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

Die Nutzung von Callstack Informationen für die Security Code Clone Detection

Using Callstack Information for the Security Code Clone Detection

Bachelorarbeit

im Studiengang Informatik

von

John Matthes

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Dr. Jil Klünder
Betreuer: M. Sc. Fabien Patrick Viertel

Hannover, 10.08.2019

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 10.08.2019

John Matthes

Zusammenfassung

Da Softwareanwendungen aus unserem Leben nicht mehr wegzudenken sind, stellen Sicherheitslücken in eben diesen enorme Bedrohung dar. Sie können zu finanziellen Schäden führen, aber auch dem Menschen direkt schaden.
Im Zentrum dieser Bachelorarbeit steht der Ausbau eines am Fachgebiet Software Engineering erstellten *Code Clone Detectors*, der Entwicklern dabei behilflich sein soll, Sicherheitslücken zu vermeiden. Dieser soll, um die Verarbeitung von Callstack Informationen, die Angaben über Sicherheitslücken eines Programmes zu dessen Laufzeit enthalten, erweitert werden. Außerdem soll eine Funktion implementiert werden, die erkannte Sicherheitsrisiken automatisch behebt oder, falls dies nicht gelungen ist, dem Nutzer die nötigen Informationen gibt, um dies selbst zu tun.
Außerdem hat der Nutzer die Möglichkeit, sollte es mit der aktuellen Schwachstellen-Datenbank und den Callstack Information nicht zu einem auffinden eines *Code Clones* kommen, den betreffenden Codeabschnitt der Datenbank hinzuzufügen. Somit kann diese mit weiteren, noch unbekannten Sicherheitslücken erweitert werden. Des Weiteren wurde eine Git-Schnittstelle implementiert, die es dem Nutzer ermöglicht, die erweiterte Datenbank hochzuladen und sie somit anderen zur Verfügung zu stellen.

Abstract

Since software applications are an inherent part of our lives, security vulnerabilities constitute a vast threat. They can cause financial damage as well as harm humans directly.
This bachelor thesis is about extension of a *Code Clone Detectors*, which was developed at the Institute for Software Engineering and which helps developers to note and avoid these vulnerabilities. This program is extended by the processing of Callstack Information, which contain data of a specific program for its runtime. Furthermore a function, which tries to fix located vulnerabilities automatically, is implemented. If this attempt fails the user is given the necessary information to do this on his own.
In addition, the user has the opportunity to enrich the Vulnerability-Database with new Vulnerable-Code, if the *Security Code Clone Detector* is unable to find a *Code Clone* to specific Callstack Information. If this happens, it may be that the Callstack Information contain data of a, yet unknown, security risk, so it's highly valuable to update and extend the existing database. To maximize the benefits of this feature, a Git-Interface was implemented, to provide the new information to other users.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Da Software ein fester Bestandteil in allen Bereichen des Lebens geworden ist, ist die Sicherheit dieser ein zentraler und existenziell wichtiger Bereich der Softwareentwicklung geworden. Wie Geschehnisse aus jüngster Vergangenheit zeigen, können bereits kleine Sicherheitslücken in Systemen zu enormen Schäden führen, zum Beispiel in Krankenhäusern.
Deshalb wurde am Fachgebiet Software Engineering von W. Brunotto der *Security Code Clone Detector* [1] entwickelt. Dieser soll den Nutzer in Form eines Eclipse-Plugins darin unterstützen, bereits bekannte Sicherheitsrisiken zu erkennen und diese wiederum zu beheben. Dies geschieht auf Grundlage einer Datenbank, die Informationen und Beispiele zu eben diesen Risiken enthält. Der *Security Code Clone Detector* betrachtet allerdings den reinen Source-Code eines Projektes oder einer Datei, sodass es sinnvoll ist, diesen so zu erweitern, dass speziell nach weiteren Problemen gesucht werden kann. Diese Bachelorarbeit beschäftigt sich mit der Erweiterung des *Security Code Clone Detectors* um die Verarbeitung von Callstack Informationen, die Daten zu Schwachstellen zur Laufzeit eines Programm enthalten, sowie dem Versuch, diese Fehler automatisch zu beheben.

1.1 Problemstellung

Da der bisherige *Security Code Clone Detector* nur den Inhalt von Quelldateien überprüft, werden Schwachstellen, die erst zur Laufzeit des Programmes auftreten, möglicherweise nicht erkannt. Deshalb sollen nun auch Callstack Informationen verarbeitet werden, um dieses Problem zu lösen. Des Weiteren beschränken sich die Datenbank und somit auch die Möglichkeiten des *Security Code Clone Detector* auf Daten, die erst einmal im Internet gefunden und extrahiert werden müssen. Somit ist es sinnvoll, dem Nutzer die Möglichkeit zu geben, neue Schwachstellen selbst der Datenbank hinzuzufügen, um diese zu erweitern. Außerdem ist es bisher sehr mühsam, die gefundenen Schwachstellen zu beheben, weshalb eine Funktion implementiert wurde, die

versucht, den Fehler automatisch zu beheben.

1.2 Lösungsansatz

Der Lösungsansatz für die Verarbeitung von Callstack Informationen war es, den bisherigen Code so anzupassen, dass nicht mehr das gesamte Projekt oder eine zuvor ausgewählte Datei untersucht wird, sondern die gegebenen Informationen genutzt werden, um die betreffende Stelle/Datei automatisch zu lokalisieren. Dazu wurden verschiedene String-Operationen angewandt, sodass dem Programm Daten zum Pfad sowie zur gesuchten, fehlerhaften Datei/Methode zur Verfügung gestellt werden konnten.
Für die Datenbank wurde ein Controller implementiert, der alle Funktionen rund um die Datenbankverwaltung beinhaltet. Dazu gehören Vebindungsaufbau und -abbruch, das Prüfen des Vorhandenseins eines bestimmten Eintrages sowie das Hinzufügen neuer Einträge. Darüber hinaus wurde das Programm um eine Git-Schnittstelle erweitert, die dafür Sorge trägt, das Repository auf Updates zu prüfen und, wenn gewünscht, die erweiterte Datenbank wiederum hochzuladen.
Um den Versuch zu unternehmen, gefundene Schwachstellen automatisch zu beheben, wurden drei Ansätze miteinander kombiniert. Zum einen werden sogenannte *Contracts* genutzt, die Informationen über die aktuelle Zeile im Code sowie über die Vorherige und die Nachfolgende enthalten. Des Weiteren wird die externe Library *diffmatchpatch* genutzt, die den Inhalt der vorhandenen Datenbankeinträge vergleicht, um die Unterschiede zwischen Schwachstelle und dem hinterlegten Fix, falls vorhanden, zu ermitteln. Zu guter Letzt werden diese Informationen miteinander kombiniert, um mit Hilfe der Levenshtein-Distanz den Ort im Quellcode zu lokalisieren, an dem der Fix eingefügt werden muss.

1.3 Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert: In Kapitel 2 werden die für das Verständnis wichtigen Grundlagen näher erläutert. Kapitel 3 beinhaltet die detaillierte Beschreibung des Konzeptes sowie der Funktionsweise des Programms. In Kapitel 4 wird auf die Implementierung der einzelnen Programmkomponenten eingegangen. Kapitel 5 befasst sich mit den Ergebnissen der automatischen Schwachstellenbehebung und deren Evaluation. In Kapitel 6 wird diese Arbeit mit verwandten Arbeiten verglichen, um den derzeitigen Stand in diesem Bereich der Informatik zu ermitteln und die Arbeit darin einzuordnen. Kapitel 7 beinhaltet schließlich einen Überblick über die Ergebnisse der Arbeit sowie einen Ausblick. Des Weiteren wird auf dessen Validität eingegangen.

Kapitel 2

Grundlagen

In diesem Kapitel wird auf die verschiedenen Grundlagen eingegangen, die für das Verständnis dieser Arbeit wichtig sind. Dazu gehören die *Code Clone Detection*, die *SQLite-Datenbank* und die *Callstack Informationen*.

2.1 Code Clone

Die grundlegende Problematik, mit der sich diese und die ihr zugrunde liegenden Arbeiten beschäftigen, sind die sogenannten *Code Clone*. Dabei handelt es sich im Allgemeinen um sich ähnelnde Abschnitte von Quellcode die vorwiegend durch *copy & paste* entstehen. Problematisch hierbei ist, dass dies nicht nur zu einer Minderung der Programmqualität beiträgt, sondern auch Fehler und Schwachstellen einfach immer weiter wiederholt und übernommen werden. Dadurch stellen sie eine nicht unbeachtliche Gefahr für Softwareanwendungen dar und sollten daher um jeden Preis vermieden werden.
Allerdings ist zu beachten, dass es verschiedene *Clone-Typen* gibt, die unterschiedlich schwer zu erkennen sind. Als Übersicht hierfür dient, in Anlehnung an die Arbeit von F. Viertel et al. [10] Figure 2, Tabelle 2.1, die jeweils kurze, prägnante Codebeispiele enthält. Dabei ist anzumerken, dass es noch einen *Typ-4 Clone* gibt, der sich zwar semantisch, aber kaum syntaktisch ähnelt, sodass dieser kaum zu erkennen ist und somit in dieser Arbeit nicht betrachtet wird.
Bei einem *Typ-1 Clone* handelt es sich um einen Abschnitt, der bis auf Kommentare und Struktur identisch mit einem anderen ist. Dies ist der am einfachsten zu erkennende *Clone-Typ*, da es kaum syntaktischen Unterschiede gibt und die Semantik gleich bleibt.
Ein *Typ-2 Clone* hingegen unterscheidet sich in Variablen-, Literal-, oder Funktionsnamen, wodurch zwar die Semantik nicht geändert wird, aber die Syntax nun anders ist. Dies erschwert das Auffinden im Vergleich zu *Typ-1*, da nicht mehr einfach nach einer kompletten Kopie eines Abschnittes gesucht

werden kann.
Der *Typ-3 Clone* ist der, bis auf den *Typ-4 Clone*, der hier aus oben genannten Gründen nicht betrachtet wird, am schwersten aufzufindende Typ, da er zu den Eigenschaften von *Typ-2 Clones* und *Typ-1 Clones* noch hinzugefügte oder gelöschte Anweisungen enthält.

	Abschnitt A	**Abschnitt B**

Typ-1	int index = 0;
	int count = index * index;
		int index = 0; //comment
		
		int count = index * index;

Typ-2	int index = 0;
	int count = index * index;
		int i = 0; //comment
		
		int c = i * i;

Typ-3	int index = 0;
	int count = index * index;
		int i = 0; //comment
		double d = i * 0.5
		int c = i * i;

Tabelle 2.1: Vergleich von Typ-1 bis Typ-3 Clones

Um dem Risiko der *Code Clones* entgegenzuwirken gibt es Programme, die sich mit dem Auffinden dieser beschäftigen. Eine kurze Erklärung einer solchen Anwendung befindet sich im nachfolgenden Abschnitt 2.2.

2.2 Code Clone Detection

Um diese Arbeit in Gänze verstehen zu können, ist es zunächst noch wichtig, kurz auf das dieser Arbeit zugrunde liegende Programm, einen *Code Clone Detector* von W. Brunotte [1], einzugehen und dessen Funktion zu beschreiben.
Ein *Code Clone Detector* versucht, auf Grundlage der ihm zur Verfügung stehenden Daten, sich wiederholende oder ähnliche Abschnitte im Quellcode einer Datei zu lokalisieren. Da diese meist auf einer Form von Normalisierung basieren, ist es ihnen somit trotz abweichender Variablen-, Literal- und Funktionsnamen möglich, einen *Code Clone* zu lokalisieren. Allerdings nimmt die Erfolgswahrscheinlichkeit mit zunehmenden Abweichungen ab.
Die zugrunde liegenden Daten sind in Form einer *Datenbank* vorhanden. Diese beinhaltet die nötigen *Code-Snippets*, die genutzt werden, um aufgrund dieser einen entsprechenden *Code Clone* zu lokalisieren. Ohne die Codebeispiele wäre es einem *Code Clone Detector* unmöglich, dies zu tun. Somit hängt die Wahrscheinlichkeit des Auffindens eines *Code Clone* maßgeblich vom Umfang dieser Datenbank ab.
Der *Security Code Clone Detector* von Wasja Brunotte [1] wurde als Eclipse-Plugin entwickelt und hilft dem Nutzer während des Programmierens, indem er ihn auf vorhandene *Code Clones* hinweist und somit mögliche

Sicherheitslücken vermieden werden können. Dies geschieht auf Grundlage einer *SQLite-Datenbank*, die eine große Menge an *Code-Snippets* enthält, welche mit Hilfe verschiedener Algorithmen, unter anderem von Yannik Evers [2] im Zuge seiner Bachelorarbeit, von unterschiedlichen Websites extrahiert worden sind.
Abbildung 2.1 ist eine Anlehnung an eine Grafik aus der Arbeit von Brunotte [1] und stellt den Ablauf des von ihm entwickelten Plugins schematisch dar.

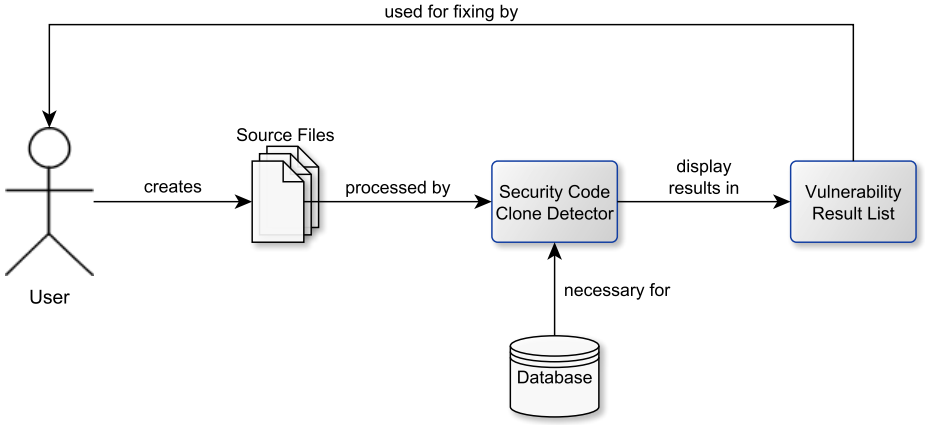

Abbildung 2.1: Schematischer Ablauf des Security Code Clone Detectors

2.3 SQLite-Datenbank

Dieser Abschnitt enthält eine kurze Erklärung zu *SQLite-Datenbanken* und deren Vorteilen, um das allgemeine Verständnis dieser Arbeit zu verbessern. Es handelt sich um ein Datenbanksystem, das, im Gegensatz zu *Client/Server SQL-Datenbanken*, einen lokalen Datenspeicher zur Verfügung stellt. Dabei zielt es besonders auf *efficiency, reliability, portability* und *accessibility* ab, sodass es ohne weiteres systemübergreifend verwendet werden kann und sich somit perfekt für die hier zu lösende Problematik eignet.
Abbildung 2.2 zeigt einen beispielhaften Aufbau einer solchen Datenbank. Es ist zu erkennen, dass es zunächst eine Menge von Tabellen gibt, die jeweils wiederum aus einer Menge von Elementen bestehen. Dabei erhält jede Tabelle einen Bezeichner, der es einem Nutzer ermöglicht gezielt auf einzelne Tabellen und deren Inhalt zuzugreifen. Außerdem wird festgelegt, welche Informationen die einzelnen Elemente enthalten können.
Welche Informationen dies sind, wird einfach über einen Variablennamen und

einem dazugehörigen Variablentyp angegeben. Somit kann man den Inhalt der einzelnen Elemente genau seinen Bedürfnissen anpassen. Außerdem gibt es die Möglichkeit, eine Variable als sogenannten *Primary Key* zu kennzeichnen. Dies bewirkt, dass der Wert dieser Variable zur eindeutigen Erkennung eines Elements genutzt werden kann, um dieses, unter anderem, aus der Datenbank extrahieren zu können. Eine weitere wichtige Eigenschaft einer Variable kann der *Foreign Key* sein. Dieser gibt an, dass der abgespeicherte Wert auf ein Element einer anderen Tabelle verweist, sodass Verbindungen zwischen Tabellen und deren Inhalt aufgebaut werden können.

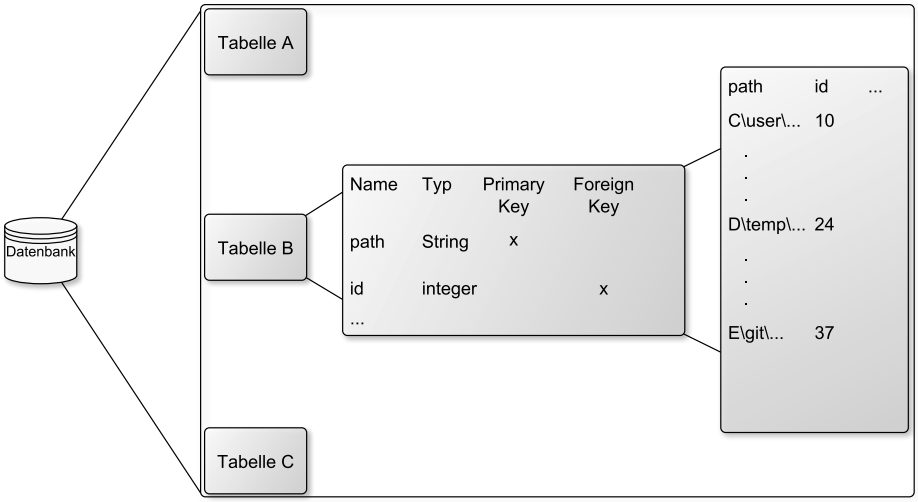

Abbildung 2.2: Schematischer Aufbau einer SQLite-Datenbank in Anlehnung an SQLiteStudio [8]

2.4 Callstack Informationen

In diesem Abschnitt werden die *Callstack Informationen* erläutert, die im geänderten *Security Code Clone Detector* dazu genutzt werden sollen, um eine Schwachstelle zu lokalisieren. Diese Informationen sind das Ergebnis des *Monitorings*, der Überwachung eines Programms zu dessen Laufzeit, das in der Lage ist, ungewöhnliches Programmverhalten zu erkennen. Somit können sie die *Code Clone Detection* um das Auffinden von Schwachstellen zur Laufzeit ergänzen, was wiederum in erhöhter Programmsicherheit resultieren kann.
Allerdings ist der Aufbau der *Callstack Informationen* nicht für den *Code Clone Detector* geeignet, sodass diese zunächst verarbeitet und in ihre Einzelinformationen zerlegt werden müssen. Erst nachdem diese Schritte

vollzogen wurden, ist es möglich, anhand der extrahierten Daten die entsprechende Schwachstelle zu suchen.

Kapitel 3

Konzept

Dieses Kapitel beschäftigt sich mit dem Konzept des *Runtime-Security-Code-Clone-Detectors* oder kurz *RSCCD*. Dabei lässt es sich in die Abschnitte Verarbeitung der Callstack Informationen, Erweiterung der Datenbank, Verwendung von Git und Versuch der automatischen Schachstellenbehebung unterteilen. Die Erweiterung der *Security-Code-Clone-Detector* Funktionalität um diese Abschnitte wird in Abbildung 3.1, angelehnt an F. P. Viertel et al. [10], dargetsellt.

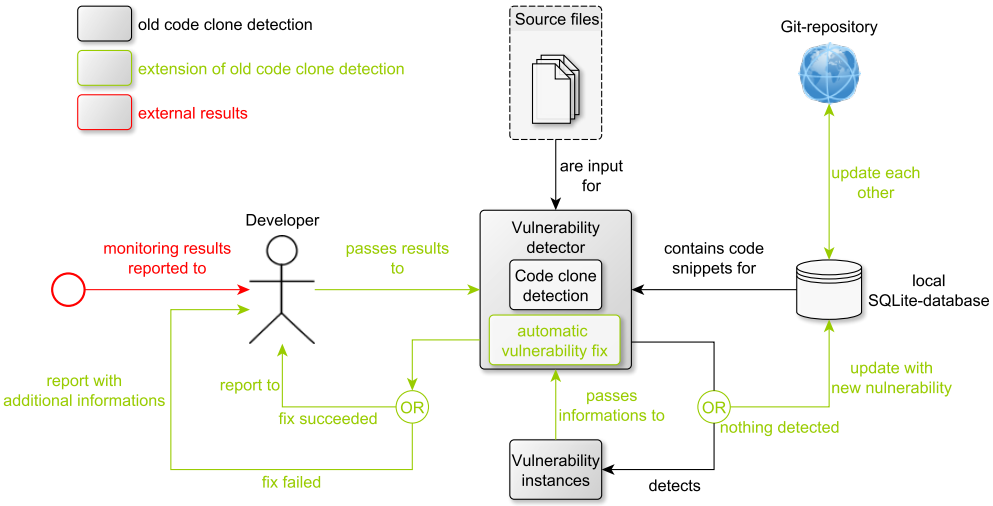

Abbildung 3.1: Schematische Darstellung des Programmablaufs

3.1 Config Datei

Bisher hatte der Nutzer die Möglichkeit, verschiedene Einstellungen des Plugins über das dazugehörige Menü zu ändern. Da das Programm nun allerdings über die Konsole gestartet werden soll, ist es nötig, dem Nutzer eine einfach zu verwendene Alternative anzubieten, um die *Usability* weiterhin aufrecht zu erhalten. Zu diesem Zweck soll eine Config-Datei angelegt werden, die in Form einer Art von Textdatei angelegt wird. Diese beinhaltet eine Reihe von notwendigen Attributen und den dazugehörigen Werten. Abbildung 3.2 zeigt zunächst eine Übersicht der verschiedenen Einstellungsmöglichkeiten.

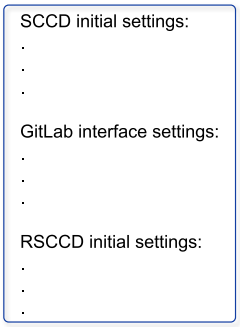

Abbildung 3.2: Aufbau der Config Datei

Zur Verbesserung der Übersichtlichkeit sollen die einzelnen Einstellungen in Funktionsgruppen unterteilt werden. Dazu gehören die für die *Code Clone Detection* nötigen Einstellungen, die aus dem Plugin übernommen wurden, die Einstellungen für die in Abschnitt 3.4 erläuterte Git-Schnittstelle und zu guter Letzt die Einstellung für die Verarbeitung der in Abschnitt 3.2 beschriebenen *Callstack Informationen.*
Der Nutzer hat nun die Möglichkeit, diese Einstellungen nach Belieben an seine Bedürfnisse anzupassen und abzuändern, sollte dies nötig sein. Das Programm ließt dann jeweils die benötigten Werte aus oder weist den Nutzer auf unzulässige Werte hin.

3.2 Verarbeitung der Callstack Informationen

Der erste Schritt des *RSCCD* ist es, die gegebenen *Callstack Informationen* so zu verarbeiten, dass der *Code Clone Detector* sie zur Suche verwenden kann. Dafür werden verschiedene Operationen genutzt, um den Eingabestring in seine Einzelteile zu zerlegen. Eine schematische Darstellung eines solchen Vorgangs ist in Abbildung 3.3 dargestellt.

Wie man anhand dieses Beispiels erkennen kann, besitzen die *Callstack*

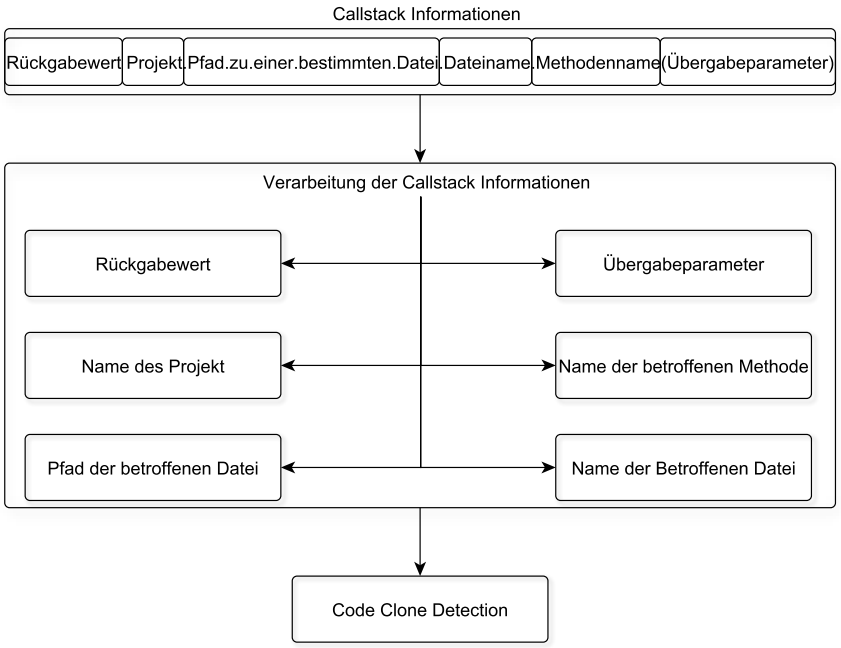

Abbildung 3.3: Schematische Verarbeitung der Callstack Informationen

Informationen eine Reihe wichtiger Daten, die allerdings in ihrem initialen Zustand nicht genutzt werden können. Nach der Extraktion ist der *Code Clone Detecor* jedoch im Besitz aller nötigen Informationen für eine Schwachstellensuche.
Diese Informationen sind der Pfad und der Name der Datei sowie der Methodenname zuzüglich dessen Return-Typs und Übergabeparameter. Anhand dieser Daten ist es nun möglich, zunächst die Datei aufzuspüren, diese zu öffnen und anschließend nach der betroffenden Methoden zu durchsuchen.
An diesem Punkt gibt es zwei mögliche Folgeabschnitte. Je nachdem, ob es dem *Code Clone Detector* möglich war, eine passende Schwachstelle innheralb der Datenbank zu finden, folgt Abschnitt 3.5 oder Abschnitt 3.3

3.3 Erweiterung der Datenbank

Sollte es dem *Code Clone Detector* nicht möglich gewesen sein, anhand der gegebenen *Callstack Informationen* eine passende Schwachstelle zu lokalisieren, wurde über das *Monitoring* ein neuer, zu behebender Fehler gefunden. Nun muss der Nutzer prüfen, ob dies korrekt ist oder ob es zu einer Falscherkennung kam, indem er die betreffende Methode noch einmal per Hand prüft. Sollte er der Meinung sein, dass das Ergebnis des *Monitoring*

korrekt ist, hat er die Möglichkeit, über eine einfache Bestätigung die neue Schwachstelle der Datenbank hinzuzufügen. Diese Bestätigung durch den Nutzer dient dem Zweck, möglichen Falscherkennungen des *Monitorings* entgegenzuwirken und zu verhindern, dass eigentlich korrekter Code als Schwachstelle der Datenbank hinzugefügt wird. Dies würde die Effektivität des Programms stark reduzieren.

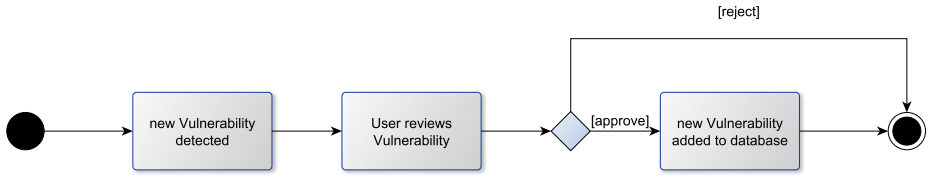

Abbildung 3.4: Schematischer Ablauf der Datenbankerweiterung

Zunächst werden die neuen Einträge in einer separaten Tabelle gespeichert, damit diese nach Möglichkeit von einer zweiten Person bestätigt werden, bevor man sie zu den anderen speichert. Dieser Schritt soll die Qualität der Datenbankeinträge gewährleisten. In Abschnitt 3.4 wird erklärt, wie die Idee der Validierung durch einen weiteren Programmierer leicht zu realiseren ist.

3.4 Verwendung von Git

Um den größtmöglichen Nutzen aus der in Abschnitt 3.3 beschriebenen, sich vergrößernden Datenbank zu ziehen, wurde das Programm um eine *Git-Schnittstelle* erweitert. Wird der *Code Clone Detector* ausgeführt, wird zuallererst eine Verbindung zu dem *Git-Repository*, das die Datenbank enthält, hergestellt und auf Updates geprüft. Sollten also von anderen Nutzern neue Schwachstellen entdeckt und hochgeladen worden sein, bekommt man direkt Zugriff auf die neuen Informationen. Sollte man andererseits selbst die Datenbank erweitert haben, kann man diese am Ende des Programms in das *Repository* pushen. Diese Vorgänge sind in Abbildung 3.5 dargestellt.

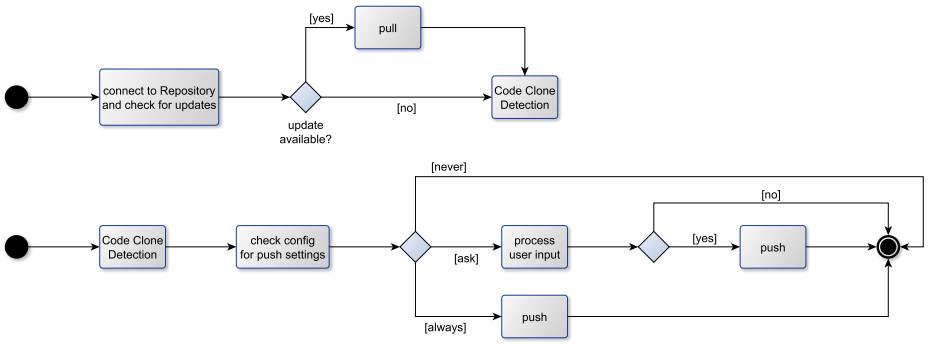

Abbildung 3.5: Schemtatischer Ablauf der Kommunikation mit dem Git-Repository

Wenn Nutzer jedoch auf die Funktion verzichten wollen, können sie einfach die Einstellungen, wie in Abbildung 3.6 zu erkennen, ändern. Diese Erweiterung des *Code Clone Detectors* hat jedoch das Potenzial, zu einem stätigen Wachstum der Datenbank beizutragen. Allerdings resultiert dies auch in einem erhöhten Wartungsaufwand für die Datenbank.

```
Use always, never or ask to specify when to push
programm.automaticPush=never
```

Abbildung 3.6: Ausschnitt aus der Config-Datei

3.5 Automatische Schwachstellenbehebung

Ist es dem *Code Clone Detector* möglich, eine Schwachstelle anhand der *Callstack Informationen* zu finden, kommt die größte Erweiterung zum Einsatz. Dabei handelt es sich um einen Algorithmus zur automatischen Schwachstellenbehebung. Dafür werden jedoch die Informationen der *Fix-Dateien* der Datenbank benötigt, sodass dies nur möglich ist, sollten diese Daten hinterlegt sein.

9	VULNERABILITY\GhC901292cf9d7d8225f8a3b96c7583e2bd8b41772d_1.java	VULNERABILITY
10	FIX\GhC901292cf9d7d8225f8a3b96c7583e2bd8b41772d_2.java	FIX

Abbildung 3.7: Vulnerability und dazugehöriger Fix

3.5.1 Extraktion der Daten

Der Algorithmus beginnt damit, das er die Datenbank nach den nötigen Informationen durchsucht, um diese im Anschluss für die weitere Verarbeitung zu extrahieren. Dazu wird sich der Aufbau der *SQLite-Datenbank*

zu Nutze gemacht, da gefundene Schwachstellen Daten zur Verfügung stellen, die genutzt werden können, um den dazugehörigen *Fix* eindeutig zu identifizieren.

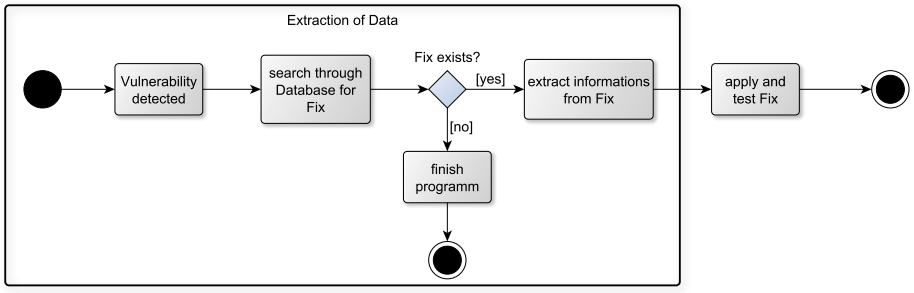

Abbildung 3.8: Ablauf der Datenextraktion

Ist dies geschehen, wird sich einer externen Library bedient, die in der Lage ist, sehr effizient die Unterschiede zweier Texte zu bestimmen. Der Name ist *diff-match-patch* [3], welche sich *Myer's diff algorithm* für diesen Zweck zu Nutze macht. Als Ergebnis erhält man eine Liste von Strings und Anweisungen, die jeweils die Unterschiede und die Art der Operation beinhalten, die nötig wären, um den einen Text in den anderen umzuwandeln. Dabei werden die *EQUAL* Anweisungen der Library ignoriert, da an diesen Stellen kein Eingriff nötig ist.
Tabelle 3.1 zeigt außerdem anhand eines einfachen Beispiels, wie sich die Art der Unterschiede auf die Anzahl der Anweisungen auswirkt. Dieser Zusammenhang zeigt, dass viele, kleine Abweichungen in einer Vielzahl von Anweisungen resultieren, sodass sich somit das Risiko für die automatische Schwachstellenbehebung erhöht. Der Grund hierfür ist, dass jede ausgeführte Operation die Gefahr birgt, an der falschen Stelle angewendet zu werden. Hierzu allerdings mehr in Abschnitt 3.5.2

Art der Unterschiede	**String A**	**String B**	**Anzahl Anweisungen**
viele kleine	aaaaaa	ababab	6
wenige große	aaaaaa	bbbbba	2

Tabelle 3.1: Verhältnis zwischen Art der Unterschiede und Anzahl der daraus resultierenden Anweisungen

Da es je nach Grad der Abweichung von *Vulnerability-File* zu *Fix-File* zu mehr oder weniger *false-positive* Erkennungen kommen kann, also Codezeilen, die man eigentlich nicht für die automatische Schwachstellenbehebung benötigt, wird das Ergebnis ein weiteres Mal gefiltert. Dies soll verhindern, dass Abschnitte eingefügt oder gelöscht werden, die dadurch das Programm

zum Abstürzen bringen würden.
Anschließend wurde die Idee von *Contracts* adaptiert. Diese beinhalten Spezifikationen für das korrekte Verhalten einer Methode, und zwar in Form von *preconditions*, *postconditions* und *intermediate assertions* [6].

preconditions Bei dieser Spezifikation handelt es sich um Bedingungen, die vor einer auszuführenden Methode erfüllt sein müssen. Dazu gehört zum Beispiel, dass bereits andere Methoden ausgeführt worden sein müssen, da das daraus resultieren Ergebnis benötigt wird.

postconditions Hierbei handelt es sich um die Bedingungen, die nach dem Ausführen einer Methode erfüllt sein müssen. Ein Beispiel hierfür ist die Voraussetzung eines bestimmten Ergebnisses, dass für den weiteren Programmverlauf benötigt wird.

intermediate assertions Dies sind die Bedingungen, die während des Ausführens einer Methode erfüllt sein müssen, zum Beispiel Variablen, die bestimmte Werte besitzen müssen.

Diese Einteilung wurde eigens für den hier entwickelten Algorithmus adaptiert. Dies geschah, indem hauptsächlich die *intermediate assertions* betrachtet werden, und zwar dahingehend, dass für jede Codezeile festgelegt wird, welche Zeile zuvor und welche danach kommen muss. Diese Zeilen sind die zuvor erwähnten *pre-* und *postconditions*. Diese Informationen können nun dafür genutzt werden, um den Fix an der korrekten Stelle einzufügen.
Die einzelnen *Contracts* beziehen sich somit nicht mehr auf ganze Methoden, sondern beschreiben die Bedingungen, die vor, während und nach einer Zeile im Code erfüllt sein müssen. Dieses Wissen über den Programmablauf innerhalb einer Methode ist das Herzstück der hier entwickelten automatischen Schwachstellenbehebung und wird im späteren Verlauf zur Lokalisation der zu fehlerhaften Stellen genutzt.

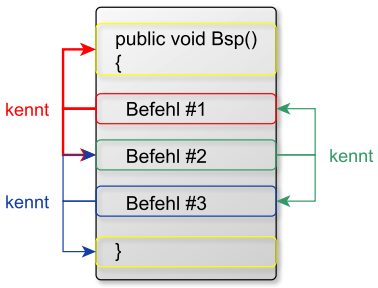

Abbildung 3.9: Adaption der *intermediate assertions*

3.5.2 Integration der Daten

Sind alle vorherigen Schritte ausgeführt worden, kommt es nun zum eigentlichen Versuch der *automatischen Schwachstellenbehebung*. Es wurden zuvor alle nötigen Informationen aus der Datenbank extrahiert, sodass es nur noch darum geht, den fehlerhaften Code anzupassen und somit die Schwachstelle zu beheben.
Dazu wird die gesamte Liste der Unterschiede aus 3.5.1 Stück für Stück verarbeitet. Für jeden Eintrag wird die Stelle gesucht, an dem der String mit der dazugehörenden Anweisung verarbeitet werden muss. Diese sind:

DELETE lösche den spezifischen String

oder

INSERT füge den spezifischen String ein

Das Auffinden der korrekten Stelle für den *Fix* geschieht mit Zuhilfenahme der bereits erstellten *Contracts* und der Nutzung der *Levenshtein-Distanz*. Die *Contracts* enthalten jeweils einen Teil des *Fixes*, sowie die Zeile, die zuvor im Code vorhanden sein muss und die, die danach kommen muss. Anhand dieser *pre-* und *postconditions* kann man nun den Ort finden, an dem die Zeile eingefügt werden muss, indem man im fehlerhaften Code mittels *Levenshtein-Distanz*, das äquivalent lokalisiert. Dabei ist zu beachten, dass die normale *Levenshtein-Distanz* die Anzahl der Operationen beschreibt, die nötig sind, um einen String in einen anderen umzuwandeln. Daher wurde das Ergebnis für diesen Zweck minimal angepasst, um die prozentuale Übereinstimmung zweier Strings darzustellen.

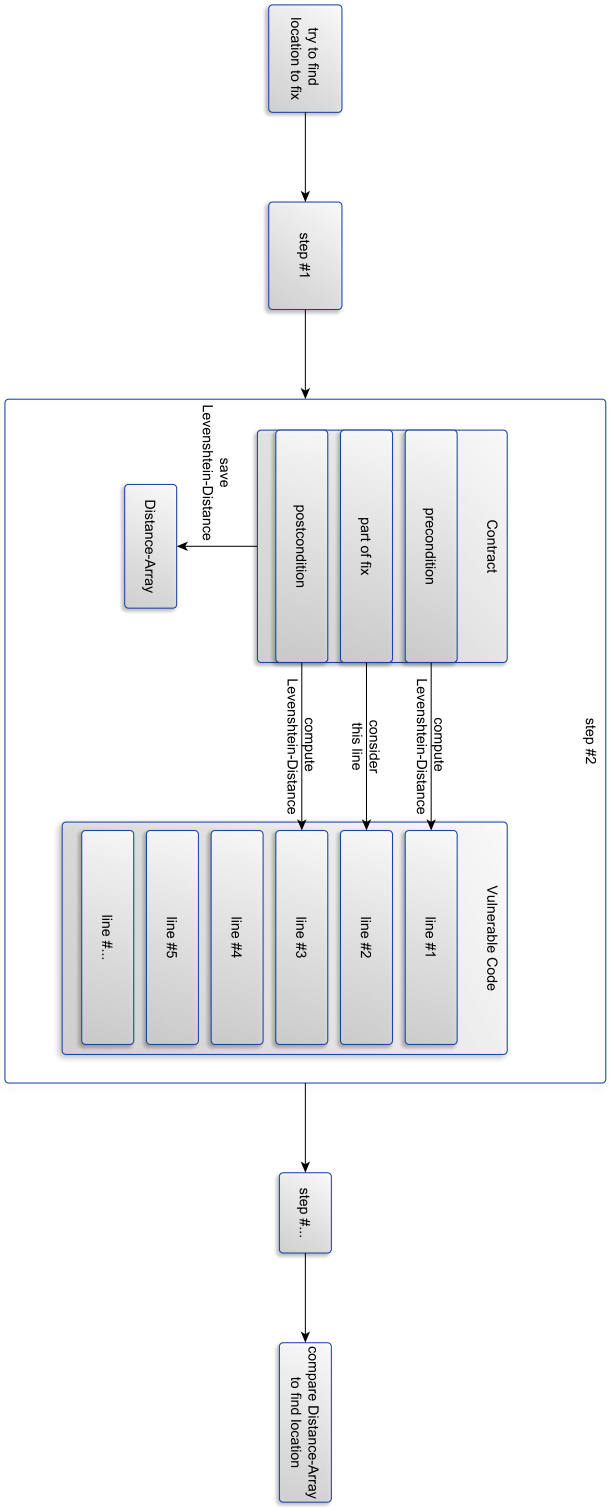

Abbildung 3.10: Schematischer Ablauf des Auffindens der Stelle für den Fix

Nachdem die Liste endgültig abgearbeitet ist, muss nun noch verifiziert werden, ob die automatische Schwachstellenbehebung zu einem fehlerfreien Code geführt hat. Dies geschieht, indem sich die *Java Compiler Schnitstelle* zu Nutze gemacht wird, um im Anschluss an den *Fix* die Datei zu kompilieren. Klappt dies fehlerfrei, wird das Programm beendet. Andernfalls wird der Fehlschlag dem Nutzer mitgeteilt und ihm werden zusätzliche Informationen über die Original *Fix-Datei* gegeben, sodass er das Problem per Hand beheben kann.

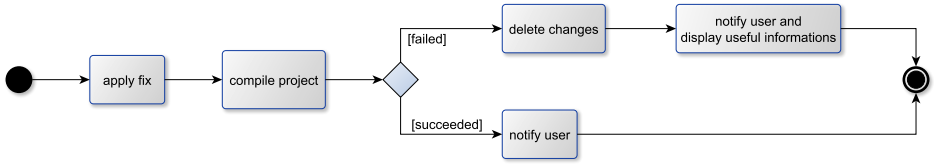

Abbildung 3.11: Schematischer Ablauf des Validierens des Fixes

Kapitel 4

Implementierung

In diesem Kapitel wird darauf eingegangen, wie das zuvor vorgestellte Konzept letztendlich umgesetzt wird. Dafür wird zunächst die Architektur beschrieben, bevor auf die einzelnen Komponenten eingegangen wird. Dies sind zum einen die Konfigurationsdatei 4.2.1, welche die Einstellungen enthält, die vor Programmstart gesetzt werden müssen, das Git-Interface 4.3.1, welches mit dem Repository kommuniziert, die neu hinzugefügte Verarbeitung der *Callstack Informationen* 4.3.2 und die Erweiterung des Datenbank-Managements 4.3.3. Zum anderen wird die Implementierung der automatischen Schwachstellenbehebung beschrieben, die sich in Extraktion 4.4.1 der nötigen Daten und der anschließenden Integration 4.4.2 dieser aufteilt.

4.1 Architektur

Um die Übersichtlichkeit des Codes zu gewährleisten, wurde die grundlegende Packagestruktur des *Code Clone Detectors* beibehalten und lediglich erweitert. Wie in Abbildung 4.1 zu sehen, wurden zwei weitere Packages hinzugefügt. Im Package *rsccd.controller* befinden sich, unter anderem, die angepassten Dateien zur Verarbeitung der *Callstack Informationen* sowie die Dateien zur Steuerung der Datenbank und der Git-Schnittstelle. Die automatische Schwachstellenbehebung befindet sich wiederum in einem eigenen Package, und zwar in *rsccd.autofix*.

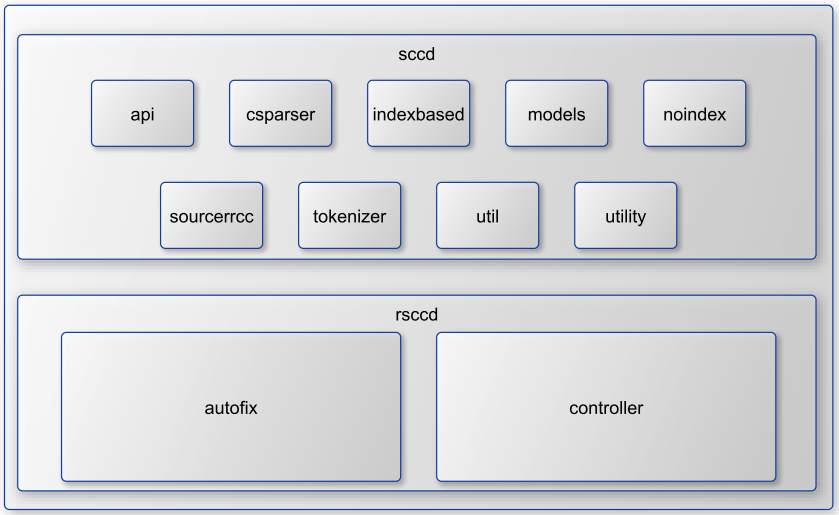

Abbildung 4.1: Package Struktur

4.2 Einstellungen vor Programmstart

Der *Code Clone Detector* wurde als Eclipse-Plugin entwickelt und besaß somit eine Benutzeroberfläche. Da die hier vorgenommenden Änderungen jedoch dazu führen sollten, dass das Programm über die Konsole ausgeführt wird, ist es nötig gewesen, dem Nutzer eine andere Möglichkeit zur Verfügung zu stellen, wie er verscheidene Eisntellungen ändern kann. Dazu wurde eine *config.properties* Datei mit einer dazugehörigen Klasse *ConfigLoader.java* angelegt.

4.2.1 Konfigurationsdatei

Wie in Abbildung 4.2 klar dargestellt ist, handelt es sich hierbei um eine einfache Textdatei mit verschiedenen Parametern. Diese sind in drei Gruppen unterteilt, und zwar in die Einstellungen, die der *Code Clone Detector* benötigt, die das Git-Interface zur Verbindung zum Repository braucht, sowie Parameter, die durch den Umbau neu hinzugekommen sind.
Nur die Zeilen die mit *programm* beginnen, werden im weiteren Programmablauf ausgelesen, sodass es einfach ist, dem Nutzer hilfreiche Kommentare und Erklärungen zu geben, ohne auf die Funktionalität achten zu müssen.

```
SCCD initial settings:
programm.outputPath=C:\\Users\\johnm\\git\\runtime-security-code-clone-detection\\Output
programm.settingsPath=C:\\Users\\johnm\\git\\runtime-security-code-clone-detection\\Settings
programm.localCVEDatabasePath=C:\\Users\\johnm\\git\\SCCD\\cve.db
programm.localSecurityDatabasePath=C:\\Users\\johnm\\git\\runtime-security-code-clone-detection\\Mixed\\results\\sqlite.db
programm.localDatabaseVulnerabilityPath=C:\\Users\\johnm\\git\\runtime-security-code-clone-detection\\Mixed\\results\\VULNERABILITY

GitLab interface settings:
programm.sshPrivateKeyPath=C:\\Users\\johnm\\.ssh\\id_rsa
programm.gitSecurityRepositoryPath=https://git.se.uni-hannover.de/Abschlussarbeiten/runtime-security-code-clone-detection.git
programm.localSecurityRepositoryPath=C:\\Users\\johnm\\git\\runtime-security-code-clone-detection
programm.gitUserName=XXXXXXX
programm.gitUserPassword=XXXXXXX
Use always, never or ask to specify when to push
programm.automaticPush=never

RSCCD initial settings:
programm.projectPath=C:\\Users\\johnm\\git\\runtime-security-code-clone-detection\\evaluation
```

Abbildung 4.2: Inhalt der Config Datei

Somit steht dem Nutzer, trotz nicht vorhandener Benutzeroberfläche, eine leicht verständliche und einfach nutzbare Möglichkeit zur Verfügung, um die Einstellungen an seine Bedürfnisse anzupassen oder sie gar zu erweitern. Sollte eine Erweiterung vorgenommen werden, muss lediglich die dafür vorgesehene Klasse *ConfigLoader.java* um eine entsprechende Methode ergänzt werden.
Bei dieser Klasse handelt es sich um eine Sammlung von einfachen *getter* Methoden, die dafür zuständig sind, die einzelnen Einstellungen zu laden. Sollten nun Erweiterungen an der *config.properties* vorgenommen werden, ist dies der Ort, an dem die dazugehörige Methode implementiert werden muss. Da der grundlegende Aufbau aller *getter* gleich ist, kann sich der Nutzer einfach daran orientieren, ohne sich im Detail mit dem Code beschäftigen zu müssen. Wenn dieses Programm also um weitere Funktionalitäten erweitert werden soll, kann dies ohne große Schwierigkeiten getan werden.

4.3 Änderung im Programmablauf

Im allgemeinen Programmablauf des *Code Clone Detectors* mussten Änderungen vorgenommen werden, da er nicht mehr wie zuvor als Plugin läuft, sondern von dem Nutzer über die Konsole ausgeführt wird. Somit ist das Programm nun auf eine passende Eigabe durch den Nutzer angewiesen. Außerdem müssen weitere Zwischenschritte aufgrund der neuen Funktionen hinzugefügt werden.

4.3.1 Kommunikation mit Git

Nachdem das Programm über die Konsole aufgerufen wurde, wird zuallererst der Stand der lokalen Datenbank überprüft. Dabei kann es zu verschiedenen Situationen kommen.

Sollte die Datenbank noch gar nicht vorhanden sein, sorgt die Git-Schnittstelle dafür, dass das Repository zunächst geklont wird. Dies ist natürlich notwendig, da ohne vorhandene Daten keine *Code Clode Detection* durchgeführt werden kann. Außerdem benötigt der Nutzer so nur das Programm, welches selber dafür sorgt, dass alles nötige vorhanden ist.
Sollte das Gegenteil der Fall sein, also die Datenbank bereits vorhanden sein, wird das Repository ledigleich auf Updates geprüft. Somit ist der Nutzer immer in Besitz der aktuellen, im Repository enthaltenen Datenbank, wodurch die Effektivität der *Code Clone Detection* maximiert wird.
Sollte eine neue Schwachstelle entdeckt werden, ist außerdem möglich, wie in Abschnitt 4.3.3 erklärt, die lokal vorhandene Datenbank zu aktualisieren und sie anschließend in das Repository zu pushen. Die dafür in der config vorgesehene Einstellung legt dabei fest, wann bzw. ob dies geschehen soll.
Diese Ziele werden über die externe Library *org.eclipse.jgit* [9] erreicht, die eine Git-Schnittstelle zur Verfügung stellt. Über diese ist es möglich, mit wenigen Zeilen Code mit Git zu kommunizieren. Natürlich sind hierfür, die in 4.3 beschriebenen Einstellungen nötig, um unter anderem die Zugriffsrechte auf das Repository zu validieren.
Sollten weitere Git-Funktionen gewünscht sein, können diese einfach in der *GitController.java* hinzugefügt werden.

```
GitLab interface settings:
programm.sshPrivateKeyPath=C:\\Users\\johnm\\.ssh\\id_rsa
programm.gitSecurityRepositoryPath=https://git.se.uni-hannover.de/Abschlussarbeiten/runtime-security-code-clone-detection.git
programm.localSecurityRepositoryPath=C:\\Users\\johnm\\git\\runtime-security-code-clone-detection
programm.gitUserName=XXXXXXX
programm.gitUserPassword=XXXXXXX
Use always, never or ask to specify when to push
programm.automaticPush=never
```

Abbildung 4.3: Git-Einstellungen der Config Datei

4.3.2 Callstack Informationen

Wie bereits zuvor erwähnt, muss das Programm nun auf eine passende Eingabe des Nutzers warten, um zu arbeiten. Diese Eingabe erfolgt in Form von *Callstack Informationen* über die Konsole. Anschließend müssen diese mit Hilfe von *StringUtils* Operationen in ihre Einzelteile zerlegt werden, wie in bereits Abbildung 3.3 gezeigt. Erst danach besitzen die Daten eine Form, die genutzt werden kann, um die fehlerhafte Datei aufzuspüren
All dies geschieht in der dafür vorgesehenen Klasse *VulnerableCode.java*, die Methoden zur Extraktion jeder Teilinformation der Eingabe enthält. Dabei sind die einzelnen Operationen auf das Format der *Callstack Informationen* angepasst, sodass diese im Moment keine anderen Eingaben verarbeiten kann. Allerdings besitzt die Klasse das Potenzial, mit ein wenig Aufwand

auch Eingaben anderer Form anzunehmen, da die verwendeten *StringUtils* Operationen auf gewünschte Trennzeichen reagieren und somit dahingehend abgeändert werden können.
Anschließend werden die so extrahierten Daten in der *VulnerableCode* Klasse noch weiterverarbeitet. Das bedeutet, dass es verschiedene Methoden gibt, die jeweils eine Teilinformation nutzen, um zum Beispiel die fehlerhafte Datei aufzuspüren oder diese nach der Methode zu durchsuchen, die die Schwachstelle enthält. Am Ende der Verarbeitung steht nun ein Objekt der Klasse, das alle Informationen der *Callstack Informationen*, sowie weitere extrahierte Daten, wie den Namen des Autors, enthält. Dieses wird nun für die eigentliche *Code Clone Detection* genutzt.

4.3.3 Datenbank-Management

Eine weitere Anforderung dieses Projektes ist das Datenbank-Management. Dabei geht es um den Fall, dass innerhalb der vorhandenen *SQLite-Datenbank* keine passenden Schwachstelleninformationen zu den gegebenen *Callstack Informationen* vorhanden sind. Dabei war es, aufgrund einer neuen Datenbankstruktur, zunächst nötig, die bereits vorhandene Datenbankkommunikation anzupassen, da es ansonsten nicht möglich gewesen wäre, an Informationen zu gelangen.
Um dieses Ziel zu realisieren, wird sich der Library *sqlite-jdbc* [7] bedient, welche den Zugriff und das Bearbeiten von *SQLite-Datenbanken* ermöglicht. Zuallererst wird eine Tablle für die neu entdeckten Schwachstellen erstellt. Diese beinhaltet alle im Zuge der *Code Clone Detection* extrahierten Daten. Des weiteren wird selbstverständlich eine Source Datei angelegt, die den entsprechenden Codeabschnitt enthält.
Sollte dieser Fall nun eintreten, liegt es am Nutzer zu prüfen, ob es sich wirklich um eine neue Schwachstelle handelt oder um eine Falscherkennung. Hier liegt Grund dafür, dass die neue Schwachstelle nicht direkt bei den anderen abgelegt wird. Da es möglich ist, dass der Nutzer sich irrt, sollten neue Einträge nach Möglichkeit von einer weiteren Person validiert werden, um die Qualität der Datenbank zu gewährleisten.
Ist dies jedoch nicht gewünscht, kann man sie direkt in die vorhandene Tabelle einpflegen, damit sie direkt bei weiteren Überprüfungen berücksichtigt wird. Allerdings bietet die weitere Tabelle der Datenbank das Potenzial, gezielt *Runtime* Schwachstellen zu suchen, was je nach Umfang der Datenbank in einer besseren Laufzeit resultieren könnte. Dies ist jedoch nur eine Vermutung, die erst noch durch umfangreiche Tests belegt werden muss.

4.4 Schwachstellenbehebung

Dieses Kapitel beschäftigt sich mit dem Algorithmus, der für eine automatische Schwachstellenbehebung in Java Source Code entwickelt wurde. Dieser wird, für eine bessere Übersichtlichkeit, in zwei Teilfunktionen unterteilt, und zwar in Extraktion und Integration, welche im folgenden näher erläutert werden.

Extraktion Dieser Abschnitt befasst sich mit der Extraktion der benötigten Daten aus der Datenbank. Die so gewonnen Informationen werden im weiteren Verlauf für die Schwachstellenbehebung genutzt.

Integration Dieser Abschnitt befasst sich mit der Integration der zuvor extrahierten Informationen aus der Datenbank. Dies ist die Hauptaufgabe der automatischen Schwachstellenbehebung.

4.4.1 Extraktion der benötigten Daten

Wurde eine Schwachstelle erfolgreich lokalisiert, ist es nun nötig, die Informationen der Datenbank so zu verarbeiten, dass das Problem gelöst werden kann. Dazu steht an erster Stelle das Auffinden der zu der Schwachstelle gehörenden *Fix-Datei*. Diese beinhaltet, wie der Name schon sagt, genau die Informationen, die der Algorithmus für sein Ziel benötigt.
Als nächstes werden diese Dateien mittels der in 3.5.1 konzeptuell beschriebenen Library *diff-match-patch* miteinander verglichen. Abbildung 4.4 zeigt, anhand von existierenden Datenbankinhalten das daraus resultierende Ergebnis. Anhand dieses Beispiels ist die Grundidee des Algorithmus deutlich zu erkennen, nämlich dass der Unterschied zwischen *Fix* und *Vulnerability* eben genau die benötigten Zeilen ist, die in den fehlerhaften Code eingefügt werden müssen. Allerdings wird auch gezeigt, dass das Ergebnis Teile des Codes enthält, die nicht mit in den Fix gelangen dürfen. Diese *false-positives* werden daher im weiteren Verlauf der automatischen Schwachstellenbehebung gefiltert. Auf die Ergebnisse und die Qualität dieser Lösung wird in Abschnitt 5 eingegangen.

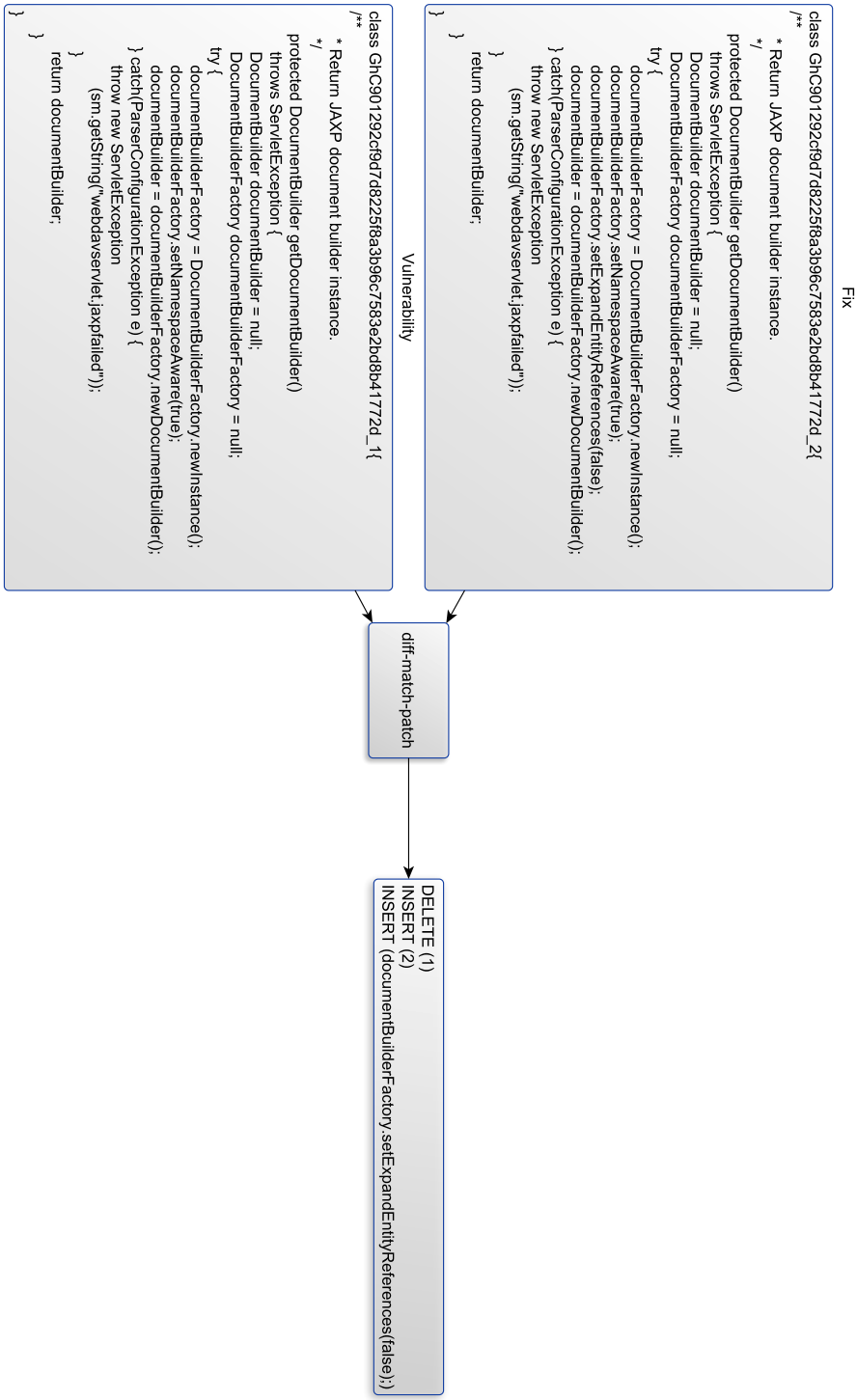

Abbildung 4.4: Ergebnis eines Vergleiches mittels *diff-match-patch*

Nun müssen diese Dateien noch in Hinblick auf ihre Struktur verarbeitet werden. Die Basis, auf dem dies geschieht, sind die in 3.5.1 bereits erwähnten *Contracts*. Eigens dafür wurde eine Datenstruktur mit allen nötigen Informationen und Methoden angelegt.
Im Verlauf des Algorithmus wird für den entsprechenden Code nun ein Array von *Contracts* erstellt. Dabei wird dieser zunächst von Kommentaren bereinigt und anschließend auf zeilenübergreifende Befehle geprüft, bevor das entsprechende Objekt hinzugefügt wird. Dies dient letztendlich der korrekten Abeitsweise der nächsten Schritte. Um alle Kommentare filtern zu können, wurde eine Methode angelegt, die nicht nur normale Kommentare wie *//...* sondern auch zeilenübergreifende erkennt.
Das Programm ist also nun im Besitz der auszuführenden Befehle und deren Inhalt sowie der Information über den dazugehörenden Kontext durch die *Contracts*.

4.4.2 Integration der benötigten Daten

Dieser Abschnitt beschreibt das eigentliche Herzstück des Algorithmus zur automatischen Schwachstellenbehebung. Wie bereits in 3.5.2 beschrieben, handelt es sich hierbei um den Teil des Algorithmus, der dafür zuständig ist, die Stellen im Code zu finden, an dem die zuvor extrahierten Daten eingefügt werden müssen. Die korrekte Arbeitsweise ist essenziell für die automatische Schwachstellenbehebung.
Daher werden zunächst die Ergebnisse der *diff-match-patch* Methode erneut gefiltert, um die Gefahr von *false-positive* Unterschieden zu verringern. Anschließend wird für alle übrig gebliebenden, unter Zuhilfenahme der *Contracts*, die korrekte Stelle lokalisiert.
Dazu wird zunächst der *Contract* gesucht, der den einzufügenden Teil des Fixes enthält. Ist dieser gefunden, werden dessen *pre-* und *postconditions* genutzt, um das Äquivalent im fehlerhaften Code zu finden. Dies geschieht auf Basis der *Levenshtein-Distanz*. Wurde so ein passender Ort lokalisiert, wird der Fix zwischen diesen Äquivalenten eingefügt.
Da es sich in den seltensten Fällen um syntaktische *Clone* handelt, sondern eher um symantische *Clone*, wurde die *Levenshtein-Distanz* so angepasst, dass sie nun die prozentuale Übereinstimmung zweier Strings bestimmt und nicht wie üblich die Anzahl der Änderungen, die vorgenommen werden müssen, um einen String in einen anderen umzuwandeln. Dies ist einfach möglich, indem man die Anzahl der Änderungen durch die Länge des Wortes teilt.
Daher ist eine hundertprozentige Übereinstimmung unwahrscheinlich, und es ist sinnvoller, die prozentual ähnlichsten Codezeilen zu verwenden.
Es ist nun also bekannt, welche Änderungen vorgenommen werden müssen, welcher *Contract* zu dieser Änderung gehört und welche Zeile im Code durch die Übereinstimmung mit dem *Contract* die gesuchte ist. Dies wird

in Abbildung 4.5 noch einmal veranschaulicht.

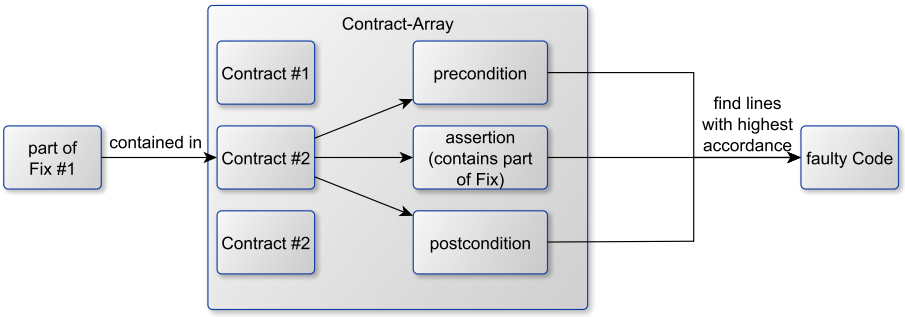

Abbildung 4.5: Auffinden des Ortes, an dem der Fix angewendet werden muss

All diese gewonnen Informationen ermöglichen nun die automatische Schwachstellenbehebung. Da es jedoch an verschiedenen Stellen zu Fehlern kommen kann, ist es nicht ratsam, die Änderungen ohne nachträgliche Überprüfung anzuwenden und beizubehalten. Zum Beispiel könnte durch Zufall ein Teil des Codes eine höhere Übereinstimmung zum betrachteten *Contract* besitzen, obwohl dieser nicht die korrekte Stelle für den Fix ist. Der Grund hierfür ist die Herangehensweise der prozentualen Übereinstimmung, die keine einhundertprozentige Sicherheit gewährleisten kann.
Um nun zu verhindern, dass die Schwachstelle fehlerhaft behoben wird, wurde eigens dafür eine Methode geschrieben, die sich des Java Compilers bedient. Sie versucht, die veränderte Datei zu kompielieren, um den vorgenommenen *Fix* zu validieren. Gelingt dies nicht, ist die Schwachstellenbehung fehlerhaft verlaufen und die Änderungen werden rückgängig gemacht. Um den Nutzer dennoch zu unterstützen, werden ihm die Inhalte der Datenbank zur Verfügung gestellt, sodass dieser das Problem selbst beheben kann.

Kapitel 5

Evaluation

Dieses Kapitel beschäftigt sich mit der Evaluation der *automatischen Schwachstellenbehebung* des *Runtime-Security-Code-Clone-Detectors*. Dabei liegt der Augenmerk darauf, ob diese in der Lage ist, die zum Beheben der Schwachstelle nötigen Codeabschnitte zu lokalisieren und sie wiederum an der richtigen Stelle einzufügen. Dies wird anhand der folgenden Menge von Testdaten evaluiert, die aus der Datenbank entnommen wurden und auch bereits zur Evaluation des *Security-Code-Clone-Detectors* [1] genutzt wurde. Allerdings musste die Menge ein wenig angepasst werden, da es in der aktuellen Datenbank noch nicht zu jeder dieser Schwachstellen auch einen Fix gibt. Diese Tabelle beinhaltet die angepasste Menge und wurde mit einem zufälligen Beispiel aus der Datenbank erweitert.

CVE	**Scoring**	**Product**
CVE-2007-5461	3.5	Apache Tomcat
CVE-2009-2693	5.8	Apache Tomcat
CVE-2010-4172	4.3	Apache Tomcat
CVE-2011-2204	1.9	Apache Tomcat
CVE-2011-3190	7.5	Apache Tomcat
CVE-2016-6723	5.4	Google Android
CVE-2017-0846	5.0	Google Android
CVE-2017-9096	6.8	iTextpdf iText

Tabelle 5.1: Evaluationsmenge aus der Datenbank

5.1 Extraktion

Die erste Aufgabe der *automatischen Schwachstellenbehebung* ist es, die Datenbank nach der benötigten *Fix-Datei* zu durchsuchen. Wenn dies erfolgreich gewesen ist, muss die entsprechende Datei noch nach den expliziten Codeabschnitten durchsucht werden, die die gefundene Schwachstelle

beheben.
Um einen solchen Algorithmus zu bewerten, bedient man sich der Maße *precision* und *recall*. In diesem Fall sind sie wie folgt definiert:

$$precision = \frac{|Gefundene, relevante\ Codeabschnitte|}{|Gefundene\ Codeabschnitte|}$$

$$recall = \frac{|Gefundene, relevante\ Codeabschnitte|}{|Relevante\ Codeabschnitte|}$$

Die Priorität dieser Werte hängt in der Regel vom jeweilligen Anwendungsbereich ab. Betrachtet man nun das gewünschte Ziel des Algorithmus, wird ersichtlich, dass ein hoher *recall* erreicht werden soll, da alle relevanten Codeabschnitte benötigt werden, um die Schwachstelle zu beheben. Auf der anderen Seite ist es jedoch unerlässlich, dass eine möglichst maximale *precision* erreicht wird, da irrelevante Abschnitte wiederrum zum Absturz des Programms führen könnten. Dieses hohe Maß in beiden Werten zu erreichen ist, das Ziel dieses Algorithmus.
Um nun diese beiden Werte bestimmen zu können, wird sich daher der, in Tabelle 5.1 dargestellten Schwachstellen bedient. Dafür werden die Zwischenergebnisse des Algorithmus mit den erwarteten Ergebnissen dahingehend verglichen, welche Codeabschnitte gefunden wurden und ob alle relevanten Abschnitte enthalten sind.
Die Zwischenergebnisse des Algorithmus zeigen jedoch schnell, dass der *recall* Wert, aufgrund der *diff-match-patch* Library, kein Problem darstellt. Mit dem hier verfolgten Ansatz ist dieser nämlich maximal, da alle Unterschiede zwischen *Fix* und *Vulnerability* extrahiert werden, sodass auf jeden Fall alle relevanten Codeabschnitte in dieser Menge enthalten sind. Daher liegt das weitere Augenmerk auf dem *precision* Wert.
Die bisherigen Ergebnisse zeigen zwar, dass alle relevanten Abschnitte extrahiert werden, jedoch haben die Tests auch gezeigt, dass aus allen Dateien Daten extrahiert worden sind, die nicht in die spätere Schwachstellenbehebung gelangen dürfen. Dazu gehören, unter anderem Kommentare, die das Risiko bergen, falsch eingefügt zu werden und somit den Algorithmus letztendlich scheitern lassen. Außerdem Unterschiede im Klassennamen, die natürlich ignoriert werden sollen, da diese keine Auswirkung auf die Funktionalität haben und zum selben Ergebnis wie zuvor genannt führen könnten, wenn sie dennoch eingefügt werden.
Daher ist der Inhalt der Datenbank von fundamentaler Wichtigkeit. Je besser der Inhalt der *Fix-Datei* an den der *Vulnerability-Datei* angepasst ist, so wie es bei der hier verwendeten Datenbank der Fall ist, desto höher ist die *precision*. Außerdem werden sämtliche Kommentare gefiltert, um das Risiko weiter zu reduzieren und den *precision* Wert weiter zu erhöhen. Darüber hinaus werden die Ergebnisse im weiteren Programmverlauf ein weiteres Mal gefiltert, um die Zahl der *false-positives* möglichst komplett zu eliminieren.
Abbildung 5.2 zeigt die Ergebnisse dieser Evaluation. Dabei wurden alle oben

genannten Daten von dem Algorthimus verarbeitet und ausgewertet. Dabei werden die Werte nach dem initialen Vergleich mit der externen Library und nach dem erneuten Filtern der daraus resultierenden Ergebnisse dargestellt.

	precision	**recall**
initialer Vergleich:	66%	100%
erneutes Filtern:	100%	100%

Tabelle 5.2: Verbesserung des precision Wertes durch erneutes Filtern

Wie man sieht, ist es nur mit Hilfe des erneuten Filterns möglich, den benötigten *precision* Wert zu erhalten. Somit ist der Algorithmus in der Lage *false-positives* zu filtern, um letztendlich einen validen *Fix* zu generieren. Dennoch sind weitreichendere Tests nötig, um die Korrektheit der Ergebnisse zu zeigen, da nur eine vergleichsweise kleine Menge an Testdaten genutzt wurde.

5.2 Integration

Nachdem die benötigten Informationen aus der Datenbank extrahiert worden sind, muss der fehlerhafte Code analysiert werden. Dies ist notwendig, um die Stellen zu lokalisieren, an denen die *Fixes* eingefügt werden müssen. Des weiteren kann es erforderlich sein, einige Variablennamen des *Fixes* anzupassen, damit der entsprechende Codeabschnitt weiterhin lauffähig ist. Zuallererst werden die in Kapitel 5.1 beschriebenen Strings auf Variablennamen geprüft. Sollte einer gefunden werden, wird versucht, anhand des dazugehörigen Variablentyps das Pendant im fehlerhaften Code zu bestimmen. Funktioniert dies, wird der *Fix* entsprechend angepasst.
Allerdings hängt die Erfolgsquote von der Anzahl der Variablen des gleichen Typs innerhalb der fehlerhaften Datei ab. Wie in Abbildung 5.1 zu sehen ist, beträgt die Wahrscheinlichkeit:

$$probability = \frac{1}{|Variablen\ gleichen\ Typs|}$$

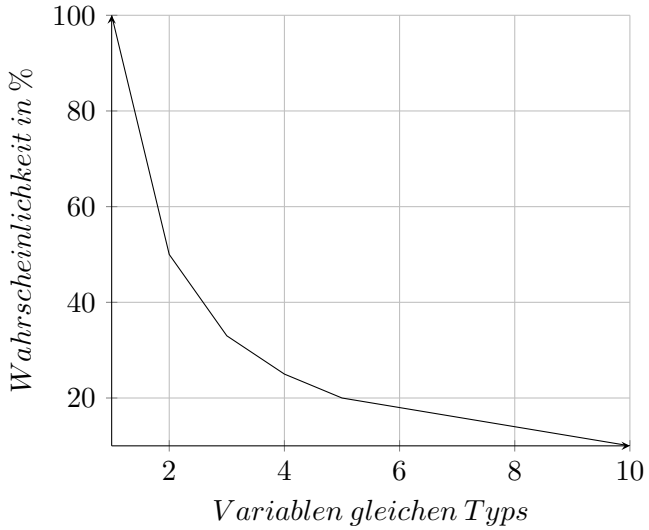

Abbildung 5.1: Wahrscheinlichkeit, korrekten Variablennamen zur Änderung auszuwählen

Somit hängt die Erfolgswahrscheinlichkeit der automatischen Schwachstellenbehebung von Typ-2 und Typ-3 Klonen von deren Komplexität ab. Allerdings zeigen die Tests trotz dieses Problems eine erstaunliche Trefferquote, sodass es zu keinen Fehlern aufgrund falscher Variablennamen kam. Allerdings sollte ersichtlich sein, dass dies keine Beweis für eine einhundert prozentige Wahrscheinlichkeit ist, sondern durch die zugrundeliegenden Daten zustande kommt. Daher dies ist ein Teil des Projektes an dem sich eine Weiterentwicklung des Algorithmus lohnt.
Anschließend werden die in Kapitel 3 beschriebenen *Contracts* genutzt, um anhand dieser die Stelle im Code zu lokalisieren, an der der *Fix* angewendet werden muss. Dabei bedient man sich, wie bereits zuvor erklärt, der *Levenshtein-Distanz*, die es ermöglicht, den Codeabschnitt mit der größten Ähnlichkeit zum *Contract* zu finden.
Allerdings wird somit lediglich die Annahme getroffen, dass die Stelle mit der größten Ähnlichkeit auch die korrekte ist. Ob sich dies bestätigt, zeigt sich erst, wenn man zum Ende der *automatischen Schwachstellenbehebung* den geänderten Code kompiliert.
Zu diesem Zweck wurden die vier Datein genutzt, mit denen schon W. Brunotte [1] seine Arbeit evaluierte. Dabei handelt es sich um vier Dateien, eine für jeden Klontyp mit den in Abbildung 5.1 genannten Schwachstellen, und eine weitere, die die behobenen Schwachstellen besitzt. Anschließend wurden all diese Schwachstellen in all diesen Dateien durch den hier entwickelten Algorithmus verarbeitet. Gelang es, den Code zu kompilieren, wurde das Ergebnis anschließend noch mit dem Inhalt der vierten Datei verglichen, um sicherzustellen, dass alles korrekt verlaufen ist. Die Ergebnisse

dieser Herangehensweise werden in Tabelle 5.3 grafisch dargestellt.

	Typ-1	**Typ-2**	**Typ-3**	**Fix**
Erfolgsrate:	50%	50%	50%	100%

Tabelle 5.3: Erfolgsrate der automatischen Schwachstellenbehebung in Bezug auf die verschiedenen Clone-Typen

Die Qualität der Ergebnisse der Integration hängt somit von verschiedenen Faktoren ab. Zum einen ist da die Komplexität bzw. der Umfang des bestehenden Codeabschnitts. Dieser kann sehr schnell zu einem Scheitern der *automatischen Schwachstellenbehebung* führen, da eine Vielzahl an Variablen das Risiko erhöht, dass es mehrere des selben Typs gibt. Dies führt wiederum zu einer geringeren Wahrscheinlichkeit, die Variablennamen richtig anzupassen.
Außerdem basiert die Lokalisierung der gesuchten Stelle im Code bisher auf einer Vermutung, die jedoch erst vollkommen bewiesen werden muss. Dies ist jedoch nicht so leicht, da es für einen eindeutigen, zweifelsfreien Beweis einer enormen Menge an Tests bedarf. Somit können die oben genannten Ergebnisse im Verlauf weiterer Forschung variieren.
Es ist jedoch ersichtlich, dass die automatische Schwachstellenbehebung Clone-Typ unabhängig möglich ist. Sollte es also für den Algorithmus möglich gewesen sein, eine Schwachstelle zu beheben, schaffte er dies auch beim Äquivalent eines anderen Typs
Mit einer Erfolgsrate von 50% ist es zwar sinnvoll den Versuch zu unternehmen, jedoch sollte es weiterhin die Möglichkeit für den Benutzer geben, den Inhalt der Datenbank abzurufen, falls der Algorithmus scheitert.

Kapitel 6

Verwandte Arbeiten

Die bereits erwähnte Arbeit von Y. Pei et al. [6] beschäftig sich mit der Nutzung von *Contracts*. Doch im Gegenteil zu Java, ist die dort verwendete Programmiersprache *Effeil* direkt mit diesen ausgestattet. Somit war es hier zuerst nötig, die Idee der *Contracts* so zu verändern, dass sie auf Java Source-Code angewendet werden kann. Sie wurden etwas verallgemeinert, sodass sie nun Codezeilen beinhalten, die vor, während und nach einer bestimmten Stelle benötigt werden, um einen korrekten Programmablauf zu sichern.
Außerdem basiert der dort verwendete Algorithmus darauf, selber Fixes zu generieren und nicht, wie in diesem Fall, vorhandenes Wissen zu nutzen. Dafür wird zunächst eine Reihe von möglichen Fixes generiert. Diese werden dann getestet, bis ein valider Fix gefunden ist, oder alle fehlgeschlagen sind. Daraus resultiert eine weitaus höhere Laufzeit im Gegensatz zu dem hier verfolgten Ansatz.
Zu guter Letzt wurde eine minimal höhere Erfolgswahrscheinlichkeit erreicht. Die Arbeit von Y.Pei er al. besitzt eine Trefferquote von 42% im Vergleich zu den hier erreichten 50%. Allerdings zeigen diese Zahlen, dass es sich um ein äußerst komplexes Problem handelt, für das es bisher keine 100% Lösung gibt.
Einen weiteren Ansatz zur automatischen Schwachstellenbehebung bietet die Arbeit von J. Hua et al. [4]. Auch hier wird versucht, ein Fix auf Grundlage des fehlerhaften Codes zu generieren. Allerdings geschieht dies hier auf Grundlage sogenannter *sketches*. Dabei handelt es sich um einzelne Abschnitte des eigentlichen Codes, für die versucht wird, ein Fix zu finden. Dieser Ansatz nennt sich *SKETCHFIX*, welcher den Vorteil hat, dass nicht ein Fix gefunden werden muss, der alle Fehler gleichzeitig behebt, sondern mehrere kleine Fixes, die jeweils einen Teil des Problems lösen.
Diese einzelnen Abschnitte, auch *Holes* genannt, sind Ausdrücke, die verschiedene Werte annehmen können, für die jeweils der korrekte Werte gesucht wird. An dieser Herangehensweise wird sich hier in gewisser Weise orientiert, mit dem Unterschied, dass der korrekte Wert bekannt ist und das *Hole*

gesucht wird, an dem dieser eingefügt werden muss.
Auch im Vergleich zu dieser Arbeit besitzt der hier entwickelte Algorithmus eine bessere Laufzeit und Fixwahrscheinlichkeit, die daraus resultiert, dass Wissen über den Fix existiert und somit genutzt werden kann, ohne ihn erst generieren zu müssen. Allerdings hat dies den Nachteil, dass eine genügend große Datenbank vorhanden sein muss, damit dieser Algorithmus arbeiten kann. Somit besitzt er zwar einige Vorteile, aber es muss viel Aufwand in den Umfang und den Inhalt der Datenbank investiert werden.
Eine Weitere verwandte Arbeiten wurde von S. Mechtaev et al. [5] verfasst. Der dort verwendete Ansatz zur Erstellung eines Fixes ähnelt der in dieser Arbeit genutzten Herangehensweise sehr. Es wird sich nämlich sogenannter *Referenz Dateien* bedient, welche Funktionalitäten in alternativer Form beinhalten, um einen Fix zu generieren. In dem hier verfolgten Ansatz wären dies die Dateien der Datenbank, die den fehlerfreien Code enthalten.

Kapitel 7

Zusammenfassung und Ausblick

Dieses Kapitel beinhaltet die Zusammenfassung der Ergebnisse und gibt anschließend einen Ausblick auf mögliche Versesserungen und weiterführende Ansätze. Außerdem wird kurz auf die *Threats to vadility* dieser Arbeit eingegangen.

7.1 Zusammenfassung

Im Zuge dieser Bachelorarbeit wurde der *Security Code Clone Detector* von Brunotte [1] weiterentwickelt. Zum einen ist es nun möglich *Callstack Informationen* zu verarbeiten, was zu einer höheren Programmsicherheit beiträgt, da nun auch Probleme zur Laufzeit des Programms erkannt und behoben werden können. Zum anderen tragen die Datenbank- und die Git-Schnittstelle zu einer besseren Schwachstellendatenbank sowie dessen Austausch mit anderen Nutzern bei. Somit ist der Nutzer nun auch selbst in der Lage, zusätzlich zu den Algorithmen zur automatischen Suche nach Schwachstellen im Internet, die Erweiterung der Datenbank voranzubringen. Die automatische Schwachstellenbehebung bietet bereits jetzt, wie die Evaluation gezeigt hat, eine gute Erfolgsquote, die jedoch nicht hoch genug ist, um ihr zweifelsfrei vertrauen zu können. Allerdings besitzt der Algorithmus noch Potenzial zur Verbesserung.

7.2 Risiken für die Validität

Wie bereits erwähnt, hängt die Erfolgsrate der automatischen Schwachstellenbehebung stark von der zugrunde liegenden Datenbank ab. Daher ist eine hohe Qualität der Datenbankeinträge signifikant wichtig und sollte unter allen Umständen beibehalten werden.

Die vergleichweise kleine Testmenge, die aufgrund der jetzigen Funktionsweise und der zeitlichen Beschränkung nicht größer gewählt werden konnte, kann Auswirkungen auf die Qualität der Evaluation haben. Dennoch sollte diese ausreichen, um erste Ergebnisse zu erlangen und somit einen Ausblick auf spätere Ergebnisse gibt.

7.3 Ausblick

Um die Evaluation zu verbessern und zu erleichtern, ist es sinnvoll, das Programm so abzuändern, dass nicht nur einzelnen *Callstack Informationen* verarbeitet werden können, sondern auch größere Datenmengen. Dies könnte zu einer schnelleren Erweiterung der Datenbank führen sowie die *Usability* für den User verbessern, da er nicht alles einzeln abarbeiten muss.
Außerdem scheint eine Benutzeroberfläche sinnvoll, die dem Nutzer zum einen das Datenbankmanagement erleichtert und zum anderen eine verbesserte Oberfläche für die Ergebnisse bereitstellt. Die Ausgabemöglichkeiten der Konsole sind dahingehend beschränkt.
Des weiteren sollte die Änderung von Variablennamen weiterentwickelt werden, da der bisherige Algorithmus dort sehr einfach ist und eine Möglichkeit, den korrekten Namen zu validieren, die Qualität der Ergebnisse verbessern wird.
Das größte Problem der automatischen Schwachstellenbehebung ist es bisher, den extrahierten Fix an der richtigen Stelle einzufügen. Dies basiert bisher auf der *Levenshtein-Distanz* und könnte mit einem anderen Ansatz möglicherweise verbessert werden.

Anhang A

Anhang

A.1 DVD

Auf der beiliegenden DVD befindet sich:

- Kopie dieser schriftlichen Arbeit
- Runtime-Security-Code-Clone-Detector in ausführbarer Form
- Handbuch zur Software
- Vollständiger Quellcode für alle Programmteile und Tools
- LaTeX Dateien zu dieser schriftlichen Arbeit
- Der Evaluation zugrunde liegenden Daten
- Internetquellen

Literaturverzeichnis

[1] W. Brunotte. Security code clone detection entwickelt als eclipse plugin. Master's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2018.

[2] Y. Evers. Suche von sicherheitsrelevantem code auf stack overflow. Bachelor's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2018.

[3] N. Fraser. Diff match and patch. `https://github.com/google/diff-match-patch`, Released: 2018. [Online; accessed 26-July-2019].

[4] J. Hua, M. Zhang, K. Wang, and S. Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 12–23, New York, NY, USA, 2018. ACM.

[5] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 129–139, New York, NY, USA, 2018. ACM.

[6] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, May 2014.

[7] T. L. Saito. Sqlite jdbc driver. `https://bitbucket.org/xerial/sqlite-jdbc/src/default/`, Released: 2007. [Online; accessed 27-July-2019].

[8] SalSoft Pawel Salawa. Sqlitestudio. `https://sqlitestudio.pl/index.rvt?act=about`, Released: 2007. [Online; accessed 02-August-2019].

[9] M. Sohn. Eclipse foundation: Jgit. `https://www.eclipse.org/jgit/`, Released: 2010. [Online; accessed 27-July-2019].

[10] F. P. Viertel, W. Brunotte, D. Strüber, and K. Schneider. Detecting security vulnerabilities using clone detection and community knowledge. *The 31st International Conference on Software & Knowledge Engineering*, 2019.