typ des Statements abhängig, ergibt sich jedoch aus der Definition der Programmiersprache. Im Folgenden einige Beispiele:

- Ein Switch-Statement ist direkter Vorgänger aller ersten Statements der enthaltenen Case-Blöcke und des Default-Blocks, soweit vorhanden
- Ein If-Statement ist direkter Vorgänger vom If-Block. Es ist ebenfalls direkter Vorgänger des Else-Blocks, falls vorhanden. Falls es keinen Else-Block gibt ist das If-Statement direkter Vorgänger des ersten Statements nach dem If-Block
- Ein Expression-Statement ist direkter Vorgänger des nachfolgenden Statements

Der Beispiel-Code aus Listing 3.3 eignet sich nicht zur Demonstration. Es gibt keinerlei Verzweigungen oder vergleichbares, weshalb jedes Statement der direkte Vorgänger des nachfolgenden Statements ist. Die Durchführung wird stattdessen an Listing 3.4 demonstriert.

```
 1  boolean f = a();
 2  while (f) {
 3    int b = c();
 4    if (b < 0)
 5      break;
 6
 7    switch (b) {
 8    case 5:
 9      f = false;
10    default:
11      break;
12    }
13  }
14  d();
```

Listing 3.4: Beispielcode für den Vorgängergraphen

Zeile	Wert von next	Neue Kanten
1	{ 2 }	2 → 1
2	{ 3, 14 }	3 → 2, 14 → 2
3	{ 4 }	4 → 3
4	{ 5, 7 }	5 → 4, 7 → 4
5	{ 14 }	14 → 5
7	{ 8, 10 }	8 → 7, 10 → 7
8	{ 9 }	9 → 8
9	{ 11 }	11 → 9
10	{ 11 }	11 → 10
11	{ 2, 14 }	14 → 11
14	∅	-

Tabelle 3.3.: Durchführung auf Listing 3.4

Tabelle 3.3 zeigt die Ausführung des Algorithmus. Für jedes Statement wird stellvertretend die Zeilennummer verwendet. Das Statment in Zeile 1 ist ein einfacher Ausdruck, weshalb es der direkte Vorgänger des nachfolgenden Statements ist. Die While-Schleife in Zeile 2 kann ausgeführt werden (Nachfolger in Zeile 3) oder nicht (Nachfolger in Zeile 14). Der Nachfolger des break-Statements in Zeile 5 ist ebenfalls Zeile 14, da der Rest der Schleife in diesem Fall nicht mehr ausgeführt wird. Das Switch-Statement in Zeile 7 kann einen von zwei Fällen annehmen, welche deshalb beide Nachfolger sind. Nach Zeile 9 wird Zeile 11 ausgeführt, da default nur als Sprungmarke für das Switch-Statement dient. Im Anschluss an Zeile 11 kann entweder die Schleife fortgeführt werden oder abgebrochen werden. Da das While-Statement bereits Vorgänger ist wird deshalb nur Zeile 14 als Nachfolger eingetragen. Der vollständige Vorgängergraph ist in Abbildung 3.2 zu sehen. Man kann leicht erkennen, dass beispielsweise die Vorgänger von Zeile 5 der Menge $\{1, 2, 3, 4, 5\}$ entsprechen.

Abhängigkeitsgraph

Der Abhängigkeitsgraph gibt die Abhängigkeiten der Variablen untereinander an. Mit ihm kann bestimmt werden, welche Voraussetzungen für die Variablen und Statements erfüllt sein müssen. Dazu sind zunächst einige Definitionen nötig.

Definition 3.2 *$assign(v)$ sei die Menge der Statements, an denen der Variable v ein Wert zugewiesen wird.*

Definition 3.3 *$ref(n)$ sei die Menge der Variablen, die von Statement n referenziert werden, ausgenommen alle var für die gilt $n \in assign(var)$. Es gilt zum Beispiel: $ref(\texttt{value=a+b}) = \{a, b\}$.*

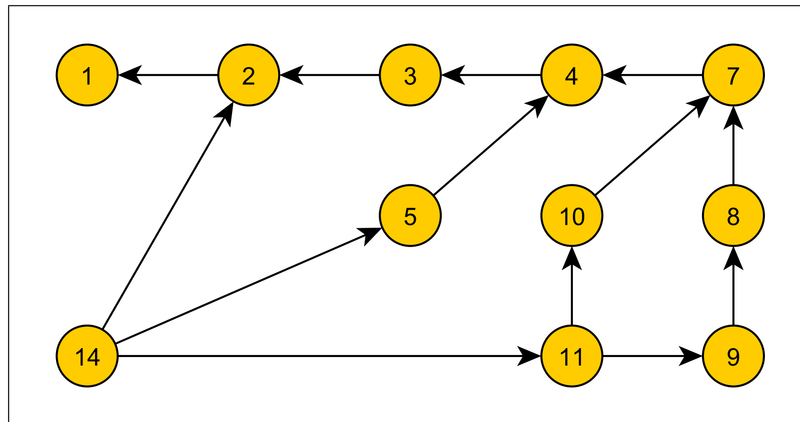

Abbildung 3.2.: Vorgängergraph von Listing 3.4

Wie beim Vorgängergraphen wird wieder ein gerichteter Graph erstellt. In diesem wird für jede im Code verwendete Variable ein Knoten erstellt. Anschließend wird nacheinander jedes Statement des Codes betrachtet. Falls ein Statement eine Zuweisung ist, wird zunächst die Variable $asgn$ bestimmt, der dort ein Wert zugewiesen wird. Danach wird eine Kante von $asgn$ zu jeder vom Statement referenzierten Variable erstellt. Algorithmus 2 gibt das Vorgehen in formalisierter Form an.

Algorithmus 2 : Bestimmung des Abhängigkeitsgraphen

Input : Liste $vars$ der im Code vorkommenden Variablen
Liste $statements$ der im Code vorkommenden Statements
Result : Abhängigskeitsgraph $G = (V, E)$

Erstelle Graphen G und füge für jedes $v \in vars$ einen Knoten mit Namen v hinzu.
foreach $s \in statements$ **do**
  if s *ist eine Zuweisung* **then**
    Sei $asgn$ die Variable, der in s ein Wert zugewiesen wird
    foreach $var \in ref(s)$ **do**
      Erstelle Kante von $asgn$ zu var
    end
  end
end

return G

Definition 3.4 *Eine Variable i ist abhängig von einer Variable j gdw. ein Pfad von i nach j im Abhängigkeitsgraph existiert. Sei $dependencies(v)$ die Menge der Variablen, von der die Variable v abhängt.*

Die Variable i ist also abhängig von allen Variablen, die innerhalb einer Zuweisung an i benutzt werden. Diese Abhängigkeiten werden später dazu benutzt, die Menge der relevanten Variablen einer Vorschrift zu bestimmen.

Zeile	Wert von asgn	ref(s)	Neue Kanten
4	socket	$\emptyset$	-
5	input	{ socket }	input $\rightarrow$ socket
7	reader	{ input }	reader $\rightarrow$ input
10	data	{ reader }	data $\rightarrow$ reader
12	connection	{ IO }	connection $\rightarrow$ IO
13	statement	{ connection }	statement $\rightarrow$ connection
15	query	{ data }	query $\rightarrow$ data
19	result	{ statement, query }	result $\rightarrow$ statement result $\rightarrow$ query
21	–	{ statement }	-

Tabelle 3.4.: Abhängigkeiten der Variablen von Listing 3.4

Zuletzt wird der Graph anhand des Beispiel-Codes auf Seite 15 demonstriert. Tabelle 3.4 zeigt die einzelnen Schritte der Ausführung des Algorithmus. Jede Tabellenzeile steht dabei für ein Statement des Codes. Das Statement der Zeile 2 ist mit `@redundant` annotiert, weshalb es für die Regel nicht berücksichtigt wird. In Zeile 5 wird ein neues InputStreamReader-Objekt erzeugt. Einer der Parameter des Konstruktors ist der Rückgabewert einer `socket`-Funktion. Somit wird eine Kante von `input` nach `socket` erstellt. Die Variable `conection` in Zeile 12 hängt von der statischen Variable `IO` ab. In Zeile 15 wird `query` das Ergebnis einer String-Konkatenation zugewiesen, in der `data` vorkommt. Somit hängt `query` von `data` ab und es wird eine Kante zwischen den beiden erstellt. In Zeile 19 hängt die Variable `result` gleich von zwei Variablen ab: Eine Funktion von `statement` wird mit `query` als Parameter aufgerufen. Somit werden Kanten von `result` nach `statement` und `query` erstellt. Zeile 21 enthält keine Zuweisung, weshalb es keine Abhängigkeiten gibt.

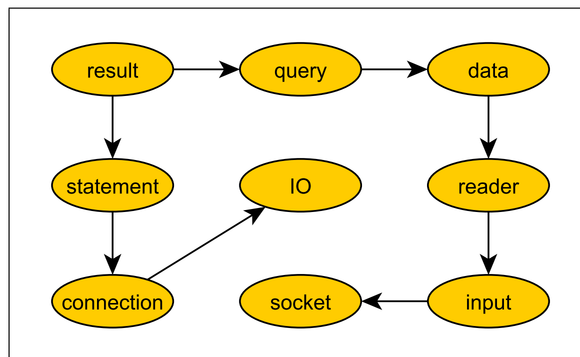

Abbildung 3.3.: Abhängigkeitsgraph von Listing 3.4

Der vollständige Abhängigskeitsgraph ist in Abbildung 3.3 zu sehen. Man kann leicht erkennen, dass beispielsweise $dependencies(data) = \{reader, input, socket\}$.

3.3.2. Strukturelle Vorschriften

Dieser Abschnitt befasst sich mit den Grundlagen der Regelerstellung und den strukturellen Vorschriften. Die Regelerstellung arbeitet mit einzelnen Funktionen, jede Funktion stellt ein Beispiel für die Regel dar. Die Funktionen werden in Vorschriften überführt und später verallgemeinert. Zunächst muss der Typ der Vorschrift bestimmt werden. Dieser ergibt sich aus den verwendeten Annotationen. Wenn eine Datenfluss-Annotation verwendet wird ist es eine Datenfluss-Vorschrift, ansonsten eine strukturelle.

Definition 3.5 *Eine Variable gilt als mit einer Annotation x annotiert falls sie diese Annotation aufweist oder der Variable in einer beliebigen Zuweisung ein Wert zugewiesen wird und das dazugehörige Statement mit x annotiert ist.*

Definition 3.6 *Ein Statment ist relevant, falls es mit `@observe` annotiert ist. Die Menge der relevanten Statements $relStats$ enthalte alle Statements, die relevant sind.*

Definition 3.7 *Eine Variable ist relevant, wenn sie mit `@observe` annotiert ist. Die Menge der relevanten Variablen $relVars$ enthalte alle Variablen, die relevant sind. Die erweiterte Menge der relevanten Variablen $relVars^+$ enthalte darüber hinaus alle Variablen, von denen relevante Variablen abhängen (diese kann durch Bestimmung der transitiven Hülle des Abhängigkeitsgraphen berechnet werden).*

Redundanzentfernung

Referenzen auf nicht relevante Variablen müssen entfernt werden, ebenso Statements, die sich dadurch erübrigen. Für die Redundanzentfernung ist die Bestimmung von $relVars^+$ notwendig. Dies sind alle Variablen, die vom späteren Code der Vorschrift benutzt werden. Mit ihnen können die Ausdrücke vereinfacht werden, indem alle Variablen entfernt werden, die nicht in $relVars^+$ vorkommen. Falls ein Statement nicht mit `@match` annotiert ist werden die Literals ebenfalls entfernt. Entfernte Teilausdrücke werden durch einen *Wildcard-Ausdruck* ersetzt, der als Platzhalter für einen beliebigen Ausdruck dient. Mehrere Wildcard-Ausdrücke können zusammengefasst werden. Zuletzt werden alle Statements aus dem Vorgängergraph entfernt, die nur Wildcard-Ausdrücke enthalten.

Erstellen der Vorschrift

Eine strukturelle Vorschrift lässt sich nun wie folgt bestimmen: Es werden alle Statements genommen, die Vorgänger eines Statements aus $relStats$ sind. Das Ergebnis ist die minimale Folge von Statements, die nötig ist, um die Vorschrift zu beschreiben. Diese ist nun zusammen mit dem Vorschrifts-Typ und der Menge der relevanten Variablen zu speichern, um sie später in der Analyse verwenden zu können. Wichtig

ist, dass auch die Annotationen mitgespeichert werden. Sie werden später von der Analyse benötigt.

3.3.3. Datenfluss-Vorschriften

Für Datenflussvorschriften sind noch weitere Berechnungen nötig. Anders als bei den strukturellen Vorschriften wird hier nicht eine Vorschrift aus dem Code erstellt, sondern beliebig viele. Dazu ist es nötig, die einzelnen annotierten Statements zu betrachten, und für jedes Statement, das mit Datenfluss-Annotation annotiert ist, eine Vorschrift abzuleiten.

Algorithmus 3 : Bestimmung der Tainted-Variablen

```
Input : Liste statements der im Code vorkommenden Statements
Result : Menge tainted der Tainted-Variablen

Erstelle leere Menge tainted.
foreach s ∈ statements do
    if s ist Zuweisung und mit @dirty annotiert then
        Sei asgn die Variable, der in s ein Wert zugewiesen wird.
        Füge asgn zu tainted hinzu.
    end
end

repeat
    foreach s ∈ statements do
        if s ist Zuweisung und nicht mit @clean oder @critical annotiert then
            Sei asgn die Variable, der in s ein Wert zugewiesen wird
            if ref(s) enthält min. ein v ∈ tainted then
                Füge asgn zu tainted hinzu.
            end
        end
    end
until Größe von tainted in aktueller Iteration unverändert;

return tainted
```

Für die korrekte Behandlung der Datenfluss-Vorschriften ist es nötig, die Propagierung der tainted-Variablen zu berechnen. Das geschieht wie folgt: Zunächst wird die Menge der Variablen bestimmt, denen in mit `@dirty` annotierten Zuweisungen ein Wert zugewiesen wird. Anschließend wird überprüft, welche Variablen von ihnen abhängen. Dazu wird $ref(s)$ für jede Zuweisung s berechnet und überprüft, ob eine der tainted-Variablen darin enthalten ist. Ausgenommen sind Statements, die als Sanitizer oder Sink ausgezeichnet sind. Das Ergebnis eines Sanitizers ist per Definition untainted und ein Sink hängt zwangsläufig von Variablen ab, die tainted sind. Das Ergebnis eines Sinks ist jedoch für die Erstellung der Vorschrift irrelevant, weshalb etwaige Propagierungen hier nicht beachtet werden. Da die nun gefundenen Variablen weitere

Propagierungen verursachen können wird der Vorgang solange wiederholt, bis in einer Iteration keine neuen Variablen gefunden wurden. Das Vorgehen ist in Algorithmus 3 formalisiert.

Das genaue Vorgehen unterscheidet sich leicht für Sources, Sinks und Sanitizer. Allen gemein ist, dass die Vorschriften unabhängig von den anderen erstellt werden. Dazu ist für jedes entsprechende Statement mit einer Kopie des Abhängigkeitsgraphen zu arbeiten, sodass lokale Änderungen an ihm vorgenommen werden können. Bei der Betrachtung eines Statements sind alle anderen Statements, die mit Datenfluss-Annotationen annotiert sind, aus dem Abhängigkeitsgraphen zu entfernen, damit sie keinen Einfluss ausüben können. Dies kommt daher, dass jede Vorschrift gesondert zu betrachten ist, und nicht direkt von den anderen abhängen soll. Wird dieser Schritt ausgelassen, dann hängen beispielsweise Sanitizer direkt von dem im Eingabecode verwendeten Source-Statement ab, da es zu seinen Abhängigkeiten zählen muss (da ein Sanitizer mit einer Variable arbeitet, die tainted ist). Dies widerspricht aber der klaren Trennung von Source, Sink und Sanitizer.

Das Vorgehen wird fortlaufend am Beispielcode auf Seite 15 demonstriert. Im Code gibt es zwei Datenfluss-Annotationen. Es werden also zwei Vorschriften aus diesem Code abgeleitet.

Source-Vorschrift

Mit Source-Vorschriften werden Variablen als tainted markiert. Daraus folgt, dass das dazugehörige Statement eine Zuweisung ist. Die Variable, der ein Wert zugewiesen wird, dient dabei als Platzhalter für die spätere Analyse und heiße im Folgenden $asgn$. Zunächst sind für $asgn$ die Abhängigkeiten zu berechnen, also $dependencies(asgn)$. Dies ergibt die für diese Vorschrift relevanten Variablen. Im nächsten Schritt werden wie im vorherigen Abschnitt die Redundanzen entfernt, basierend auf der eben berechneten Variablenmenge. Die Vorschrift besteht dann aus allen Statements, die Vorgänger des Source-Statements sind. Für die Analyse sind die relevanten Variablen zu vermerken, $asgn$ ist dabei besonders hervorzuheben, damit dies später genutzt werden kann, um den tainted-Wert richtig zu setzen.

Das Beispiel hat in Zeile 10 ein Source-Statement. Dort hat $asgn$ den Wert `data`. Wie bereits beschrieben ist $dependencies(data) = \{reader, input, socket\}$. Nun werden die Redundanzen auf Basis dieser Variablen entfernt. Das Ergebnis ist in Listing 3.5 zu sehen. Dabei steht _ für einen Wildcard-Ausdruck. Alle Statements nach dem Source-Statement fallen direkt weg, da sie keine der relevanten Variablen referenzieren. Ansonsten würden sie im nächsten Schritt entfernt werden, da sie keine Vorgänger sind. Dieser Schritt ist in diesem Fall wegen des linearen Kontrollflusses nicht nötig. Das Listing entspricht der finalen Source-Vorschrift.

```
1 Socket socket = new Socket(_,_);
2 InputStreamReader input = new InputStreamReader(socket.
3   getInputStream(), _);
4 BufferedReader reader = new BufferedReader(input);
5 
6 @dirty
7 data = reader.readLine();
```

Listing 3.5: Vereinfachte Source-Vorschrift

Sink-Vorschrift

In Sink-Vorschriften sind mehr Details zu beachten als in Source-Vorschriften. Ein Sink ist ein Statement, das eine Regelverletzung darstellt, wenn Variablen mit gesetztem tainted-Wert mit ihm verwendet werden können. Daraus folgt, dass mindestens eine der verwendeten Variablen im Eingabecode mit `@dirty` annotiert ist oder aus einer solchen Variable abgeleitet wurde. Für den Sink ist es jedoch egal, wie diese Variable entstanden ist. Hier zählt nur, dass es sie gibt. Die Abhängigkeiten, die das Sink-Statement mit solchen Variablen hat, sind deshalb bei der Redundanzentfernung zu vernachlässigen.

Dazu wird wie folgt vorgegangen: Sei s das Sink-Statement, $tainted$ die Menge der tainted-Variablen innerhalb des Eingabecodes (also das Ergebnis von Algorithmus 3). Zunächst wird $ref(s)$ bestimmt. Daraus werden alle Variablen entfernt, die in $tainted$ vorkommen. Die entstehende Menge entspreche im folgenden $relVars$. Basierend darauf wird $relVars^+$ bestimmt. Als nächstes werden damit die Redundanzen entfernt. Hierbei fallen alle Abhängigkeiten zu tainted-Variablen weg. Wichtig ist jedoch die Sonderstellung von s selbst. In s müssen die dort verwendeten tainted-Variablen *zwingend* erhalten bleiben, um in der Analyse als Platzhalter zu dienen. Für dieses Statement ist also die Schnittmenge von $ref(s)$ und $tainted$ zu $relVars^+$ hinzuzufügen. Zuletzt werden wie bei den Source-Vorschriften alle Statements in die Vorschrift aufgenommen, die Vorgänger von s sind.

Das Beispiel hat in Zeile 19 ein Sink-Statement. Zunächst müssen die tainted-Variablen in der Funktion berechnet werden. In Zeile 10 gilt `data` aufgrund seiner Annotation als tainted. Dieser Wert wird in Zeile 15 in die Variable `query` propagiert. Zeile 19 ist ein Sink und propagiert deshalb nicht. Es gilt also $tainted = \{data, query\}$. $ref(s)$ ist $\{statement, query\}$, aus dieser Menge wird $query$ entfernt, da es in $tainted$ ist. Die verbleibende Variable ist $statement$, für die gilt $dependencies(statement) = \{connection, IO\}$. Daraus ergibt sich $relVars^+ = \{statement, connection, IO\}$. Mit dieser Menge werden nun die Redundanzen entfernt. Alle Statements vor Zeile 12 fallen dabei weg, da dort keine dieser Variablen referenziert wird. Für das Sink-Statement ist zusätzlich noch die Variable $query$ mit in die Menge aufzunehmen, weshalb der Verweis auf die Variable $query$ dort nicht entfernt wird. Sehr wohl entfernt wird aber die Zuweisung an die Variable $query$, da dort keine relevanten Variablen verwendet werden. Zuletzt werden die Statements auf die Menge der Statements reduziert, die

```
1 Connection conection = IO.getDBConnection();
2 Statement statement = conection.createStatement();
3
4 @critical
5 Boolean result = statement.execute(query);
```

Listing 3.6: Vereinfachte Sink-Vorschrift

Vorgänger des Sink-Statements sind. Dadurch fällt das letzte Statement in Zeile 21 ebenfalls weg. Übrig bleibt die vollständige Sink-Vorschrift, die in Listing 3.6 zu sehen ist.

Sanitizer-Vorschrift

Die Vorgehensweise bei Sanitizer-Vorschriften entspricht zu großen Teilen der von Sink-Vorschriften. Mit Sanitizern werden tainted-Variablen gesäubert, weshalb auch Sanitizer zwingend von solchen Variablen abhängen. Hier ist aber in gewissem Maße relevant, wie die Variable zwischen Source und Sanitizer verwendet wird. Ein Sanitizer wird in vielen Fällen Teil eines If-Blocks sein, wo die Variable auf Gültigkeit oder Ungültigkeit überprüft wird. Dieser ist deshalb natürlich wichtiger Bestandteil des Sanitizers. Deshalb sind die tainted-Variablen in diesem Fall nicht aus $relVars$ zu entfernen. Da alle Source-Statements zu Beginn der Analyse aus dem Vorgängergraph entfernt wurden, entspricht das Ergebnis dem Verhalten der Variable zwischen Source und Sanitizer. Das restliche Vorgehen entspricht dem der Sink-Vorschriften.

Das Beispiel hat kein Sanitizer-Statement, das Vorgehen ist jedoch analog zu den Sink-Vorschriften.

3.4. Regelverfeinerung

Der letzte Schritt der Regelerstellung ist die Verfeinerung von Regeln. Jede im vorherigen Schritt gewonnene Regel ist zunächst sehr spezifisch: Von Variablennamen abgesehen wird nur Code gefunden, der den Beispielen ähnelt, aus dem die Regel entstanden ist. Es ist natürlich möglich, eine Regel um weitere Beispiele zu erweitern und so alternative Vorschriften zu gewinnen. Doch auch diese sind sehr spezifisch, und später in der Analyse wächst der Rechenaufwand mit jeder Vorschrift. Schöner wäre es, wenn Vorschriften, die sich nur in einigen Details unterscheiden, vereinigt werden, und dann bestenfalls sogar fehlerhaften Code finden, der nicht explizit als Beispiel gegeben war. So würden durch Verallgemeinerung der Vorschriften verfeinerte Regeln entstehen.

Die Möglichkeiten Regeln zu verallgemeinern sind natürlich vielfältig, weshalb hier nur beispielhaft ein Verfahren vorgestellt wird, das für die spätere Evaluierung als geeignet

```
void funcA(int b) {
  int a = b+2;
}

void funcB(int b) {
  int a = b-3;
}

void funcC(int b) {
  int a = b*5;
}
```

Listing 3.7: Beispielfunktionen für die Regelverfeinerung

erschien. Weiterhin wird als Einschränkung angenommen, dass eine Regel nur einen einzelnen linearen Flusspfad hat. Für Vorschriften, die mehrere Flusspfade besitzen, müssten alle Pfade gesondert betrachtet werden. Zusätzlich müssen die Ergebnisse der verschiedenen Flusspfade miteinander vereinigt werden. Durch diese Einschränkung wird die Komplexität der folgenden Vorgehensweise also erheblich reduziert.

Das Vorgehen wird anhand des einfachen Beispielcodes in Listing 3.7 erläutert. Es besteht aus drei Funktionen die verallgemeinert werden sollen. Die Gemeinsamkeiten der drei Funktionen sind ersichtlich: Alle bestehen aus einer Zuweisung und alle führen eine Operation auf eine Variable und ein Literal aus. Die einzigen Unterschiede sind der Wert des Literals und der Operator. Geht man davon aus, dass keine Annotationen an der Zuweisung vorliegen, dann würde das Literal schon bei der Regelerstellung entfernt werden. In dem Fall ist der einzige Unterschied zwischen den drei Funktionen also der Operator, der Rest ist identisch. In diesem einfachen Beispiel ist das schnell ersichtlich und ließe sich auch leicht programmatisch erkennen, doch für komplexeren Code sind die Gemeinsamkeiten viel aufwändiger zu finden. Deshalb bietet es sich an, die Suche nach Gemeinsamkeiten und Unterschieden auf ein Data-Mining-Problem abzubilden, im speziellen auf das Finden von *Closed Association Rules*. Diese Herangehensweise wurde früher schon von [18] zur Analyse von Source-Code verwendet (vgl. Abschnitt 2.3).

Für das Finden der Association Rules ist es notwendig, die Funktionen auf jeweils eine Transaktion abzubilden. Eine Transaktion besteht aus einer Menge nominaler Werte, den sogenannten Items. Es ist also nötig, ein Schema zu entwickeln, mit dem sich die Bestandteile der Statements eindeutig auf solche Werte abbilden lassen. Hervorzuheben ist, dass diese Algorithmen mit Mengen arbeiten, also kein Wert doppelt vorkommt. Es macht also zum Beispiel keinen Unterschied, ob es eine Integer-Variable oder hundert gibt. Auch würde jede Addition auf den selben Wert abgebildet werden. Es lassen sich so zwar trotzdem Regeln finden, aber die Gefahr, dass man ungewollte Änderungen vornimmt ist umso größer, je mehr Elemente des Codes auf den selben Wert abgebildet werden. Hier setzt die vorher gemachte Einschränkung an, dass nur Vorschriften betrachtet werden, die aus einem einzelnen Flusspfad bestehen. Wenn

die Position eines Statements innerhalb des Pfades mit in den nominalen Wert integriert wird, dann werden deutlich weniger gleichartige Elemente auf den selben nominalen Wert abgebildet. Dies ist dann nur noch innerhalb des selben Statements möglich.

In [18] wurde aus jedem Statement ein Hash berechnet, der als Item benutzt wurde. In diesem wurde jedoch die Position des Statements nicht berücksichtigt. Deshalb wird das folgende Schema vorgeschlagen, in dem die Werte in der gegebenen Reihenfolge als String enkodiert werden:

1. die Position innerhalb des Pfades
2. der Typ des Statements
3. der Typ des Ausdrucks
4. von der Verwendung abhängige Werte des Ausdrucks

Alle Teile des Wertes sind dabei durch einen Punkt voneinander getrennt. Dadurch entsteht eine Hierarchie. Durch das gemeinsame Präfix zweier Werte lässt sich bestimmen, wie groß die Ähnlichkeiten zwischen ihnen sind. Alle Werte, die im selben Statement benutzt werden, haben die selbe Position. Wenn darüber hinaus die Statements den gleichen Typ haben stimmt auch der zweite Wert, usw. Ganz am Ende kommen Parameter eines Statement-Typs, etwa der Wert eines Literals oder der Name des verwendeten Typs. Im Folgenden sei der Typ eines Items sein nominaler Wert ohne diesen Parameter.

Funktion	**Transaktion**
funcA	1.EXP.OP.ASSIGN, 1.EXP.ID.int, 1.EXP.OP.ADD
funcB	1.EXP.OP.ASSIGN, 1.EXP.ID.int, 1.EXP.OP.SUB
funcC	1.EXP.OP.ASSIGN, 1.EXP.ID.int, 1.EXP.OP.MULT

Tabelle 3.5.: Transaktionen der Beispielfunktionen aus Listing 3.7

Tabelle 3.5 zeigt, wie die Transaktionen für das gegebene Beispiel aussehen können. EXP steht dabei für ein Expression-Statement, OP für einen Operator und ID für einen Bezeichner. Wie man sieht unterscheiden sich die drei Transaktionen nur im letzten Wert, dem zweiten Operator. Es wurden nur Items für die Ausdrücke der Statements verwendet, keine für die Statements selbst. Dies wäre redundant, da jedes Statement, in dem Verallgemeinerungen möglich sind, aus mindestens einen Ausdruck besteht. Mit den Transaktionen können nun die Frequent Closed Itemsets berechnet werden. In diesem Beispiel gibt es vier Stück, welche in Tabelle 3.6 aufgezählt sind.

Neben den Ausgangstransaktionen wurde noch ein weiteres Itemset gefunden, welches eine echte Teilmenge von allen drei Transaktionen ist. Dieses Itemset beinhaltet genau die maximale Menge der Gemeinsamkeiten aller Transaktionen. Als nächstes werden mit den gewonnenen Itemsets die Closed Association Rules berechnet. Das Ergebnis ist in Tabelle 3.7 zu sehen. Wie erwartet gibt es drei Regeln, wobei die Gemeinsamkeiten jeweils die Unterschiede implizieren.

Itemset	Support
1.EXP.OP.ASSIGN, 1.EXP.ID.int	3
1.EXP.OP.ASSIGN, 1.EXP.ID.int, 1.EXP.OP.ADD	1
1.EXP.OP.ASSIGN, 1.EXP.ID.int, 1.EXP.OP.SUB	1
1.EXP.OP.ASSIGN, 1.EXP.ID.int, 1.EXP.OP.MULT	1

Tabelle 3.6.: Frequent Itemsets der Transaktionen aus Listing 3.5

Antezedens	Konsequenz	Supp.	Conf.
1.EXP.ID.int, 1.EXP.OP.ASSIGN	1.EXP.OP.ADD	3	1/3
1.EXP.ID.int, 1.EXP.OP.ASSIGN	1.EXP.OP.SUB	3	1/3
1.EXP.ID.int, 1.EXP.OP.ASSIGN	1.EXP.OP.MULT	3	1/3

Tabelle 3.7.: Association Rules der Transaktionen aus Listing 3.5

Nun gilt es, diese Association Rules für die Verallgemeinerung der Funktionen zu benutzen. Dazu werden die Association Rules zunächst gruppiert. Alle Association Rules mit gleicher linker Seite werden in eine Gruppe eingeteilt. Im Beispiel gehören alle Regeln zur selben Gruppe. Innerhalb einer Gruppe unterscheiden sich nur die rechten Seiten der Association Rules, weshalb die linke Seite für die nächsten Schritte ignoriert werden kann. Betrachtet wird nur noch die Vereinigungsmenge aller rechten Seiten einer Gruppe.

Diese gibt die Elemente an, in denen sich die Funktionen unterscheiden. Wenn es also möglich ist, die Funktionen zu verallgemeinern, dann muss es möglich sein, mindestens einen Teil der Items dieser Menge auf das selbe, verallgemeinerte Item zu überführen. Dazu wird der folgende einfache Algorithmus vorgeschlagen: Nehme ein Item *first* aus der Menge. Bestimme nun alle Items, die den selben Typ haben, und versuche diese Items miteinander so zu vereinigen, dass sie einer allgemeineren Version von *first* entsprechen. Falls dies mit mindestens einem Teil der Items (einschließlich *first*) gelingt, entferne diese. Ansonsten lässt sich *first* nicht verallgemeinern und muss entfernt werden. Führe dies solange fort, bis keine Items mehr in der Menge übrig sind. Das Vorgehen ist in Algorithmus 4 dargestellt.

Es wird also versucht, alle gleichartigen Items miteinander zu vereinigen. Jedes Item stehe dabei stellvertretend für alle Teilausdrücke von denen es erzeugt wurde. Die Vereinigung selbst kann vielfältig sein und aus mehreren Verfahren bestehen, welche abhängig vom Typ der Items ausgeführt werden. Im Beispiel würde versucht werden, einen verallgemeinerten Operator zu finden, der stellvertretend für Addition, Subtraktion und Multiplikation stehen kann. Dies könnte beispielsweise ein „beliebige mathematische Operation"-Operator sein. Das Ergebnis der Verallgemeinerung könnte also wie in Listing 3.8 aussehen, wobei $\oplus$ für diesen Operator stehe.

Natürlich sind die Ergebnisse einer solchen Verallgemeinerung fehleranfällig. Es kann passieren, dass es zu vermehrten FPs kommt, da mit dem allgemeineren Code eventuell auch Nutzungen gefunden werden, die nicht fehlerhaft sind. Die Verallgemeinerung ist deshalb ein iterativer Prozess, man muss sowohl aus fehlerhaftem als auch

Algorithmus 4 : Vereinigung

Input : Menge *rules* der Assocation Rules einer Gruppe

Sei *items* die Menge, die alle Elemente der rechten Seiten aus *rules* beinhaltet.
while *items ist nicht leer* **do**
  Sei *first* ein Element aus *items*.
  Sei *matching* die Menge der Elemente aus *items*, die den selben Typ wie *first* haben.
  Versuche, die Elemente in *matching* mit *first* zu vereinigen.
  if *es ließen sich Elemente vereinigen* **then**
    Entferne alle Elemente aus *items*, die sich vereinigen ließen.
  else
    Entferne *first* aus *items*.
  end
end

```
1 void func(int b) {
2   int a = b ⊕ literal;
3 }
```

Listing 3.8: Verallgemeinertes Beispiel aus Listing 3.7

aus fehlerfreiem Code lernen. Ein FP könnte in der nächsten Iteration dazu genutzt werden, die Verallgemeinerung einzuschränken, oder den entsprechenden Code explizit als fehlerfrei zu vermerken. Ebenso ist es denkbar, mit stochastischen Methoden zu arbeiten, um die Wahrscheinlichkeit einer Fehlerkennung abzuschätzen und entsprechende Gegenmaßnahmen einzuleiten. Dies geht jedoch über den Rahmen dieser Arbeit hinaus und sei nur als Ausblick angesprochen.

3.5. Nutzung der Vorschriften in der Analyse

Die Vorgehensweise der Analyse ist in großem Maße abhängig vom verwendeten Analyse-Framework. Die Analyse kann entweder direkt am Syntax-Baum stattfinden, oder kompilierten Bytecode verwenden. Der Anschaulichkeit halber werden die Konzepte in diesem Abschnitt so vorgestellt, wie so von einer Analyse am Syntax-Baum verwendet werden. Weiterhin wird die Analyse auf isolierte Funktionen eingeschränkt, ist also intraprozedural. Eine interprozedurale Analyse ist ungleich aufwändiger als eine intraprozedurale. In einer idealen interprozeduralen Analyse müsste der Datenfluss durch das gesamte Programme analysiert werden. Dies ist nicht effizient möglich, weshalb es vieler fortgeschrittener Techniken bedarf, welche die Zusammenhänge zwischen den einzelnen Funktionen lediglich approximieren. Dies war im Rahmen dieser Arbeit zeitlich nicht umsetzbar.

Das Ergebnis der Regelerstellung ist eine Menge von Vorschriften. Es kann davon beliebig viele geben und sie können verschiedene Typen haben. Während der Analyse werden diese Vorschriften genutzt, um fehlerhaften Code zu identifizieren. Eine Vorschrift besteht wie erläutert aus einer Folge von Statements und einer Menge an Variablen, die innerhalb dieser Statements benutzt werden. In der Analyse müssen entsprechende „Partner“ für diese Variablen gefunden. Es ist deshalb erforderlich, dass eine Liste mit den aktuell im analysierten Code existierenden Variablen geführt wird. Bei einer Variablendeklaration muss die neue Variable in die Liste aufgenommen werden, falls der Gültigkeitsbereich einer Variable endet muss sie wieder entfernt werden. Da die Namen der Variablen im Allgemeinen nicht mit denen übereinstimmen, die in der Regel verwendet werden, müssen dynamisch Abbildungsfunktionen gebildet werden, mit denen die Variablennamen aufeinander abgebildet werden.

Die Vorschriften entsprechen dem minimal notwendigen Code, um ein bestimmtes Ereignis zu erzielen. Es ergibt also Sinn, für jede Vorschrift einen Automaten zu erstellen. Jeder Zustand beschreibt dabei eine Stelle in der Vorschrift, während die Übergänge das als nächstes zu findende Statement angeben. Wenn es gefunden wurde wird ein Zustandswechsel ausgelöst.

Ein endlicher Automat ist für diese Analyse nicht ausreichend. Der Automat muss sich neben dem aktuellen Zustand noch die oben beschriebene Variablenbelegung merken. Doch auch mit dieser Erweiterung ist der Automat für die Analyse noch nicht ausreichend, da er nur einen einzelnen aktuellen Zustand hat. Ein Sink kann jedoch beispielsweise einmal erstellt werden und anschließend mehrfach mit verschiedenen Daten an verschiedenen Stellen eine Regelverletzung darstellen. Deshalb ist es notwendig, dass der Automat mehrere Zustände gleichzeitig verwalten kann. Um Mehrdeutigkeiten zu vermeiden ist zwischen den verschiedenen Arten von Zuständen zu unterscheiden. Im Folgenden bezeichne *Zustand* eine Positionen innerhalb des Automaten, die über Transitionen mit anderen Zuständen verbunden ist. Eine *Konfiguration* sei ein Tupel, welches auf einen Zustand des Automaten verweist und zusätzlich eine Variablenbelegung speichert. Eingaben des Automaten müssen an jede der aktiven Konfigurationen weitergereicht werden, die dann Versuchen einen Zustandswechsel herbeizuführen. Bei so einem Wechsel muss eine neue Konfiguration mit dem Folgezustand abgeleitet werden, die alte Konfiguration jedoch auch beibehalten werden. Konfigurationen bleiben solange erhalten bis sie aufgrund ihrer Variablenbelegung die Gültigkeit verlieren, etwa wenn der Gültigkeitsbereich einer Variable verlassen wird.

Zusammengefasst besteht der Automat aus den folgenden Elementen:

- Eine Menge von Zuständen, die die Positionen in der Vorschrift angeben
- Eine Menge von Transitionen, die angeben, welche Statements einen Zustandswechsel auslösen
- Ein Startzustand und ein Endzustand
- Eine Menge von Konfigurationen, die auf einen Zustand verweisen und eine Variablenbelegung speichern

Die maximal mögliche Anzahl Konfigurationen wächst dabei exponentiell mit der An-

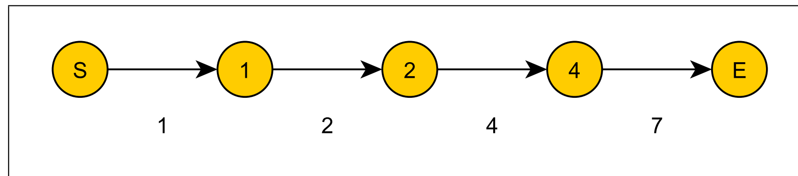

Abbildung 3.4.: Automat der Source-Vorschrift aus Listing 3.5

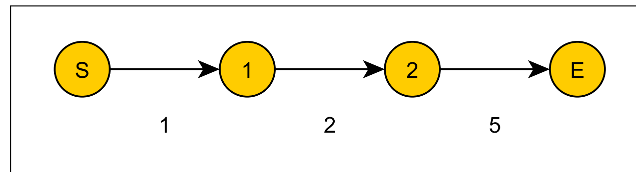

Abbildung 3.5.: Automat der Sink-Vorschrift aus Listing 3.6

zahl Statements im analysierten Code, da sich die Anzahl Konfigurationen in jedem Schritt potentiell verdoppeln kann. Dies klingt zunächst problematisch, in der Praxis zeigt sich jedoch, dass sich pro Statement nur relativ wenige neue Konfigurationen ableiten lassen. Um den Worst-Case zu erreichen müsste jedes Statement einen Übergang in jeder bestehenden Konfiguration auslösen, was mit sinnvollem Code nur schwer zu erreichen ist. Zudem werden regelmäßig Konfigurationen entfernt, wenn die verwendeten Variablen ihre Gültigkeit verlieren. Darüber hinaus werden Konfigurationen, die auf den Endzustand verweisen, nicht in die Konfigurationsmenge aufgenommen.

Die Abbildungen 3.4 und 3.5 stellen die Automaten dar, die aus der Source-Vorschrift in Listing 3.5 und der Sink-Vorschrift in Listing 3.6 entstehen. Ein Knoten steht für eine Position innerhalb der Vorschrift. S gibt den Startzustand an, E den Endzustand. Eine Nummer steht für die Zeile des zuletzt gefundenen Statement. Die Zahlen an den Kanten geben die Zeile des Statements an, welches für einen Zustandswechsel gefunden werden muss. Um im Source-Automaten von Zustand 4 zu Zustand E zu wechseln ist beispielsweise das Statement `data = reader.readLine()` nötig.

Die Automaten erwarten eine Folge von Statements, mit denen sie neue Konfigurationen ableiten können. Für Datenfluss-Vorschriften ist es aber nicht ausreichend, einfach den zu analysierenden Code als Ganzes zu übergeben. Ob eine Variable tainted oder untainted ist hängt von ihrem Fluss durch die Funktion ab. Auf einem Pfad kann sie untainted sein, während ein anderer möglicher Pfad eine Regelverletzung enthalten kann. Es ist deshalb notwendig, jeden einzelnen Flusspfad durch die Funktion gesondert zu betrachten. Die gängigsten Analyse-Frameworks beinhalten schon Verfahren, mit denen sich diese Pfade berechnen lassen, sodass bei der Implementierung hierdurch nur wenig zusätzlicher Aufwand entsteht. Die Folge ist natürlich, dass der Zeitbedarf der Analyse mit zunehmender zyklomatischer Komplexität ansteigt. Neben den Datenfluss-Regeln ergibt es Sinn, auch für strukturelle Regeln die verschiedenen Flusspfade zu analysieren, da nur auf diese Art die `@required`-Annotation zuverlässig ausgewertet werden kann: Falls ein Zustand erreicht wird, dessen Transition auf

ein mit `@required` annotiertes Statement verweist, wird dies innerhalb der Konfiguration vermerkt. Sollte es möglich sein, eine neue Konfiguration aus ihr ableiten, dann kann die Konfiguration gelöscht werden. Sollte sie aus einem anderen Grund entfernt werden, etwa weil das Ende des Flusspfades erreicht ist, oder weil der Gültigkeitsbereich einer Variable verlassen wurde, so muss ein Regelverstoß ausgegeben werden. Somit wird ein Fehler ausgegeben, falls es mindestens einen Pfad gibt, auf dem das notwendige Statement nicht vorkommt.

Während der Analyse wird jede Funktion für sich betrachtet. Innerhalb der Funktion werden alle möglichen Kontrollflusspfade analysiert, die durch die Funktion möglich sind. Zu Beginn eines Pfades müssen alle Konfigurationen zurückgesetzt werden, da jeder Pfad für sich betrachtet wird. Die gefundenen Verstößе werden über alle Pfade aggregiert. Alle zur Regel zugehörigen Vorschriften werden zur selben Zeit analysiert. Ein Statement kann in beliebig vielen Vorschriften einen Zustandswechsel auslösen. Dies ist etwa der Fall, wenn verschiedene Sources eine gemeinsame Abhängigkeit haben. Der Zustand der Variablen muss außerhalb der Automaten verwaltet werden, da voneinander unabhängige Vorschriften auf ihren Zustand zugreifen müssen. Eine Source-Vorschrift muss eine Variable auf tainted setzen, was die Sink-Vorschrift später auswerten muss. Würde der Wert der Variablen innerhalb des Automaten gespeichert werden wäre dies nicht möglich.

Wenn der Endzustand erreicht ist, wird eine Aktion ausgelöst, die vom Typ der Vorschrift abhängt:

- Bei strukturellen Vorschriften wird ein Regelverstoß vermerkt
- Bei Source-Vorschriften wird die Variable der Zuweisung auf tainted gesetzt
- Bei Sanitizer-Vorschriften wird für die Variable vermerkt, dass sie bereinigt wurde
- Bei Sink-Vorschriften wird ein Regelverstoß vermerkt

Hervorzuheben ist die Aktion der Sanitizer-Vorschriften. Es ist nicht ausreichend, die Variable einfach auf untainted zu setzen. Dann wird zwar im aktuellen Pfad kein Verstoß gefunden, aber es kann vorkommen, dass der Sanitizer in einem anderen Pfad nicht ausgeführt wird und dort entsprechend ein Verstoß registriert wird. Deshalb ist es notwendig, dass bei einem ausgeführten Sanitizer die Variable trotzdem als tainted weiterbehandelt wird, jedoch in ihr vermerkt wird, dass sie einen Sanitizer durchlaufen hat. Wenn bei einem Sink ein Verstoß registriert wird, ist dieser Status auszulesen und mit dem Verstoß zu speichern. Nachdem alle Pfade analysiert wurden, werden die vermerkten Verstöße gefiltert. Falls mehrere Pfade den selben Verstoß registrierten (also die selben Source, den selben Sink, und die selbe tainted-Variable), werden die Duplikate entfernt. Ist einer der redundanten Verstöße jedoch als bereinigt vermerkt worden, so müssen alle Vorkommen dieses Verstoßes entfernt werden.

Durchführung an einem Beispiel

Zuletzt wird die Durchführung an Listing 3.1 demonstriert. Die Source-Vorschrift aus Listing 3.5 und die Sink-Vorschrift aus Listing 3.6, deren Automaten schon in den Abbildungen 3.4 und 3.5 dargestellt sind, werden als Regel verwendet. Da sie aus Listing 3.1 abgeleitet wurden ist davon auszugehen ist, dass ein Fehler gefunden wird.

Zeile	Beschreibung
0	Erstelle Startkonfigurationen SourceS und SinkS
1	Keine Aktion
3	Neue Konfiguration Source1 im Source-Automaten
4	Neue Konfiguration Source2 im Source-Automaten
6	Neue Konfiguration Source3 im Source-Automaten
8	Neue Konfiguration Source4 im Source-Automaten. Setze `data` auf tainted
10	Neue Konfiguration Sink1 im Sink-Automaten
11	Neue Konfiguration Sink2 im Sink-Automaten
13	Propagiere tainted-Wert von `data` zu `query`
15	Neue Konfiguration Sink3 im Sink-Automaten. Ausgabe einer Regelverletzung
17	Keine Aktion

Tabelle 3.8.: Schritte der Analyse des Codes aus Listing 3.1

Tabelle 3.8 zeigt die einzelnen Schritte der Analyse des Codes. Jede Zeile referenziert dabei eine Zeile des Beispiels und beschreibt die Aktion, die für das dazugehörige Statement durchgeführt wird. Die Tabellen 3.9 und 3.10 geben die einzelnen Konfigurationen an, die während der Analyse entstehen. Jeder Konfiguration wird ein Name zugewiesen, damit sie in Tabelle 3.8 referenziert werden kann. Weiterhin wird angegeben, aus welcher Konfiguration sie abgeleitet wurde, welchen Zustand sie referenziert und welche Variablenabbildungen in der Konfiguration gespeichert sind. Diese sind von der Form $vorschrift \rightarrow analyseCode$.

Name	Abgl. aus	Zust.	Abbildungen
SourceS	-	S	$\emptyset$
Source1	SourceS	1	{ socket → socket }
Source2	Source1	2	{ socket → socket, input → input }
Source3	Source2	4	{ socket → socket, input → input, reader → reader }
Source4	Source3	E	{ socket → socket, input → input, reader → reader, data → data }

Tabelle 3.9.: Konfigurationen des Source-Automaten aus Abbildung 3.4

Das Beispiel besitzt nur einen Flusspfad, weshalb auch nur eine Ausführung der Automaten nötig ist. Zu Beginn der Ausführung werden für beide Automaten Startkonfigu-

Name	Abgl. aus	Zust.	Abbildungen
SinkS	-	S	$\emptyset$
Sink1	SinkS	1	{ connection $\rightarrow$ connection }
Sink2	Sink1	2	{ connection $\rightarrow$ connection, statement $\rightarrow$ statement }
Sink3	Sink2	E	{ connection $\rightarrow$ connection, statement $\rightarrow$ statement, query $\rightarrow$ query }

Tabelle 3.10.: Konfigurationen des Sink-Automaten aus Abbildung 3.4

rationen erstellt. In Zeile 1 kann keine neue Konfiguration abgeleitet werden, sodass sie zu keiner Aktion führt. Die Zeilen 2 bis 8 führen jeweils zu einer neuen Konfiguration im Source-Automaten. Jede Konfiguration wird dabei um eine Variablenabbildung ergänzt, die in diesem Fall allerdings der Identität entspricht. In Zeile 8 wird ein Endzustand des Source-Automaten erreicht, weshalb die Variable `data` auf tainted gesetzt wird. Zeile 10 kann eine neue Konfiguration des Sink-Automaten erzeugen. Hier wird nur die Variable `connection` zum Mapping hinzugefügt, da `IO` eine statische Variable ist. In Zeile 12 wird der tainted-Wert von `data` auf `query` propagiert. Zeile 15 führt schließlich einen Endzustand des Sink-Automaten herbei. Somit wird hier eine Regelverletzung ausgegeben. Es findet keine Abbildung von `result` statt, da bei Sinks der Rückgabewert explizit ignoriert wird.

Somit wurde im Beispiel insgesamt eine Regelverletzung gefunden: In Zeile 8 wurde ein Wert aus einer nicht vertrauenswürdigen Quelle ausgelesen, und in Zeile 15 in einem SQL-Query verwendet.

4. Prototypische Umsetzung

In diesem Kapitel wird anhand des während der Arbeit entwickelten Prototypen eine Übersicht über die Implementierung der Konzepte aus Kapitel 3 gegeben. Der Schwerpunkt soll dabei nicht auf der genauen Funktionsweise des Prototypen liegen, sondern mehr darin, die algorithmischen Schwierigkeiten aufzuzeigen und zu diskutieren. Der Prototyp selbst ist ein in Java implementiertes Kommandozeilenprogramm. Die Regeln werden ebenfalls aus Java-Code erstellt. Die Implementierung lässt sich in folgende Teilprobleme aufteilen, die größtenteils den Abschnitten aus Kapitel 3 entsprechen:

1. Wahl eines Analyse-Frameworks, für das die Implementierung stattfinden soll
2. Einlesen des Source-Codes in ein Format, welches Analysen und Operationen zur Vereinfachung erlaubt
3. Erstellen von Vorschriften aus diesem Format (vgl. Abschnitt 3.3)
4. Verallgemeinerung der Vorschriften (vgl. Abschnitt 3.4)
5. Speicherung der Vorschriften als Regelmenge für das gewählte Framework
6. Implementierung einer Regel für selbiges Framework, welche die Vorschriften einlesen und damit Analysen durchführen kann (vgl. Abschnitt 3.5)

Einige Features der Java-Sprache wurden aus Zeitgründen für den Prototypen vernachlässigt. Als Basis wurde die Java-1.5-Grammatik verwendet, mit folgenden Einschränkungen:

- Es werden nur primitive und qualifizierte Typen unterstützt, keine Generics
- Switch-Statements werden nicht unterstützt
- Bei Try-Catch-Statements ist nur ein einzelner Catch-Block und kein Finally-Block erlaubt

Auf Code, der mit den entstanden Regeln analysiert wird, trifft nur die erste Einschränkung zu.

4.1. Analyseprogramme

Zunächst gilt es, ein passendes Programm auszuwählen, für das die Regeln erstellt werden können. Dazu werden bestehende Frameworks zur statischen Analyse von Java-Code diskutiert. Ziel ist es, das passendste Framework für die Implementierung der Konzepte auszuwählen. Sie alle zu besprechen würde den Rahmen sprengen,

weshalb eine Vorauswahl stattgefunden hat, wo die geeignetsten Kandidaten ausgewählt wurden. Primäres Kriterium war der Grad der Erweiterbarkeit, aber auch der Verbreitungsgrad wurde beachtet. Ein Framework was von niemandem benutzt wird hat nur einen begrenzten Nutzen. Kommerzielle Programme wurden nicht berücksichtigt. Verglichen werden FindBugs, was mit Bytecode arbeitet und PMD, welches direkt den Source-Code analysiert. Sie zählen zu den verbreitetsten erweiterbaren Programmen zur statischen Analyse von Java-Code.

4.1.1. FindBugs

FindBugs ist eines der am meisten genutzten statische Analyseprogramme für Java. Es startete aus der Erkenntnis, dass sich einige blatante Fehler mittels simpler Analysen finden lassen [5], die dafür nicht alle möglichen Verstöße zurückgeben müssen [21]. Inzwischen erkennt es über 300 Probleme. FindBugs versucht, einen Kompromiss zwischen Korrektheit und Anzahl Warnungen zu finden. So ist es nicht darauf ausgelegt, alle möglichen Probleme zu finden. Es versucht stattdessen, nur die Probleme zu melden, bei denen eine hohe Chance besteht, dass der User sie auch finden möchte. Das kann unter Umständen zu FNs führen, hält die Anzahl Funde jedoch in einem überschaubaren Rahmen und erhöht somit die Effizienz des Benutzers. FindBugs arbeitet mit Bytecode, sodass das Programm in kompilierter Form vorliegen muss. Es verfügt über eine intraprozedurale Datenfluss-Analyse, die unter anderem für das Überprüfen von Null-Pointer-Dereferenzierungen genutzt wird [22].

FindBugs kann um neue Regeln erweitert werden, indem das Interface `Detector` implementiert wird. Da es direkt mit dem kompilierten Bytecode arbeitet ist hierzu Vorwissen über dessen Struktur zwingend nötig. Es bietet dazu zwei verschiedene Implementierungen des Visitor-Patterns an. Die erste verwendet eine einzelne Funktion `sawOpCode(int seen)`, die für jeden Bytecode-Befehl aufgerufen wird. Die Regeln können damit ihre internen Daten, im Allgemeinen ein Automat, anpassen. Die zweite Implementierung bietet für jeden Bytecode eine eigene Funktion, etwa `visitBranchInstruction(BranchInstruction obj)` für Branches.

Die Datenfluss-Fähigkeiten sind vergleichsweise fortgeschritten. Die Abstraktion ähnelt der theoretischen Problemdefinition [21]. Um eine neue Datenfluss-Analyse zu entwickeln kann die Klasse `DataflowAnalysis` erweitert werden. Diese kann von `Fact` abgeleitete Objekte erzeugen und in der Analyse weiterreichen (etwa Variablen die tainted sind). Für interprozedurale Analysen können Zusammenfassungen des Verhaltens von Funktionen genutzt werden, etwa welche Werte für eine Funktion immer zum Rückgabewert `null` führen. Eine echte interprozedurale Analyse ist nicht möglich.

FindBugs wird unter anderem in Googles Testprozess eingesetzt und hat dort schon über 1000 Probleme gefunden [5].

4.1.2. PMD

PMD ist ein erweiterbares Open-Source Framework für statische Analysen, das erstmals 2002 erschienen ist und seitdem stetig weiterentwickelt wird. Inzwischen hat PMD über 100 Regeln für sechs verschiedene Sprachen. Es analysiert direkt den Source-Code des Programms, sodass es nicht in kompilierter Form vorliegen muss. Die Regeln sind in eine Vielzahl Kategorien aufgeteilt, sog. Rulesets. Die verschiedenen Rulesets können für Analysezwecke frei gewählt und kombiniert werden. Die Breite der Regeln ist groß, einige befassen sich mit rein stilistischen Problemen (beispielsweise das Braces-Ruleset, welches überprüft, ob If- und While-Statements geschweifte Klammern für ihren Inhalt nutzen) bis hin zu grundlegenden Datenfluss-Analysen. Neue Rulesets können einfach erstellt werden, es sind XML-Dateien, die auf bestehende Regeln verweisen. Hier können ihnen auch Parameter übergeben werden. PMD benutzt JJTree um den AST zu generieren, und nutzt dafür eine eigens entwickelte Grammatik. Neue Regeln können durch Erweiterung der Klasse `AbstractJavaRule` erstellt werden. Darüber hinaus existiert eine Regel, mit der XPath-Abfragen (*XML Path Language*) durchgeführt werden können. Für viele einfache Abfragen, wie etwa das vorher angesprochene Braces-Ruleset, ist diese Art Regel schon ausreichend. Für Regeln, die nicht nur reine Abfragen auf den AST durchführen, ist jedoch zwingend eine in Java geschriebene Regel notwendig. PMD wird mit einem Programm ausgeliefert, das die Erstellung von XPath-Regeln erleichtert.

Für diese Regeln bedient sich PMD dem Visitor-Pattern. `AbstractJavaRule` stellt für jede Konstruktion in der Grammatik eine `visit`-Methode bereit, die von der implementierten Regel überschrieben werden kann. Parameter für diese Methoden ist der Knoten, an dem die Produktion im AST vorkommt, und der Regelkontext, mit dem Informationen über die analysierte Datei abgefragt werden können. Mittels der JJTree-Funktionen kann auch auf die Kinder bzw. die Eltern des übergebenen Knotens zugegriffen werden. Die Ergebnisse der Analyse können in die Standardausgabe oder als CSV-, HTML- oder XML-Datei ausgegeben werden.

PMD galt lange Zeit als ungeeignet, da es keine Datenflussfähigkeiten hatte [21] und die mitgelieferten Regeln primär stilistischer Natur waren [22]. Der Aufbau des ASTs wird ebenfalls als unintuitiv und übermäßig allgemein kritisiert [21]. Inzwischen verfügt PMD allerdings über eine Datenflusskomponente. Sie ist jedoch bisher nur intraprozedural. PMD kann nicht mehr als eine Datei gleichzeitig analysieren, weshalb Analysen die das Zusammenspiel mehrerer Klassen betrachten, nicht möglich sind. Wenn alle Rulesets aktiviert sind gibt PMD eine vergleichsweise exzessive Anzahl Warnungen aus, was den Nutzen verringert [22].

4.1.3. Wahl des Frameworks

FindBugs ist ohne Frage das angemessenere Werkzeug für statische Analysen. Die Datenfluss-Fähigkeiten sind deutlich fortgeschrittener als die von PMD, es verfügt über einen höheren Verbreitungsgrad und es erlaubt flexiblere Analysen, in gewissem Maße auch über eine Funktion hinaus. Jedoch hat es hat eine nicht zu verachtende Hürde:

Es wird nicht der Source-Code analysiert, sondern der kompilierte Bytecode. Dazu ist ein detailliertes Wissen über den Bytecode und das Verhalten des Compilers nötig, um die Konzepte dieser Arbeit zu implementieren. Dies war zeitlich nicht umzusetzen, weshalb die Wahl letztendlich auf PMD fiel. Es ist zwar im Vergleich zu FindBugs schwächer, jedoch bietet es trotzdem alle benötigten Funktionen an.

4.2. Einlesen des Source-Codes

Im Prototypen selbst musste zunächst der Source-Code in eine brauchbare Form überführt werden. Dazu wurde eine ANTLR3-Grammatik (ANother Tool for Language Recognition Version 3 [23]) für Java 1.5 verwendet. Diese Grammatik musste leicht erweitert werden. Java erlaubt Annotationen nur an Typen- und Variablendeklarationen, allerdings nicht an beliebigen Statements. Dies ist jedoch für die Implementierung zwingend notwendig. Die erweiterte Grammatik erlaubt deshalb Annotationen an beliebigen Statements. Die Konsequenz ist, dass annotierte Beispiele kein gültiger Java-Code mehr sind. Dieser Umstand sollte allerdings aufgrund des potentiellen Nutzens der abgeleiteten Regeln vertretbar sein. Die ANTLR3-Grammatik besteht aus zwei Teilen: Einer Grammatik, die den Quellcode in eine Baumstruktur überführt, und eine Grammatik, die auf dieser Baumstruktur arbeitet (ein sog. Tree Walker). Der Tree-Walker hat ohne weiteres keine Funktion, er überprüft nur, ob die Eingabe der Grammatik entspricht. Er eignet sich aber dazu, eigene Ergänzungen durch Inline-Code hinzuzufügen. Dies wurde intensiv genutzt, um den Syntax-Baum in eine eigene Repräsentation zu übersetzen. Diese wurde darauf ausgelegt, die während der Regelerstellung und Analyse benötigten Operationen einfach durchführen zu können. In diesem Schritt werden auch einige Vereinfachungen durchgeführt. Variablendeklarationen werden in einfache Zuweisungen übersetzt, da es irrelevant ist, ob eine Zuweisung während oder nach der Deklaration stattfindet.

Die erste Hürde bestand daraus, eine geeignete Form für diese Repräsentation zu finden. Für den Prototypen wurde die Sprache in dieser Zwischencode-Form leicht abstrahiert. Es gibt dort nur noch vier verschiedene Grundelemente: Funktionen, Typen, Statements und Ausdrücke. Von Statements und Ausdrücken gibt es eine Vielzahl verschiedener Ausprägungen, weshalb es sich hier angeboten hat, jeweils eine abstrakte Basisklasse zu definieren und die Ausprägungen davon erben zu lassen. Die Basisklasse unterstützt die notwendige Funktionalität, die im späteren Verlauf notwendig ist. Insbesondere erlaubt sie Hierarchien durch eine Eltern-Kind-Relation. Die abgeleiteten Klassen ergänzen ihre eigene Funktionalität und dienen darüber hinaus der einfachen Unterscheidung des Typs.

Typen benötigen eine besondere Handhabung. Für spätere Operationen und die eindeutige Zuordnung von Typen ist der vollqualifizierte Name notwendig (Beispiel: java.lang.String statt String). Dieser wird im Allgemeinen nicht innerhalb des Codes verwendet. Während des Parsens müssen also die von der Datei verwendeten Imports vermerkt werden. Wenn ein Typ verarbeitet wird, wird mit den Imports und der Java-Reflection-Methode `Class.forName(name)` versucht, den vollqualifizierten Namen

zu ermitteln. Darüber hinaus gibt es zwei Spezialfälle: Basistypen wie String und primitive Typen wie int. Alle Basistypen befindet sich im Package java.lang, welches nicht explizit importiert werden muss. Dieses wird also ebenfalls geprüft. Der vollqualifizierte Name eines primitiven Typen ist der Typname selbst, er ändert sich nicht.

Weiterhin ist es hilfreich, schon in diesem Schritt Informationen über den Code zu sammeln. Während des Parsens wird für jede Funktion und jedes Scope innerhalb der Funktion eine Liste mit Variablen erstellt, die zusammen mit dem Funktionsobjekt gespeichert werden. Alle Ausdrücke, die eine Variable referenzieren, werden mit einem Verweis auf diese Liste versehen.

4.3. Regelerstellung

Die Regelerstellung folgt im Großen und Ganzen dem beschriebenen Vorgehen in Abschnitt 3.3. Die Probleme bestanden größtenteils daraus, die abstrakte Beschreibung anhand der Graphen in eine handhabbare Form umzusetzen.

Der Vorgängergraph wird im Prototypen nur auf sehr einfache Art approximiert. Wenn er dazu genutzt wird, alle Statements zu entfernen, die kein Vorgänger eines gegeben Statements s sind, werden einfach alle Statements entfernt, die nach s kommen. Dieses Vorgehen ist nicht ideal, funktioniert aber insbesondere für die unterstützten Statements annehmbar. Der einzige Fall, wo es nicht gut funktioniert, ist in einem Else-Block eines If-Statements. Die Statements des If-Blocks werden in diesem Fall nicht entfernt, da sie im Code vor dem Else-Block kommen. Ein ähnliches Verhalten wäre bei Switch/Case-Statements zu finden, welche jedoch von vornherein nicht unterstützt werden.

Die Operationen des Abhängigkeitsgraphen wurden durch ein iteratives Vorgehen implementiert. Eine gegebene Variablenmenge wird solange um ihre direkten Abhängigkeiten erweitert, bis sie in ihrer Größe gleich bleibt. Das Vorgehen ähnelt dem von Algorithmus 3, arbeitet nur in eine andere Richtung. Jedes Statement implementiert dazu die Methode `findDependencies`, welche die Abhängigkeiten der gegebenen Variablen innerhalb des Statements zurückgibt.

Für die Redundanzentfernung gibt es ebenfalls eine Methode, die jedes Statement und jeder Ausdruck implementieren muss: `removeRedundancies`. Sie nimmt als Eingabe eine Menge relevanter Variablen, und entfernt alle Referenzen auf andere Variablen. Literals werden ebenfalls vereinfacht, soweit nicht anderweitig annotiert. Ein Statement bzw. Ausdruck weist dafür zunächst seine Kinder an, die Redundanzen zu entfernen. Falls keine relevanten Variablen mehr verbleiben wird das Kind durch eine Wildcard ersetzt (Ausdrücke) bzw. entfernt (Statements). Anschließend entfernt es seine eigenen Redundanzen.

Die Regelerstellung ließ sich nun relativ einfach wie in Abschnitt 3.3 beschrieben umsetzen. Die Ergebnisse werden in Vorschriften gespeichert. Ein Spezialfall musste berücksichtigt werden: Die Analyse (vgl. Abschnitt 4.6) hatte Schwierigkeiten, Sanitizer zu erkennen, da eine Konfiguration dort nur durch eine Zuweisung um eine Variable er-

gänzt werden kann. Aus diesem Grund wurde an den Anfang der Sanitizer-Vorschriften eine Wildcard-Zuweisung an die tainted-Variablen hinzugefügt. Zu Beginn der Arbeit wurde vorausgesetzt, dass eine Variable nur über eine Zuweisung auf tainted gesetzt werden kann. Die Sources selbst wurden aus den Abhängigkeitsgraphen der Sanitizer-Vorschriften entfernt. Deshalb gibt es mindestens eine Zuweisung an entsprechende Variablen, die nicht schon Teil der Sanitizer-Vorschrift ist. Das Wildcard-Statement hat also keine negativen Auswirkungen.

4.4. Regelverfeinerung

Die Eingabe für die Verfeinerung sind die Vorschriften aus dem vorherigen Schritt. Für das Finden der Itemsets und Association Rules wurde die SPMF-Bibliothek [24] verwendet. Dabei handelt es sich um eine Open-Source-Implementierung einer Vielzahl von Data-Mining-Algorithmen, die leicht in ein Java-Programm zu integrieren ist. Jede Funktion wird nach dem Schema aus Abschnitt 3.4 in eine Transaktion übersetzt, wobei sich die Transaktion merkt, aus welchen Ausdrücken die Werte entstanden sind. Die Transaktionen werden nach dem Typ der Vorschrift in Gruppen eingeteilt, die zusammen verfeinert werden. Die Bibliothek arbeitet mit numerischen Werten, sodass die nominalen Werte in Zahlen überführt werden müssen. Hierfür ist es ausreichend, dass verschiedene Werte verschiedene Zahlen haben, sodass die verschiedenen Werte einfach durchnummeriert werden konnten. Die von der Bibliothek generierten Association Rules entsprachen nicht der Definition nach Abschnitt 2.2. Hier war also eine nachträgliche Filterung nötig, um redundante Regeln zu entfernen. Der Aufwand dafür war allerdings überschaubar, da die Itemsets selbst definitionsgemäß waren.

Mit den Association Rules kann die eigentliche Verfeinerung durchgeführt werden. Dazu wurde Algorithmus 4 implementiert. Der einzige nicht triviale Schritt ist dabei die Vereinigung der generierten Teilmenge von Items. Für diesen Schritt wurde eine abstrakte Basisklasse `AbstractUnifier` erstellt, von der geerbt werden kann, um ein Verfahren zu implementieren. Der Algorithmus versucht mit allen registrierten Unifiern eine Vereinigung durchzuführen, bis entweder eine Vereinigung möglich war oder kein weiterer Unifier verbleibt. Die Unifier geben eine Liste der Items zurück, die sie vereinigen konnten. Diese Items werden dann aus der Suchmenge entfernt. Die Menge an Unifiern ist beliebig erweiterbar. Im Rahmen des Prototypen wurden folgende zwei Unifier entwickelt:

- Die in Abschnitt 3.4 beschriebene Methode zur Verallgemeinerung mathematischer Operatoren
- Eine Verallgemeinerung von Typen auf gemeinsame Oberklassen oder Interfaces

Die Verallgemeinerung der Operatoren gestaltet sich einfach. Zunächst wird überprüft, ob das erste Item durch einen verallgemeinerten Operator ausgedrückt werden kann. Ist dies der Fall werden alle Items gesucht, die auf den selben Operator verallgemeinert werden können. Diese werden entsprechend angepasst und schließlich zurück-

Algorithmus 5 : Sammeln der Oberklassen

Input : Typ *type*
Output : Menge der Oberklassen von *type*

Erstelle leere Menge *supers*.
while *type ist nicht vom Typ* `Object` **do**
  Füge *type* zu *supers* hinzu.
  Füge alle Interfaces von *type* zu *supers* hinzu.
  type ← Oberklasse von *type*.
end

return *supers*

gegeben, damit sie aus der Suchmenge entfernt werden können.

Die Oberklassen-Verallgemeinerung ist ungleich aufwändiger. Jede Klasse in Java kann eine Oberklasse und eine beliebige Anzahl implementierter Interfaces haben. Bei der Bestimmung der Gesamtmenge der Oberklassen eines Typs muss das Vorgehen solange iterativ mit der Oberklasse wiederholt werden, bis die `Object`-Klasse erreicht ist, welche Oberklasse aller Java-Klassen ist. Algorithmus 5 zeigt das Vorgehen. Dies wird zunächst mit dem ersten Item durchgeführt. Danach wird es mit jedem Item der Eingabemenge wiederholt. Falls ein Item mindestens eine Oberklasse mit dem ersten Item gemein hat, wird es in einer separaten Liste vermerkt. Anschließend wird überprüft, welche der Oberklassen am häufigsten in diesen Items vorkommt. Diese Oberklasse ersetzt schließlich die Typen der passenden Items, welche letztlich aus der Suchmenge entfernt werden.

Die implementierten Verfahren sind nicht optimal und sollen nur beispielhaft dazu dienen, die Plausibilität dieses Vorgehens aufzuzeigen. Es wird falls möglich immer unreflektiert eine Verallgemeinerung durchgeführt. Dies kann schnell zu einer zu starken Verallgemeinerung führen. Insbesondere bei den Operatoren kann dies zu fehlerhaften Ergebnissen führen. Möglich wäre, die Confidence- und Support-Werte mit einfließen zu lassen, entweder, um die Verallgemeinerung zu steuern oder, um die Ergebnisse im nächsten Schritt, der Analyse, zu gewichten. Auch ist es denkbar, wie in Abschnitt 3.4 angesprochen, FPs aus der Analyse zu nutzen, um die Unifier einzuschränken.

4.5. Speicherung der Vorschriften

Damit die Vorschriften später im Analyse-Framework eingelesen werden können müssen sie in einem geeigneten Dateiformat gespeichert werden. Im Prototyp wurde XML verwendet, für welches es einfach zu bedienende Bibliotheken gibt. Jede verwendete Klasse deklariert dazu die Methoden `serialize`, mit der das Objekt in ein XML-Element überführt werden kann, und `deserialize`, mit dem aus einem XML-Element ein neues Objekt erzeugt werden kann. Die abstrakte Oberklasse der Statements und Ausdrücke speichert bzw. lädt die Elemente, die allen Ausprägungen gemein sind,

während die Spezialisierungen nur ihre eigenen Daten hinzufügen. Der Prozess wurde erheblich durch die Reflection-Fähigkeiten von Java vereinfacht. Für jedes Objekt wird der vollqualifizierte Name im XML-Baum gespeichert, und während des Einlesens als erstes extrahiert. Damit lässt sich mit `Class.forName` ein Handler-Objekt der Klasse erzeugen. Aus diesem kann mit der `getConstructor(types)`-Methode ein Verweis auf den Konstruktor erzeugt werden, der die gegeben Typen als Parameter nimmt. Mit diesem wird schließlich die `newInstance(parameter)`-Methode aufgerufen, die ein neues Objekt erzeugt. Dazu muss jede Klasse nur einen Konstruktor definieren, der ein XML-Element als Parameter nimmt. Durch dieses Vorgehen ließen sich die Vorschriften leicht und ohne großen Aufwand extern speichern und wieder einlesen. Neu ergänzte Klassen müssen nur die entsprechenden Funktionen bzw. den Konstruktor definieren, ansonsten sind keine weiteren Änderungen im Programm nötig.

Zuletzt muss ein PMD-Ruleset gespeichert werden, welches die Regel referenziert. Das Ruleset ist selbst eine XML-Datei und besteht aus einer Menge an Verweisen auf Regeln. Ein solcher Verweis wiederum besteht aus dem vollqualifizierten Namen der Regelklasse und einigen Feldern die sie beschreiben. Darüber hinaus können noch Parameter angegeben werden, die die Regel später auslesen kann. Im Prototyp wird die XML-Repräsentation der Regeln innerhalb des XML-Dokuments als String gespeichert. Bei der Instanziierung der Regel wird dieses Feld ausgelesen und in einen XML-Baum überführt.

4.6. Integration in Analyse-Framework

Wie beschrieben wurde das PMD-Framework durch eine Regel ergänzt, welche die vorher generierten Vorschriften als Parameter nimmt. Für diese Regel sind drei von PMDs `visit`-Methoden relevant: Compilation Unit (eine neue Datei), Import und Funktionsdeklarationen. Bei einem Import-Statement wird das Paket vermerkt, um später bei der Typenerkennung genutzt zu werden. In einer neuen Compilation Unit wird die Liste der Imports zurückgesetzt. Eine Funktion ist schließlich der Eintrittspunkt in die Analyse, sie wird als ganzes analysiert. PMD verfügt über ein System zur Typenerkennung, jedoch setzt es voraus, dass alle verwendeten Klassen explizit mit ihrem vollqualifizierten Namen importiert werden, d.h. Importe wie java.lang.* werden nicht unterstützt. Da dies in vielen Programmen aber üblich ist, wurde die eigens entwickelte Methode zur Typenerkennung verwendet. Für jede Regel wird eine neue Instanz der PMD-Regel-Klasse erzeugt, die ihre Parameter aus den vorher gespeicherten XML-Dateien ausliest.

Der AST, den PMD verwendet, erschwert in seiner Form die Analyse beträchtlich. Der Code wird an vielen Stellen nur unzureichend geparst. Beispielsweise werden Operatoren einfach als String in verschiedenen Ausdruck-Knoten gespeichert, ein Teil der Ausdrücke (Primäre Ausdrücke, Ausdrücke die keine Werte ändern) ist in einer Art Liste gespeichert (ein PrimayPrefix gefolgt von einer beliebigen Anzahl PrimarySuffix) anstatt baumartig angeordnet zu sein, Zuweisungen ergeben sich rein daraus, dass eine AssignmentOperator-Node zwischen zwei anderen Nodes gespeichert wird, etc.

Es wurde deshalb die Entscheidung getroffen, den AST in die Form zu übersetzen, die während der Regelerstellung verwendet wurde. Dies hat zusätzlich den Vorteil, dass sich der analysierte Code leicht mit dem Regelcode vergleichen lässt. Die Klassen `StatementConverter` und `ExpressionConverter` führen diese Konvertierung durch.

Die eigentliche Analyse baut auf der Datenfluss-Komponente von PMD auf. Für jede Funktion werden alle möglichen Pfade berechnet, die durch die Funktion genommen werden können. Jeder Pfad wird einzeln analysiert und die Ergebnisse aggregiert. Strukturelle Vorschriften werden ebenfalls auf diese Art analysiert, da dies eine zuverlässigere Erkennung von Befehlsfolgen ermöglicht. Die Klasse `DataFlowAnalyzer` realisiert die Analyse eines solchen Pfades. Der Pfad wird Statement für Statement durchgegangen. Die Statements werden vom AST in den Zwischencode überführt und bei Variablendeklarationen werden neue Variablen erzeugt, die in den Statements referenziert werden. Für jede Vorschrift der Regel wird ein Automat erzeugt, der später in diesem Abschnitt beschrieben wird. Das aktuelle Statement wird an jeden dieser Automaten übergeben, anschließend werden etwaige taint-Propagierungen durchgeführt. Nach Ende des Pfades werden die Funde zurückgegeben.

Den Kern der Analyse stellt der Automat dar, `RuleMachine`. Er wird basierend auf den Statements einer Vorschrift erstellt. Zunächst werden die Zustände und Zustandsübergänge generiert, indem die Statements der Vorschrift eingelesen werden. Danach wird die Startkonfiguration erstellt, zusammen mit einem „leeren“ Variablen-Mapping (d.h. ohne Variablen, auf die abgebildet wird). Diese Konfiguration wird in der Konfigurationsmenge gespeichert, die durch neue Konfigurationen ergänzt wird, sobald sie sich ableiten ließen. Mit jedem eingegebenen Statement wird nun versucht mit jeder aktuellen Konfigurationen eine neue Konfiguration abzuleiten. Wichtig ist, dass die neuen Konfigurationen erst bei der nächsten Eingabe verwendet werden, da es sonst zu falschen Ergebnissen führen könnte, wenn mit einem Statement zwei ähnliche aufeinanderfolgende Zustände abgedeckt werden.

Bei der Ableitung neuer Konfiguration wird das Eingabe-Statement mit dem Regel-Statement verglichen, auf das die Zustandsübergänge verweisen. Zunächst wird geprüft ob sich das Variablen-Mapping der alten Konfiguration ergänzen lässt, indem versucht wird, den Platzhalter im Regel-Statement durch eine etwaige neue Variable zu ersetzen. Dazu muss natürlich der Typ der Variable übereinstimmen. Ein bestehendes Mapping kann durch eine neue Variable überschrieben werden, da es möglich ist, das Ergebnis eines Funktionsaufrufs einer anderen Variable zuzuweisen (Beispiel: `s = s.substring(5)`). Falls das scheitert lässt sich aus der aktuellen Konfiguration keine neue ableiten. Ansonsten wird das Eingabestatement vereinfacht und mit dem Regel-Statement verglichen. Dazu bietet jedes Statement die Methode `compareWith`, die als Eingabe ein anderes Statement und ein Mapping nimmt. Ein Wildcard-Statement stimmt dabei mit jedem möglichen Statement überein. Im Erfolgsfall wird eine neue Konfiguration mit dem aktualisierten Mapping und einem Verweis auf den Folgezustand erstellt. Falls der Folgezustand der Endzustand einer Sink-Vorschrift ist wird anders vorgegangen. In diesem Fall wird versucht, einen Teilausdruck im Eingabe-Statement zu finden, der mit dem Regel-Statement übereinstimmt. Ist dieser gefunden

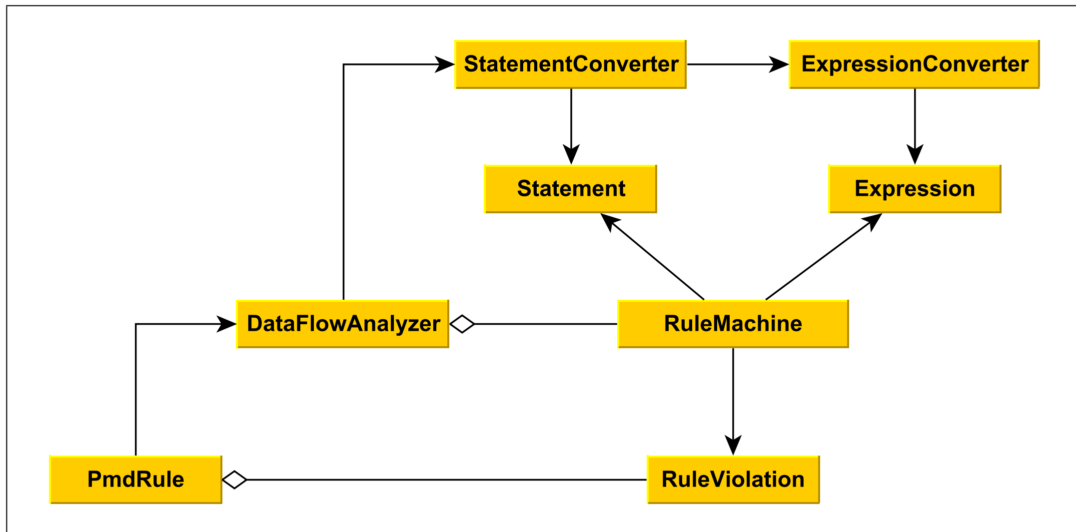

Abbildung 4.1.: Vereinfachtes Zusammenspiel der Klassen während der Analyse

wird überprüft, ob eine der verwendeten Variablen tainted ist. Ist dies der Fall, wird eine neue Konfiguration erstellt und registriert.

Bei einer neuen Konfiguration wird zunächst geprüft, ob diese auf den Endzustand verweist. Ist dies nicht der Fall wird sie einfach zur Konfigurationsmenge hinzugefügt. Falls es ein Endzustand ist, wird ein zusätzliches Ereignis ausgelöst, was vom Typ der Vorschrift abhängt. Bei einer Source-Vorscrift wird die Variable, der im letzten Statement ein Wert zugewiesen wurde, auf tainted gesetzt. Bei einer Sink- und einer strukturellen Vorschrift wird eine Regelverletzung registriert. Bei einer Sanitizer-Vorschrift wird in der Variable vermerkt, dass sie bereinigt wurde. Dieser Wert wird auch in der Regelverletzung gespeichert, wodurch sich später die Ergebnisse filtern lassen. Dieser Vorgang wurde detailliert in Abschnitt 3.5 erläutert. Die Handhabung der mit `@required`-annotierten Regel-Statements entspricht ebenfalls der dort beschriebenen Vorgehensweise.

Abbildung 4.1 stellt ein vereinfachtes UML-Diagramm der Klassen dar, die während der Analyse verwendet werden. `RuleViolation` entspricht einer Regelverletzung, `PmdRule` ist die für PMD entwickelte Regel.

Die in diesem Kapitel entwickelte Regel ist ausreichend um die Spezifikationen der Methode umzusetzen. Es werden nur relativ grundlegende Analysemethoden verwendet, doch ist das Vorgehen beliebig ausbaufähig.

5. Evaluation

In diesem Kapitel soll die entwickelte Methode anhand des Prototypen evaluiert werden. Betrachtet wird zunächst, ob das ableiten der Regeln aus dem Code funktioniert. Darüber hinaus wird getestet, wie der Lerneffekt ausfällt wenn mehrere Beispiele in eine Regel einfließen. Dieses Kapitel ist wie folgt organisiert: Zunächst wird in Abschnitt 5.1 das Bewertungsmaß vorgestellt, mit dem die Ergebnisse der Evaluation ausgewertet werden. Abschnitt 5.2 stellt den Datensatz vor, anhand dessen die Evaluation durchgeführt wird. Hier wird auch der genaue Testaufbau erklärt. In Abschnitt 5.3 werden die Ergebnisse der einzelnen Tests vorgestellt. Die Auswertung der Evaluation wird schließlich in Abschnitt 5.4 diskutiert.

5.1. Bewertungsmaß

Als Bewertungsmaß werden die *Precision*- und *Recall*-Werte der Ergebnisse benutzt. Sie stammen aus dem Gebiet des Information-Retrievals. Precision gibt die Wahrscheinlichkeit an, dass eine gefundene Regelverletzung relevant ist, also gefunden werden sollte. Es entspricht damit der Genauigkeit der Ergebnisse. Recall gibt die Wahrscheinlichkeit an, mit der eine relevante Regelverletzung gefunden wurde, also die Trefferquote.

Im Folgenden geben TP, FN und FP jeweils die Anzahl an TPs, FNs und FPs an. Dann gilt nach [25]:

$$Precision = \frac{TP}{TP + FP} \text{ und } Recall = \frac{TP}{TP + FN}$$

Die Precision entspricht also dem Anteil der gefundenen TPs an der Gesamtanzahl der gefundenen Regelverletzungen, und der Recall entspricht dem Anteil der gefundenen TPs an der Gesamtzahl vorhandener Regelverletzungen im analysierten Code. Nur einer der beiden Werte wäre wenig aussagekräftig. Der Recall ist 1.0 wenn alle Eingaben als Fehlerhaft „erkannt“ werden, unabhängig davon ob sie es sind. Die Precision spielt sehr konservativen Analysen zu, die nur wenige Verletzungen ausgeben, bei denen aber kein Zweifel besteht. Die Kombination der beiden Werte beugt dem vor, da sich ein Wert direkt auf den anderen auswirkt. Jedoch ist es einfacher, einen einzelnen Wert für Vergleiche zu benutzen. Dies ist mit dem F-Maß möglich, welches im einfachsten Fall das harmonische Mittel von Precision und Recall ist [26]:

$$F = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Je näher dieser Wert an 1.0 ist desto besser das Ergebnis.

5.2. Testaufbau

Als Datensatz wird die Juliet-Test-Suite verwendet [27]. Sie wurde vom National Institute of Standards and Technology (NIST) speziell zum Bewerten von statischen Analyse-Programmen entwickelt. Es gibt sie sowohl für Java als auch für C/C++. Die zum Verfassungszeitpunkt aktuellste Java-Version 1.2 besteht aus insgesamt 858 verschiedenen Testfällen. Jeder Testfall testet genau ein Sicherheitsproblem aus der Common Weakness Enumeration (vgl. [11]). Diese Sicherheitsprobleme werden CWE genannt. Jedes CWE hat eine Nummer und einen Titel. In einem Testfall können allerdings auch andere CWEs beiläufig vorkommen. Die Stellen, an denen ein CWE verletzt wird, sind entsprechend kommentiert. Von den meisten Testfällen gibt es mehrere Ausprägungen, die sich nur in der Komplexität des Daten- bzw. Kontrollflusses unterscheiden. Schließt man sie mit ein gibt es insgesamt 25477 Testfälle. Sie sind nach den betroffenen CWEs in insgesamt 112 Gruppen eingeteilt, wobei manche CWEs deutlich mehr Testfälle haben als andere. Die meisten Testfälle beinhalten sowohl gültige als auch ungültige Verwendungen.

Die Testfälle sind zum Teil deutlich einfacher als natürlicher Code. Laut der Dokumentation sind viele Testfälle bewusst der einfachste Code, der zu einer Regelverletzung führt. Dies führt zum Teil zu unrealistischen Testbedingungen. Jedoch hat auch natürlicher Code gravierende Nachteile. Damit eine Evaluation Sinn ergibt müssen die Fehler des Codes dokumentiert sein. Dies ist ein erheblicher Aufwand ohne Garantie auf Vollständigkeit und Korrektheit. Alle Regelverletzungen in den Juliet-Testfällen sind hingegen innerhalb des Codes kommentiert, sodass eine zeitaufwändige Dokumentation nicht extra nötig ist. Speziell für diese Arbeit bietet sie sich auch deshalb an, weil es von jedem Problem mehrere verschiedene Beispiele gibt, aus denen gelernt werden kann. In einem normalen Programm wäre es schwer, genügend Beispiele zu einem Problem zu finden, um den Lerneffekt zu evaluieren.

Auch wenn die Juliet-Testfälle vergleichsweise einfacher Natur sind ist es keinesfalls trivial, sie korrekt zu analysieren. Eine Studie von NASA-Angestellten [20] hat die Effizienz mehrerer Programme anhand der Juliet-Test-Suite miteinander verglichen. Das Ergebnis war, dass die Programme nur in sehr wenigen CWEs eine hohe Genauigkeit hatten. Einige CWEs wurden von keinem der getesteten Programme entdeckt. Keines der Programme wurde für sehr gut befunden. Anschließend wurden die selben Programme an dokumentierten Sicherheitslücken in Open-Source-Programmen evaluiert. Die Ergebnisse waren stimmig mit denen der Juliet-Test-Suite. Insgesamt wird deshalb angenommen, dass die Suite angemessen für die Evaluierung des Prototypen ist.

Aufgrund des Umfangs ist eine Auswahl der Testfälle nötig. Die Daten- und Kontrollflussvariationen unterscheiden sich zwischen den Testfällen nicht, sodass von jedem Testfall nur jeweils die grundlegendste Variation genommen wird. Da der Prototyp rein intraprozedural arbeitet erübrigt sich der Test auf den Variationen, da die meisten Variationen interprozedural sind. Von den 112 CWEs werden die ausgewählt, die zu den Top25 der Common Weakness Enumeration zählen. Darüber hinaus wurde ein CWE ausgewählt, von dem erwartet wurde, dass der Prototyp Schwierigkeiten mit ihm hat. Für jedes dieser CWE wird eine Regel erstellt. Die Analyse findet schließlich auf al-

CWE	Beschreibung	Top 25	Testfälle
CWE78	Command Injection	2	12
CWE89	SQL Injection	1	59 [1]
CWE129	Improper Validation of Array Index	-	72
CWE134	Uncontrolled Format String	23	18
CWE190	Integer Overflow or Wraparound	24	69

Tabelle 5.1.: Transaktionen der Beispielfunktionen

len CWEs statt, um eventuelle FPs zu evaluieren. Tabelle 5.1 zeigt die verwendeten CWEs.

Da der Lerneffekt evaluiert werden soll werden für jedes CWE vier Tests durchgeführt: eine Regel, die mit 5% der Beispiele angelernt wurde, einer Regel mit 10%, einer mit 15% und einer mit 20%. Falls der jeweilige Prozentsatz keiner ganzen Zahl von Testfällen entspricht wird aufgerundet. Da zwei der CWEs nur sehr wenige Beispiele haben, wird gegebenenfalls um ein weiteres Beispiel erhöht, falls zwei Prozentsätze die gleiche Anzahl Beispiele hätten. Hier sollte man erkennen können, welchen Effekt zusätzliche Beispiele auf die Regel haben. Die meisten CWEs enthalten Testfälle mit verschiedenen Kombinationen von Sinks und Sources. Hieran sollte erkennbar sein, ob die Aggregierung der verschiedenen Vorschriften funktioniert.

Jeder Testfall enthält eine `bad`-Funktion, die eine Regelverletzung darstellt. Darüber hinaus enthalten die meisten Testfälle noch eine oder mehrere `good`-Funktionen, welche die `bad`-Funktion soweit anpassen, dass sie keine Regelverletzung mehr darstellen. Deshalb kann es im selben Testfall zu einem TP und zu einem FP kommen.

5.3. Ergebnisse

Für jeden Test werden aus der Anzahl TPs, FPs und FNs die Precision, Recall und F-Maß-Werte berechnet. Jeder Test wird auf allen 857 [2] Testfällen ausgeführt.

Command Injection

Beispiele	TP	FP	FN	Precision	Recall	F-Maß
1 (8.3%)	1	0	11	1.0	0.08	0.16
2 (16.6%)	2	0	10	1.0	0.17	0.29
3 (25.0%)	3	0	9	1.0	0.25	0.40
4 (33.3%)	4	0	8	1.0	0.33	0.50

Tabelle 5.2.: Ergebnisse des CWE78 Command Injection

[1] Juliet enthält 60 SQL-Injection-Testfälle, einer davon konnte jedoch von PMD nicht gelesen werden
[2] Wie beschrieben konnte ein Testfall nicht von PMD gelesen werden

In dieser Gruppe hat sich kein Lerneffekt ergeben, es wurden nur die Testfälle gefunden, aus denen die Regel erstellt wurde. Betrachtet man die Testfälle lässt sich das damit begründen, dass sie kaum Gemeinsamkeiten haben. Jeder der 12 Testfälle zielt auf einen anderen Source ab, die sich nicht verallgemeinern lassen. Bei so diversifizierten Problemquellen ist also eine größere Menge an Beispielen zum Lernen nötig. Positiv ist, dass es keine FPs gab.

SQL Injection

Beispiele	TP	FP	FN	Precision	Recall	F-Maß
3 (5.1%)	6	0	53	1.0	0.10	0.18
6 (10.2%)	15	0	44	1.0	0.25	0.41
9 (15.3%)	30	0	29	1.0	0.51	0.67
12 (20.3%)	45	0	14	1.0	0.76	0.87

Tabelle 5.3.: Ergebnisse des CWE89 SQL Injection

Anders als bei der Command Injection hat sich hier ein sehr großer Lerneffekt ergeben. Es wurden deutlich mehr Testfälle als fehlerhaft erkannt, als zum anlernen der Regel verwendet wurden. Das zeigt, dass die Aggregierung der Vorschriften sehr gut funktioniert hat. 20% der Testfälle haben schon dazu ausgereicht, fast 75% der Fehler zu erkennen. FPs gab es keine, sodass das Ergebnis dieser Gruppe insgesamt als sehr positiv gewertet wird.

Improper Validation of Array Index

Beispiele	TP	FP	FN	Precision	Recall	F-Maß
4 (5.6%)	6	6	66	0.5	0.08	0.14
8 (11.1%)	12	12	60	0.5	0.17	0.25
11 (15.3%)	17	17	55	0.5	0.23	0.32
15 (20.8%)	25	25	47	0.5	0.35	0.41

Tabelle 5.4.: Ergebnisse des CWE129 Improper Validation of Array Index

In diesem Test zeigt sich, dass für die Behandlung von Arrays eine ausgefeiltere Analyse notwendig ist. Zwar gab es einen angemessenen Lerneffekt, bei dem die angelernten Beispiele auf andere Testfälle übertragen werden konnten, jedoch ist die Precision sehr niedrig. Für jeden gefundenen Testfall wurde sowohl das fehlerhafte als als auch das fehlerfreie Beispiel als fehlerhaft erkannt. Die Sanitizer haben hier also nicht funktioniert. Jedoch enthält die Analyse auch keinen speziellen Code für Arrays, sie werden behandelt wie jede andere Variable. Ein solches Ergebnis ist deshalb nicht überraschend. Mit einer fortgeschritteneren Implementierung der Analyse sollte sich das Ergebnis erheblich verbessern lassen.

Uncontrolled Format String

Beispiele	TP	FP	FN	Precision	Recall	F-Maß
1 (5.5%)	1	0	17	1.0	0.06	0.11
2 (11.1%)	4	0	14	1.0	0.22	0.36
3 (16.7%)	6	0	12	1.0	0.33	0.50
4 (22.2%)	8	0	10	1.0	0.44	0.62

Tabelle 5.5.: Ergebnisse des CWE134 Uncontrolled Format String

Die Ergebnisse dieser Gruppe entsprechen größtenteils denen der Command-Injection. Es gab einen geringen Lerneffekt, der jedoch bei weitem nicht mit der SQL-Injektion vergleichbar ist. Auch hier lässt sich das Ergebnis auf die starke Diversifizierung der Testfälle zurückverfolgen. Positiv ist wieder anzumerken, dass es keine FPs gab.

Integer Overflow

Beispiele	TP	FP	FN	Precision	Recall	F-Maß
4 (5.8%)	18	21	51	0.46	0.26	0.33
7 (10.1%)	21	25	48	0.46	0.30	0.37
11 (16.0%)	30	33	39	0.48	0.43	0.46
14 (20.3%)	36	41	33	0.47	0.52	0.49

Tabelle 5.6.: Ergebnisse des CWE190 Integer Overflow

Die Ergebnisse dieses Tests haben positive und negative Seiten. Einerseits gab es einen ordentlichen Lerneffekt. Schon mit wenigen Beispielen ließ sich ein Vielfaches an Verletzungen finden. Andererseits ist die Precision aufgrund der vielen FPs niedrig. Die FPs der vierten Zeile verteilen sich wie folgt auf verschiedene CWEs:

- 22 FPs wurden in der Integer-Underflow-Kategorie gefunden, davon 11 in `bad` und 11 in `good`
- 16 FPs wurden in der Division-by-Zero-Kategorie gefunden, 8 in `bad` und 8 in `good`
- 3 FPs stammen aus `good`-Funktionen der Integer-Overflow-Kategorie

Zunächst wurden innerhalb der Integer-Overflow-Testfälle nur sehr wenige FPs gefunden. Das zeigt, dass die Sanitizer hier zumindest größtenteils funktionieren. Ein Großteil der restlichen FPs entstanden durch zu starke Verallgemeinerung der Operatoren. Durch die Verallgemeinerung führten auch Divisionen und Subtraktionen zu einer Verletzung, obwohl das nicht vorgesehen war. Hier wäre eine differenziertere Verallgemeinerung notwendig, oder eine, die iterativ aus FPs lernen kann. Die verbleibenden FPs traten bei Multiplikation der Form $x \cdot (-x)$ auf, welche zu einem Integer-Underflow führen können. Auch hier ist eine tiefergehende Unterscheidung während der Analyse notwendig.

Zum Vergleich wurde der Test mit deaktivierter Verallgemeinerung wiederholt.

Beispiele	TP	FP	FN	Precision	Recall	F-Maß
4 (5.8%)	10	6	59	0.63	0.14	0.24
7 (10.1%)	13	8	56	0.62	0.19	0.29
11 (16.0%)	22	13	47	0.65	0.32	0.43
14 (20.3%)	28	16	41	0.64	0.41	0.50

Tabelle 5.7.: Ergebnisse des CWE190 Integer Overflow ohne Verfeinerung

Ohne Verallgemeinerung gibt es deutlich weniger TPs, woran zu erkennen ist, dass die Verallgemeinerung zumindest das gewünschte Ziel erfüllt. Es gibt jedoch auch deutlich weniger FPs. Alle auftretenden FPs sind von der Form $x \cdot (-x)$, welches eine inhärente Einschränkung der Analyse-Implementierung zu sein scheint. Das F-Maß ist in den ersten drei Fällen deutlich niedriger als mit Verallgemeinerungen. Nur mit 20% der Testfälle als Lernmenge sind beide Varianten gleichauf. Die Verallgemeinerung hatte also durchaus einen positiven Effekt. Die erhöhte Anzahl an FPs wird durch mehr TPs ausgeglichen. Ein FN wiegt generell schwerer als ein FP, weshalb die Verallgemeinerung trotz allem als Verbesserung eingestuft wird. Jedoch sind ihre Schwächen hinsichtlich der zu starken Verallgemeinerung klar ersichtlich.

5.4. Diskussion

Insgesamt sind die Ergebnisse der Evaluation durchaus positiv. Insbesondere die Aggregierung und Kombination verschiedener Vorschriften hat sehr gut funktioniert, wie die SQL-Injection-Gruppe zeigt. Die Verallgemeinerung wurde hauptsächlich im Integer-Overflow-Test genutzt. Hier hat sich gezeigt, dass sie Potential hat die Analyse zu verbessern. In der gegenwärtigen Form ist sie jedoch nicht differenziert genug und führt zu einer vermehrten Anzahl Fehlerkennungen. Die Analyse selbst muss ebenfalls noch ausgebaut werden. Der Prototyp beschränkt sich auf die reine Erkennung der Befehlsfolgen, ohne, dass Wissen über mögliche Variablenwerte einfließen kann. Unter Nutzung dieser Werte sollte sich die Verlässlichkeit deutlich erhöhen lassen. Sehr viel Potential besteht auch darin, die Erkenntnisse anderer Regeln in eine Regel einfließen zu lassen. Viele der Testfälle verwenden ähnliche Sinks und Sources, jedoch kann bisher jede Regel nur aus ihren eigenen Beispielen lernen. Ohne diese Einschränkung könnte mit weniger Beispielen eine höhere Trefferquote erzielt werden. Jedoch muss auch dies differenziert betrachtet werden, da eine Source einer Regel nicht zwangsläufig in anderem anderen Kontext ebenfalls als Source gelten muss. Hier wäre also eine Abwägung nötig.

Für die verwendete SPMF-Bibliothek hat sich gezeigt, dass sie nicht gut skaliert. Sie kann nur einen CPU-Kern nutzen, und aufgrund der unzureichenden Generierung der Association Rules arbeitet sie sehr ineffizient. Schon Transaktionen mit unter 20 Items benötigten mehrere Stunden. Durch eine angemessene Implementierung sollte der Zeitaufwand auf ein Bruchteil davon reduziert werden können, wie etwa [18] zeigte.

6. Fazit

In dieser Arbeit wurde gezeigt, dass es möglich ist, auf leichtgewichtige Art Regeln für die statische Codeanalyse zu erstellen. Es wurde eine Annotationssprache entwickelt, die es erlaubt, eine Vielzahl von Regeln direkt anhand von Beispielcode zu definieren. Mit diesen Annotationen wurden die Abhängigkeiten innerhalb des Codes berechnet und die minimale Codesequenz bestimmt, die zur Regelverletzung notwendig ist. Anschließend wurde ein Verfahren vorgeschlagen, mit dem die Regeln verallgemeinert werden können. Für die eigentliche Analyse wurde eine Oberregel für ein Open-Source-Framework erstellt, welche die erstellten Regeln als Parameter nimmt. Zuletzt wurde eine prototypische Umsetzung Anhand einer Test-Suite evaluiert.

Die Ergebnisse dieser Evaluation zeigten, dass das Vorgehen funktioniert. Mithilfe einer kleinen Teilmenge der Testfälle war es in den meisten Fällen möglich, eine Übertragung auf vorher unbekannte Testfälle durchzuführen. Dies funktionierte insbesondere für die wichtige SQL-Injektion sehr zuverlässig. Jedoch stellte sich heraus, dass die Analyse weiter ausgebaut werden muss, um zuverlässig vom Kontext abhängige Fehler wie den Integer-Overflow zu erkennen. Da dies jedoch keine grundsätzliche Einschränkung der entwickelten Methode ist, sondern lediglich durch die konkrete Implementierung des Analyseverfahrens verursacht wird, besteht hier definitiv noch Potential. Mit ausgefeilteren Analysetechniken, die dem aktuellen Stand der Technik entsprechen, sollten sich diese Probleme ausräumen lassen.

Letztendlich legt diese Arbeit nur den Grundstein für die direkte Regelableitung aus Beispielcode. Sie zeigt die Plausibilität des Vorgehens und bietet Ansätze für weitere Entwicklungen. Für eine zuverlässige Nutzung ist jedoch insbesondere die Analyse noch weiter auszubauen.

6.1. Ausblick

Es gibt naturgemäß eine Vielzahl von Möglichkeiten, wie die entwickelte Methode ausgebaut werden kann. An dieser Stelle soll eine Auswahl davon diskutiert werden.

Die entwickelte Annotationssprache wurde bewusst klein gehalten, um den Umfang einzugrenzen und sie leicht erlernbar zu halten. In künftigen Entwicklungen ist ein Ausbau der Annotationssprache denkbar. Sie könnte etwa um weitere Parameter erweitert werden, mit denen sich Bedingungen angeben lassen. Diese könnten von der Analyse ausgewertet werden. Ebenso ist es denkbar, dass die Datenfluss-Annotationen ausgebaut werden, um auch Aktionen zu erfassen, die per Referenz an Funktionen übergebene komplexe Objekte betreffen.

Besonders viel Entwicklungspotential bietet die Regelverfeinerung. Diese wurde eher beispielhaft eingeführt. Sie ist bisher auf einen Fluss durch die Methoden beschränkt und verallgemeinert völlig undifferenziert. Um erstere Einschränkung aufzuheben sind Algorithmen nötig, die gut mit einer wachsenden Zahl an Pfaden skalieren. Für letzteres müssen weitere Informationen erfasst und verwendet werden. Auch muss es möglich sein, aus FPs zu lernen. Darüber hinaus sind natürlich auch noch weitere Verfeinerungsalgorithmen denkbar.

Wie bereits angesprochen ist die Implementierung der Analyse im Prototyp der bisher größte Schwachpunkt. Hier gibt es aber auch am meisten Potential. Die Möglichkeiten, mit denen sie verbessert werden kann, sind nach oben offen. Es gibt neben der offensichtlichen Erweiterung auf interprozeduralen Analysen eine Vielzahl von Techniken, mit denen bessere Ergebnisse möglich sind: Spezielle Behandlung von Arrays, das Bilden und Auswerten von Constraints, etc.

Die Übertragung der Analyse auf kompilierten Bytecode bietet ebenfalls interessante Möglichkeiten und Herausforderungen. In der jetzigen Form wird nur mit dem Syntax-Baum des Source-Codes gearbeitet. Der Bytecode entspricht in seiner Form jedoch eher Assembler-Code, sodass hier eine Abbildung der Statements auf die entstehenden Befehle nötig ist. Dies erfordert eine gänzlich andere Herangehensweise bei der Analyse.

A. Inhalt der CD

Die beiliegende CD enthält folgenden Inhalt:

- Diese Ausarbeitung als PDF-Dokument
- Die Ergebnisse der Evaluation
- Den Source-Code des entwickelten Prototypen
- Eine Textdatei mit Verwendungshinweisen

Literaturverzeichnis

[1] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2009.

[2] E. Dalci and J. Steven, “A framework for creating custom rules for static analysis tools,” *Proc. Static Analysis Summit*, pp. 3–8, 2006.

[3] B. Chess and J. West, *Secure programming with static analysis*. Pearson Education, 2007.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.

[5] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, “Experiences Using Static Analysis to Find Bugs,” *Software, IEEE*, vol. 25, no. 5, pp. 22–29, 2008.

[6] FindBugs, “Find bugs in java programs.” `http://findbugs.sourceforge.net/`. Letzter Zugriff: 14.08.2014.

[7] PMD. `http://pmd.sourceforge.net/`. Letzter Zugriff: 14.08.2014.

[8] CLang, “CLang static analyzer.” `http://clang-analyzer.llvm.org/`. Letzter Zugriff: 14.08.2014.

[9] W. Landi, “Undecidability of static analysis,” *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[10] V. Livshits and M. Lam, “Finding Security Vulnerabilities in Java Applications with Static Analysis.,” *Usenix Security*, 2005.

[11] Common Weakness Enumeration, “Top 25 most dangerous software errors.” `http://cwe.mitre.org/top25/index.html`. Letzter Zugriff: 14.08.2014.

[12] G. Piateski and W. Frawley, *Knowledge Discovery in Databases*. Cambridge, MA, USA: MIT Press, 1991.

[13] D. L. Olson and D. Delen, *Advanced data mining techniques*. Springer, 2008.

[14] J. Pei, J. Han, and R. Mao, “CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets.,” *ACM SIGMOD workshop on research issues in data mining and knowledge discovery*, vol. 4, no. 2, pp. 21–30, 2000.

[15] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, “Discovering frequent closed itemsets for association rules,” *Database Theory—ICDT’99*, 1999.

[16] M. Martin, B. Livshits, and M. Lam, “Finding application errors and security flaws using PQL: a program query language,” *ACM SIGPLAN Notices*, pp. 365–383,

2005.

[17] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, “Context-sensitive program analysis as database queries,” *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS ’05*, pp. 1–12, 2005.

[18] Z. Li and Y. Zhou, “PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code,” *ACM SIGSOFT Software Engineering Notes*, pp. 306–315, 2005.

[19] D. Evans and D. Larochelle, “Improving security using extensible lightweight static analysis,” *software, IEEE*, vol. 19, no. 1, pp. 42–51, 2002.

[20] K. Goseva-Popstajanova and A. Perhinschi, “Using static code analysis tools for detection of security vulnerabilities.” `http://www.nasa.gov/sites/default/files/01-11_using_static_code_analysis_tools_0.pdf`. Letzter Zugriff: 14.08.2014.

[21] C. Christopher, “Evaluating Static Analysis Frameworks,” *Analysis of Software Artifacts*, pp. 1–17, 2006.

[22] N. Rutar, C. Almazan, and J. Foster, “A Comparison of Bug Finding Tools for Java,” *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pp. 245–256, 2004.

[23] ANTLR. `http://www.antlr.org/`. Letzter Zugriff: 14.08.2014.

[24] P. Fournier-Viger, “SPMF: An Open-Source Data Mining Library.” `http://www.philippe-fournier-viger.com/spmf/`. Letzter Zugriff: 14.08.2014.

[25] J. Davis and M. Goadrich, “The relationship between precision-recall and roc curves,” in *Proceedings of the 23rd international conference on Machine learning*, pp. 233–240, ACM, 2006.

[26] J. Makhoul, F. Kubala, R. Schwartz, R. Weischedel, *et al.*, “Performance measures for information extraction,” in *Proceedings of DARPA broadcast news workshop*, pp. 249–252, 1999.

[27] National Institute of Standards and Technology, “Test suites.” `http://samate.nist.gov/SARD/testsuite.php`. Letzter Zugriff: 14.08.2014.

Abbildungsverzeichnis

2.1. AST eines simplen C-Programms (aus [3], Seite 75) 4

3.1. Schritte der Regelableitung 10
3.2. Vorgängergraph von Listing 3.4 18
3.3. Abhängigkeitsgraph von Listing 3.4 19
3.4. Automat der Source-Vorschrift aus Listing 3.5 30
3.5. Automat der Sink-Vorschrift aus Listing 3.6 30

4.1. Vereinfachtes Zusammenspiel der Klassen während der Analyse 43

Liste der Algorithmen

1. Bestimmung des Vorgängergraphen 16
2. Bestimmung des Abhängigkeitsgraphen 18
3. Bestimmung der Tainted-Variablen 21
4. Vereinigung 28

5. Sammeln der Oberklassen 40

Listings

2.1. Beispiel für SQL-Injektion 6

3.1. Beispiel-Code, aus dem eine Regel abgeleitet werden soll (aus der Juliet-Test-Suite [20], leicht angepasst) 11
3.2. Beispiel für `@required` 13
3.3. Annotierte Version von Listing 3.1 15
3.4. Beispielcode für den Vorgängergraphen 17
3.5. Vereinfachte Source-Vorschrift 23
3.6. Vereinfachte Sink-Vorschrift 24
3.7. Beispielfunktionen für die Regelverfeinerung 25
3.8. Verallgemeinertes Beispiel aus Listing 3.7 28

Tabellenverzeichnis

3.1. Verwendete Annotationen 12
3.2. Parameter von `@match` 13
3.3. Durchführung auf Listing 3.4 17
3.4. Abhängigkeiten der Variablen von Listing 3.4 19
3.5. Transaktionen der Beispielfunktionen aus Listing 3.7 26
3.6. Frequent Itemsets der Transaktionen aus Listing 3.5 27
3.7. Association Rules der Transaktionen aus Listing 3.5 27
3.8. Schritte der Analyse des Codes aus Listing 3.1 32
3.9. Konfigurationen des Source-Automaten aus Abbildung 3.4 32
3.10. Konfigurationen des Sink-Automaten aus Abbildung 3.4 33

5.1. Transaktionen der Beispielfunktionen 46
5.2. Ergebnisse des CWE78 Command Injection 46
5.3. Ergebnisse des CWE89 SQL Injection 47
5.4. Ergebnisse des CWE129 Improper Validation of Array Index 47
5.5. Ergebnisse des CWE134 Uncontrolled Format String 48
5.6. Ergebnisse des CWE190 Integer Overflow 48
5.7. Ergebnisse des CWE190 Integer Overflow ohne Verfeinerung 49

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 15.08.2014

____
Pascal Elster