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

Security Code Clone Detection entwickelt als Eclipse Plugin

Masterarbeit

im Studiengang Informatik

von

Wasja Brunotte

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Joel Greenyer
Betreuer: M. Sc. Fabien Patrick Viertel

Hannover, 17.05.2018

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig 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 17.05.2018

Wasja Brunotte

Zusammenfassung

Security Code Clone Detection entwickelt als Eclipse Plugin

In dieser Masterarbeit wird ein Werkzeug zur automatisierten Schwachstellensuche in Quellcode vorgestellt.

Software wird heutzutage immer komplexer und ist im Alltag allgegenwärtig. Ebenso dringt sie in immer sensiblere Bereiche vor. Daher sollte eine Suche nach Schwachstellen in den Software-Entwicklungsprozess integriert sein. Die Suche nach Schwachstellen ist zum einen aber sehr zeitintensiv, zum anderen erfordert sie Expertenwissen im Hinblick auf die Schwachstellen.

Zur Lösung dieser Probleme wurde im Rahmen dieser Masterarbeit ein Eclipse Plugin entwickelt, was die Schwachstellensuche im Quellcode automatisiert, den Entwickler darüber informiert und Lösungsvorschläge gibt. Hierbei kommt ein Clone Detector zum Einsatz, der den geschriebenen Code auf bekannte Sicherheitslücken untersucht. Der Clone Detector nutzt dafür ein vorhandenes Security Code Repository, das bereits bekannte Schwachstellen beinhaltet. Sollten Schwachstellen gefunden werden, wird der Entwickler darauf hingewiesen und - sofern vorhanden - werden ihm Details zur Schwachstelle angezeigt, mögliche Fixes angeboten oder auch Exploits, die die Schwachstelle ausnutzen, gezeigt.

Die Schwachstellensuche läuft vollautomatisch im Hintergrund ab, so dass der Entwickler bei seiner Arbeit nicht gestört wird. Mit Hilfe des Plugins wird er frühzeitig auf etwaige Schwachstellen hingewiesen und hat die Möglichkeit, diese gleich zu beheben und somit sichereren Code zu schreiben.

Abstract

Security Code Clone Detection developed as Eclipse plug-in

This master thesis presents a tool for automated vulnerability search in source code.

Nowadays software is becoming more complex and is an integral part of everyday life. Thus, a search for vulnerabilities should be a part of the software development process. On the one hand identifying vulnerabilities is quite time consuming, on the other hand it requires expert knowledge in interpretation of vulnerability analysis.

As a solution, this thesis presents an Eclipse plug-in which automatically scans source code for known vulnerabilities and helps the developer to fix them. This is done by code clone detection. To achieve the goal of detecting vulnerable code clones the clone detector uses a local security code repository with known vulnerabilities. If a vulnerability is found, the tool reports it to the developer and shows detailed information about the vulnerability like the CVE information. Even a possible fix or an exploit for this vulnerability will be presented if these information exist.

Searching for vulnerabilities runs in a background process so that the developer is not interrupted during work. By means of the plugin the developer is informed about vulnerabilities at an early stage of the development and it offers him the possibilty to fix them immeadiately which leads to more secure code.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Heutzutage werden Software-Systeme immer komplexer und sind im Alltag nicht mehr wegzudenken wie in Autos ([1]), Smart Devices etc. Die Vernetzung dieser Systeme nimmt stetig zu und bildet die tägliche Basis für Firmen, Forschung und Wirtschaft. Daher sind Security und Privacy Aspekte eine der größten Herausforderungen in dieser vernetzten Welt, so Mayer [19].

Das Bundesamt für Sicherheit in der Informationstechnik (BSI) berichtet sogar: "Studien zufolge werden pro Tag weltweit mehrere Tausend neue Computer gekapert und für fremde Zwecke missbraucht. Ein neu ans Internet angeschlossener PC wird bereits nach wenigen Minuten erstmals angegriffen." ([2]). Das bedeutet für die Software-Entwicklung dem Qualitätsmerkmal Sicherheit (ISO/IEC 9126) im gesamten Entwicklungszyklus eine hohe Priorität einzuräumen, denn die Auswirkungen einer Sicherheitslücke in Software-Systemen können von eher gering (Umgehung eines Kopierschutzes in einem Videospiel) bis hin zu desaströs sein, wie ein böswilliges Eindringen in ein Atomkraftwerkkontrollsystem, so Devanbu et al. [7].

Demzufolge ist es wichtig darauf zu achten, schon während der Entwicklung bereits bekannte Sicherheitslücken zu erkennen und zu beseitigen.

1.1 Problemstellung

Die händische Suche nach Schwachstellen ist sehr zeitintensiv und damit auch kostspielig. Zudem erfordert sie ein gewisses Maß an Expertenwissen, was die Identifikation von Sicherheitslücken im Quellcode angeht. Ein Entwickler müsste zum einen über Hintergrundwissen verfügen, um eigenständig Schwachstellen identifizieren zu können. Zum anderen müsste er auch sämtlichen Quellcode händisch überprüfen, den er nicht selbst geschrieben hat wie z.B. Bibliotheken von Drittanbietern.

Ein Entwickler würde also für die händische Suche nach Schwachstellen einige Zeit investieren müssen. Gleichzeitig hat er aber Fristen innerhalb von Software-Projekten einzuhalten. So wäre ein Entwickler möglicherweise

geneigt, die Schwachstellensuche kurz vor Abgabetermin zu erledigen oder ganz auszulassen, das wiederum könnte unsichere Software zur Folge haben.

1.2 Lösungsansatz

Die händische Suche soll mit Hilfe des im Rahmen dieser Arbeit entwickelten Security Code Clone Detectors automatisiert und so der Entwickler während des Schreibens von Java Code bereits über mögliche Schwachstellen informiert werden. Hierbei fließt das benötigte Wissen über mögliche Schwachstellen in ein Security Code Repository ein, welches vom Security Code Clone Detector zur Schwachstellensuche genutzt wird, wie in Abbildung 1.1 zu sehen. Diese Daten sollen dem Benutzer sowohl bei der Identifikation als auch Behebung gefundener Schwachstellen helfen.

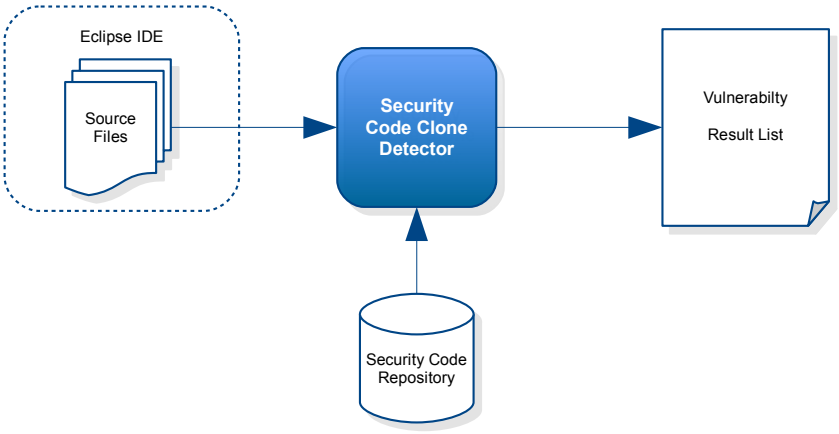

Abbildung 1.1: Schematische Darstellung Security Code Clone Detector

Die Suche nach Schwachstellen erfolgt beim Security Code Clone Detector auf Basis von Code Clones. Es wird also zum einen direkt nach Code Fragmenten gesucht von denen bekannt ist, dass sie ein oder mehrere Schwachstellen aufweisen, zum anderen müssen aber auch ähnliche Code Fragmente erkannt werden. Das sind Code Fragmente, die sich semantisch ähneln, aber z.B. andere Variablennamen verwenden oder zusätzliche bzw. weniger Statements enthalten.

Der Security Code Clone Detector wird als Eclipse Plugin realisiert. Das bietet den Vorteil, dass die Suche nach Schwachstellen von Anfang an in den Entwicklungsprozess integriert ist. Der Benutzer bekommt bereits während der Arbeit, also schon beim dem Schreiben von Code, Hinweise zu möglichen Schwachstellen, vgl. Abbildung 1.1.

1.3 Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert. Kapitel 1 gibt eine kurze Einleitung in das Thema, erläutert den Lösungsansatz und beschreibt die Struktur der Arbeit. In Kapitel 2 werden die Anforderungen für den Security Code Clone Detector detailliert beschrieben sowie auf die Stakeholder eingegangen. Die benötigten Grundlagen zum weiteren Verständnis dieser Arbeit werden in Kapitel 3 vermittelt. Hier werden Schwachstellen beschrieben und der Leser in das Thema Clone Detection eingeführt. Kapitel 4 widmet sich dem Konzept des Security Code Clone Detectors. Anschließend wird in Kapitel 5 dessen Implementation sowie Funktionsweise erläutert. Eine Evaluation des Security Code Clone Detectors erfolgt in Kapitel 6. Kapitel 7 beschäftigt sich mit verwandten Arbeiten und soll einen Überblick über den aktuellen Stand im Bereich Clone Detection vermitteln. Abschließend erfolgt im Kapitel 8 eine Zusammenfassung und ein Ausblick.

Kapitel 2

Anforderungen

In diesem Kapitel werden die Anforderungen an das Security Code Clone Detection Plugin erläutert, beginnend mit den Stakeholdern in Abschnitt 2.1. Daran anknüpfend werden die funktionalen und die nichtfunktionalen Anforderungen (2.2 und 2.3) aufgeführt und anschließend eine Priorisierung der Anforderungen in Abschnitt 2.4 vorgenommen. Zuletzt werden die daraus resultierenden Arbeitspakete in 2.5 vorgestellt.

2.1 Stakeholder

Das Eclipse Plugin zur Security Code Clone Detection richtet sich an die Zielgruppe der Software-Entwickler. Das Plugin soll sie dabei unterstützen, frühzeitig Sicherheitslücken im eigenen, von ihnen selbst geschriebenem Quellcode, zu erkennen. Dadurch soll schon während der Programmierung dafür gesorgt werden, sichereren Code zu schreiben und bekannte Schwachstellen gar nicht erst in den eigenen Code aufzunehmen. Zudem soll der Security Code Clone Detector die Entwickler, die nur teilweise über Security-Kenntnisse verfügen, bei der Schwachstellenanalyse ihres Programmcodes unterstützen. Hierbei ist es wichtig, dass die Software-Entwickler während der Arbeit nicht gestört oder gar behindert werden. Zu viele Irritationen, sei es durch Blockieren der IDE oder zu viele Falschmeldungen, könnten die Entwickler davon abhalten, das Plugin produktiv einzusetzen.

Eine weitere Zielgruppe sind Sicherheitsexperten. Also Personen, deren Aufgabe es ist, Sicherheit und Qualität in Software zu gewährleisten. Laut dem Artikel „Sicherheit im Software-Entwicklungsprozess“ von Matthias Rohr [25] können dies *Security-Tester* oder auch *Security Champions* sein. Laut Rohr sind Security-Tester „verantwortlich für die Durchführung von Sicherheitstests“ und ein Security Champion ist die „zuständige Person innerhalb eines Projektes bzw. Teams“. Sie könnten von der Nutzung des Plugins profitieren, da sie somit bekannte Schwachstellen im Code identifizieren und analysieren könnten.

Die dritte Zielgruppe sind Mitarbeiter und Studenten am Fachgebiet für Software Engineering. Das Plugin wurde als Forschungsprojekt im Rahmen dieser Masterarbeit entwickelt und für sie ist ein modularer Aufbau und gut dokumentierter Programmcode wichtig.

2.2 Funktionale Anforderungen

[R1] Die Software soll als Eclipse Plugin realisiert werden

Eclipse ist eine integrierte Entwicklungsumgebung, die häufig von Java-Entwicklern verwendet wird. Da sie sich durch Plugins erweitern lässt, kann die Suche nach Schwachstellen hier direkt in den Entwicklungsprozess integriert werden. Zur Entwicklung des Plugins wird die Eclipse-Version Oxygen.3 Release (4.7.3) verwendet und Java 1.8.0_171.

[R1.1] Das Plugin soll in der Sprache Java geschrieben werden

Das Fachgebiet für Software Engineering hat vorgegeben, dass das Plugin mit der Programmiersprache Java entwickelt werden soll.

[R2.1] Programmcode auf Schwachstellen untersuchen

Während des Entwicklungsprozesses soll der geschriebene Code auf Clones untersucht werden, die auf möglicherweise bekannte Schwachstellen hindeuten.

[R2.2] Java-Code auf Schwachstellen untersuchen

Als Vorgabe soll Programmcode der Programmiersprache Java auf Schwachstellen untersucht werden.

[R2.3] Bestehendes Wissen vorhandener (Code-) Datenbanken nutzen

Damit Quellcode-Clones gefunden werden können, wird Referenzcode benötigt. In diesem Fall Code, von dem bekannt ist, dass er eine Schwachstelle enthält. Es sollen hierbei keine Intraprojekt-Clones gesucht werden, also redundanter Code innerhalb eines Eclipse-Projekts, sondern ein lokales Security Repository muss eingebunden werden, welches Code mit Schwachstellen und zugehörigen Informationen dazu beinhaltet.

[R2.4] Der Entwickler soll bei der Arbeit nicht gestört werden

Eine Clone-Suche soll eigenständig im Hintergrund laufen, ohne dass der Entwickler seine Arbeit unterbrechen muss.

[R2.5] Optional: Prototypische Implementation für eine weitere Sprache

Es wäre schön, wenn prototypisch eine weitere Programmiersprache neben Java vom Plugin unterstützt werden würde.

[R3] Die Software meldet dem Benutzer, wenn Clones gefunden wurden

Es ist notwendig, dass der Benutzer nach einer Clone-Suche Feedback erhält, ob Clones gefunden wurden. Wenn keine Clones gefunden wurden erscheint dies ebenfalls in der Konsolenausgabe.

[R4] Bestehende Clone Detection Ansätze analysieren und ggf. kombinieren

Es sollen vorhandene Clone Detection Ansätze gesichtet und analysiert werden hinsichtlich ihrer Eignung für das Plugin. Zudem sollten bestehende Ansätze ggf. kombiniert werden, um zum Beispiel bessere Laufzeiten oder bessere Erkennungsraten zu erzielen sowie bessere Abdeckung der Clone-Typen.

[R5] Gut strukturierter und kommentierter Programmcode

Es wird vom Fachgebiet für Software Engineering gefordert, dass der Programmcode des Plugins gut strukturiert und kommentiert ist, damit Institutsmitarbeiter oder Studenten das Plugin weiterentwickeln können.

[R5.1] Modularer Aufbau der Software

Es wird gefordert, dass das Plugin modular aufgebaut ist, um Wartungs- sowie Erweiterungsarbeiten zu erleichtern.

[R6] Unterstützung der Clone-Typen 1 bis 3

Das Fachgebiet für Software Engineering fordert, das die Clone-Typen 1 bis 3 vom Plugin unterstützt werden.

[R7] Das Plugin soll konfigurierbar sein

Der Benutzer soll festlegen können, ob die automatische Clone-Suche nach jedem Speichervorgang der Quellcodedatei geschehen soll oder nach einem vorgegebenen Zeitintervall. Zudem muss der Benutzer die Pfade zum lokalen Security Respository festlegen können sowie das Arbeitsverzeichnis für den Clone Detector.

2.3 Nichtfunktionale Anforderungen

[NR1] Die Clone-Suche darf die Eclipse-UI nicht blockieren

Die Suche nach Schwachstellen muss als Hintergrundprozess ausgeführt werden, damit der Benutzer weiterhin die Eclipse-UI bedienen kann.

[NR2] Die Software soll erweiterbar sein

Es soll so die Möglichkeit geschaffen werden, weitere Programmiersprachen bei der Schwachstellensuche zu integrieren oder ggf. auch andere Clone Detection-Ansätze verwenden zu können.

[NR2.1] Durch eine einheitliche API sollen andere Clone Detection-Ansätze integriert werden können

Damit andere Clone Detection-Ansätze integriert werden können, muss ein einheitliches Application Programming Interface geschaffen werden, welches sowohl vom Plugin als auch vom Clone Detector implementiert wird.

[NR3] Die Clone-Suche soll auf Änderungen im Programmcode reagieren

Wenn der Entwickler neuen Code schreibt und das Dokument speichert, soll die Clone-Suche erneut gestartet werden.

[NR4] Die Clone-Suche soll möglichst performant sein

Damit der Entwickler möglichst früh gewarnt werden kann, soll die Suche nach Schwachstellen möglichst schnell durchgeführt werden. Dadurch soll früh in den Entwicklungsprozess eingegriffen werden können und so der Verbreitung von unsicherem Code entgegengewirkt werden. Als zeitliche Obergrenze sind hier maximal 2 Minuten pro Code-Datei vorgegeben.

[NR5] Ein vorhandenes Security Repository soll genutzt werden

Damit die Software Schwachstellen im Quellcode finden kann, wird ein Security Code Repository benötigt. Dort befinden sich Code-Dateien, die bekannte Schwachstellen enthalten. Anhand dieser kann die Software dann nach sogenannten Security Code Clones suchen. Für diese Anforderung wird der Security Code Exporter für Github genutzt. Weitere Informationen dazu sind in Abschnitt 4.5 zu finden.

[NR6] Der Benutzer erhält nach einer Clone-Suche eine Liste mit Informationen zu gefundenen Schwachstellen

Damit der Benutzer weiß, ob und wo Clones gefunden wurden, sollen Informationen dazu bereitgestellt werden in Form einer detaillierten Liste. Darin enthalten sind welche Schwachstelle gefunden wurden - sofern vorhanden mit CVE Informationen. An welcher Stelle sie im eigenen Code (Zeilenangaben) zu finden sind und der Name der entsprechenden Methode, die das Code Fragment enthält.

[NR7] Optional: Der Benutzer erhält per Mausklick Detailinformationen zu Schwachstellen

Sind zu einer gefundenen Schwachstelle weitere Informationen verfügbar, soll der Benutzer diese per Mausklick abrufen können. Diese Detailinformationen enthalten die CVE Details wie Schweregrad, eine kurze Zusammenfassung des Problems, mögliche Fixes und ggf. auch Exploits. Ein Link zur CVE soll ebenfalls enthalten sein.

[NR8] Code Clones werden direkt in der UI farblich hervorgehoben

Damit der Entwickler möglichst rasch ohne langes Suchen die gefundenen Schwachstellen im Code lokalisieren kann, sollen die betroffenen Code-Fragmente bzw. Methoden (die den Code enthalten) farblich hervorgehoben werden.

[NR9] Die Software soll portabel sein

Der Nutzerkreis des Plugins soll nicht auf ein bestimmtes Betriebssystem wie z.B. Windows beschränkt sein, sondern es sollen neben Windows zusätzlich Linux und Mac OS unterstützt werden.

2.4 Priorisierung der Anforderungen

Die Priorisierung der erhobenen Anforderungen erfolgt nun anhand des Kano-Modells der Kundenzufriedenheit (siehe Kano-Theorie der Kundenzufriedenheitsmessung von Jörg A. Hölzing [11]). „Das Kano-Modell zeigt den Zusammenhang zwischen der Kundenzufriedenheit und der Erfüllung von Kundenanforderungen.“, nach Jochem [14]. Die Kundenanforderungen werden hierbei in drei Gruppen unterteilt. Die Basisanforderungen, Leistungsanforderungen und die Begeisterungsanforderungen wie in Abbildung 2.1 zu sehen.

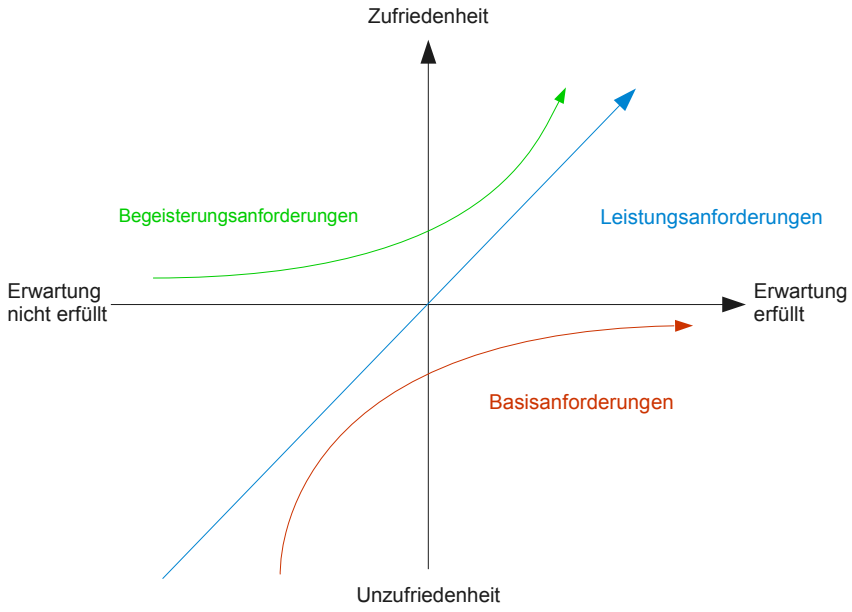

Abbildung 2.1: Kano-Modell in Anlehnung an [11], S.85

Die Basisanforderungen haben eine hohe Erfüllungspflicht in einem Produkt. Ohne diese kann es keine Kundenzufriedenheit geben. Jedoch wird die alleinige Berücksichtigung der Basisanforderungen nicht zur vollen Zufriedenheit des Kunden beitragen. Wie im Modell zu sehen, nähern sie sich asymptotisch der „Erwartung erfüllt“-Achse. D.h. immer mehr erfüllte Basisanforderungen steigern die Kundenzufriedenheit nicht immer weiter. Zudem ist die Erhebung dieser Anforderungen nicht ganz einfach, da der Kunde diese möglicherweise als selbstverständlich ansieht.

Zur Erreichung der Kundenzufriedenheit sind die Leistungsanforderungen notwendig. Im Kano-Modell als blaue Linie dargestellt. Im Gegensatz zu den Basisanforderungen, wird der Kunde immer zufriedener, je mehr

Leistungsanforderungen berücksichtigt werden. Dabei werden diese durch den Kunden vorgegeben.

Die dritte Gruppe bilden die Begeisterungsanforderungen. Sie tragen stark zur Kundenzufriedenheit bei und sind dem Kunden noch gar nicht bewusst, dass er sie braucht. Durch die Begeisterungsanforderungen kann ein Produkt sich von etwaiger Konkurrenz abheben.

Die Priorität der Basisanforderungen beim Security Code Clone Detector ist hoch. Sie müssen vom Plugin erfüllt werden. Die Leistungsanforderungen werden je nach Wichtigkeit mit mittlerer bis hoher Priorität gewichtet. Geringe bis mittlere Priorität werden den Begeisterungsanforderungen beigemessen. Realisiert werden diese zum Schluss, wenn die wichtigen Kundenanforderungen erfüllt worden sind.

In Tabelle 2.1 sind die Anforderungen nach dem Kano-Modell klassifiziert.

[R2.1], [R2.2], [R4], [NR1] und [NR4] sind Basisanforderungen. Ohne diese ist der Security Code Clone Detector nicht oder nur eingeschränkt nutzbar bzw. realisierbar. [R1] und [R1.1] sind vom Kunden geforderte Leistungsanforderungen. Hierfür ist eine Einarbeitung in die Eclipse Plugin-Entwicklung notwendig. Da ein Großteil weiterer Anforderungen hierauf beruhen, ist die Priorität hoch. [R2.3] bis [R3] sind ebenfalls als hoch priorisiert, da sie geforderte Kernfunktionen des Plugins darstellen.

Als mittel eingestufte Leistungsanforderungen sind [R5] bis [NR9]. [R5] und [R5.1] ermöglichen eine Austauschbarkeit des zu verwendenden Clone Detectors und [NR9] sorgt für Portabilität bei Verwendung verschiedener Betriebssysteme. [NR5] erhält die Priorität hoch. Damit das Plugin Schwachstellen finden kann, wird ein Security Code Repository benötigt, das Code mit bekannten Schwachstellen enthält. Der Security Code Exporter für Github lädt diese Daten herunter, so dass das Plugin diese zur Schwachstellensuche nutzen kann. [R6] bis [NR2.1] sind Begeisterungsanforderungen mit Priorität mittel. Sie führen zu sehr hoher Kundenzufriedenheit. [R2.5] bis [NR8] sind ebenfalls Begeisterungsanforderungen mit niedriger Priorität. Sie werden zum Schluss realisiert, sofern genügend Zeit zur Verfügung steht.

2.5 Arbeitspakete

Die verschiedenen Arbeitspakete ergeben sich aus den priorisierten Anforderungen. Die Arbeitspakete bauen aufeinander auf und sind ebenfalls absteigend der Priorität nach geordnet.

Clone Detector-Ansätze

Zur Erfüllung von [R4] ist eine Sichtung und Einarbeitung in vorhandene

Anforderung	**Klassifikation**	**Priorität**
R2.1	Basisanforderung	Hoch
R2.2	Basisanforderung	Hoch
R4	Basisanforderung	Hoch
NR1	Basisanforderung	Hoch
NR4	Basisanforderung	Hoch
R1	Leistungsanforderung	Hoch
R1.1	Leistungsanforderung	Hoch
R2.3	Leistungsanforderung	Hoch
R2.4	Leistungsanforderung	Hoch
R3	Leistungsanforderung	Hoch
R5	Leistungsanforderung	Mittel
R5.1	Leistungsanforderung	Mittel
NR5	Leistungsanforderung	Hoch
NR9	Leistungsanforderung	Mittel
R2.5	Begeisterungsanforderung	Niedrig
R6	Begeisterungsanforderung	Mittel
R7	Begeisterungsanforderung	Mittel
NR2	Begeisterungsanforderung	Mittel
NR2.1	Begeisterungsanforderung	Mittel
NR3	Begeisterungsanforderung	Niedrig
NR6	Begeisterungsanforderung	Niedrig
NR7	Begeisterungsanforderung	Niedrig
NR8	Begeisterungsanforderung	Niedrig

Tabelle 2.1: Anforderungen klassifiziert nach dem Kano-Modell

Code Clone Detection-Ansätze notwendig. Dieses Paket stellt die Basis für das gesamte Plugin und somit auch für [R2.1] und [R2.2] dar.

Eclipse Grundlagen

Damit die Anforderung [R1] erfüllt werden kann, erfolgt eine Einarbeitung in das Eclipse-Plugin-Framework.

Modularität und API

Damit die Modularität und Erweiterbarkeit des Plugins gewährleistet ist - wie in [R5.1], [NR2] und [NR2.1] gefordert -, muss eine einheitliche API entwickelt und von den einzelnen Modulen implementiert werden. Dabei ist stets [R5] zu berücksichtigen.

Security Code Repository

Als Basis für die Schwachstellensuche dient das Security Code Repository, welches vom *Security Code Exporter für Github* angelegt wird. Es muss sichergestellt sein, dass das Plugin diese verarbeiten und nutzen kann [NR5].

Schwachstellensuche

Die Suche nach möglichen Schwachstellen folgt der Umsetzung nach [R6], [NR1], [NR3] und [NR4].

Visualisierung der Clone-Suche

Für eine gute Usability werden die Anforderungen [R3], [NR6], [NR7] und [NR8] umgesetzt.

Kapitel 3

Grundlagen

Die Grundlagen, die zum weiteren Verständnis des Security Code Clone Detectors nötig sind, sollen in diesem Kapitel vermittelt werden. Dafür beschäftigt sich zuerst der Abschnitt 3.1 mit der Definition des Begriffs der Schwachstelle. Anschließend werden die Grundlagen zum Thema Clone Detection in Abschnitt 3.2 diskutiert.

3.1 Schwachstelle

Zuerst soll definiert werden, was im Rahmen dieser Arbeit unter dem Begriff der Schwachstelle zu verstehen ist. Diese Erläuterungen beruhen auf der Arbeit „Modeling the Security Ecosystem - The Dynamics of (In)Security“ von Frei et al. [9]. Schwachstellen zu definieren ist eine nicht ganz einfache Aufgabe, denn sie ist stark abhängig von den Beteiligten und deren Absichten bzw. Zielen, so Frei et al. Betrachtet man beispielsweise ein bestimmtes Software-Problem als *Fehler*, *Feature* oder *Schwachstelle* hängt davon ab, ob man mit einem Forscher, dem Anbieter oder verschiedenen Usern der Software spricht.

Die Definition des Begriffs „Schwachstelle“ erfolgt nun anhand des Common Vulnerabilities and Exposures (CVE) Konsortiums (siehe Abschnitt 3.1.2):

> „A »vulnerability« is a weakness in the computational logic (e.g., code) found in software and some hardware components (e.g., firmware) that, when exploited, results in a negative impact to confidentiality, integrity, OR availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety).“ [1]

[1] `https://cve.mitre.org/about/terminology.html`, [11.04.2018]

Der Definition der CVE nach, ist eine Schwachstelle also ein Fehlverhalten in Software (aber auch Hardware), dass bei Ausnutzung negative Auswirkungen auf Vertraulichkeit, Integrität oder Verfügbarkeit hat. Behoben werden kann die Schwachstelle - und darauf liegt der Fokus dieser Arbeit - typischerweise durch Änderungen im Code.

Ein Beispiel für eine Schwachstelle ist in Code-Listing 3.1 dargestellt. In diesem Beispiel kann ein Angreifer mittels *SQL Injection* eine erfolgreiche Authentifizierung auch ohne ein gültiges Passwort erreichen. SQL Injections gehören zur Gruppe der *code-injection* Angriffe, bei denen Daten, die vom Benutzer des Systems kommen in eine SQL-Abfrage eingebunden werden [10].

```
1  public boolean login(String user, String pw) throws SQLException
2  {
3    Statement sm = connection.createStatement();
4    String query = "SELECT id FROM Users WHERE user= '" + user + "' AND pw='
       " + pw + "'";
5    ResultSet result = sm.executeQuery(query);
6    boolean isLoggedIn = result.next();
7
8    result.close();
9    sm.close();
10
11   return isLoggedIn;
12 }
```

Code-Listing 3.1: Schwachstelle mit SQL Injection

Zeile 4 im Code-Listing 3.1 enthält die SQL-Abfrage, in die ohne eine Prüfung der Daten die Parameter `user` und `pw` eingebunden sind. Anschließend wird dieses zusammengesetzte SQL-Statement in Zeile 5 ausgeführt. Falls ein Benutzer mit dem Namen `user` und dem zugehörigem Passwort `pw` existiert, wird die Datensatz-ID zurückgegeben und die Authentifizierung des Benutzers war erfolgreich (Variable `isLoggedIn == true`).

Angenommen es existiert der Benutzer „Alice“ mit zugehörigem Passwort „123456“. Ein Aufruf der Methode `login(„Alice“, „123456“)` gibt `true` zurück. Der Aufruf `login(„Alice“, „Mallory“)` würde `false` liefern und es würde keine erfolgreiche Authentifizierung am System erfolgen. Wie nun diese Schwachstelle auszunutzen ist und wie sie behoben werden kann, ist in den Abschnitten 3.1.3 respektive 3.1.4 zu lesen.

Die Begriffe Schwachstelle und Sicherheitslücke werden in dieser Arbeit synonym verwendet.

3.1.1 Lebenszyklus einer Schwachstelle

Der Lebenszyklus einer Schwachstelle[2] $v \in V$ (wobei V Menge aller Schwachstellen) besteht aus sechs Ereignissen: *creation*, *discovery*, *exploit availability*, *disclosure*, *patch availability* und *patch installation* (zu sehen in Abbildung 3.1).

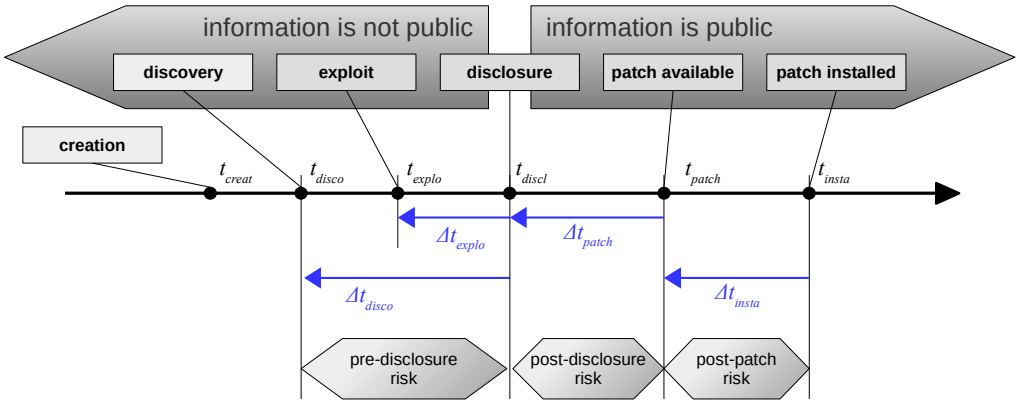

Abbildung 3.1: Vulnerability Lifecycle in Anlehnung an [9]

Frei et al. gehen beim *creation* event (t_{creat}) davon aus, dass Schwachstellen im Allgemeinen versehentlich als Resultat von Programmierfehlern auftreten. Das *discovery*-Ereignis (t_{disco}) ist der früheste Zeitpunkt, zu dem die Sicherheitslücke entdeckt wird. In der Regel ist dieser Zeitpunkt nicht öffentlich bekannt, sondern die Öffentlichkeit erfährt erst während des (public) *disclosure* events (t_{discl}) davon. Das *exploit*-Ereignis (t_{explo}) ist der früheste Zeitpunkt, zu dem ein Exploit (Abschnitt 3.1.3) für die Schwachstelle verfügbar ist. Zum Zeitpunkt *patch availability* (t_{patch}) existiert (frühestens) ein Patch bzw. Fix (3.1.4), der vom Hersteller herausgegeben wird. Das Event *patch installation* (t_{insta}) definiert den Zeitpunkt, an dem die User von der Beseitigung der Schwachstelle profitieren können. Dieser Zeitpunkt ist nicht eindeutig definierbar, da zwischen Nutzergruppen (corporate und home users) unterschieden werden muss.

Während der gesamten Zeit zwischen discovery und patch installed kann die Sicherheitslücke von Angreifern ausgenutzt werden und ein System ist somit potenziell gefährdet. Die von Frei et al. benannte *pre-disclosure risk*-Phase lässt sich durch

$$\Delta t_{disco}(v) = t_{discl}(v) - t_{disco}(v)$$

beschreiben. Also die Zeit von der Entdeckung der Sicherheitslücke bis zur Veröffentlichung. Während dieser Zeit ist die Schwachstelle einer unbekann-

[2] Vulnerability Lifecycle nach Frei et al. [9]

ten Gruppe bekannt. Das können Hersteller und Forscher sein, aber auch Angreifer, die dieses Schwachstelle eigenständig entdeckt haben.

Die *post-disclosure*-Phase wird durch

$$\Delta t_{patch}(v) = t_{patch}(v) - t_{discl}(v)$$

ausgedrückt. Hiermit ist der Zeitraum von der Veröffentlichung bis zum Patch Release durch den Hersteller gemeint. Abschließend folgt die *post-patch*-phase (Δt_{insta}). Auf die Länge der Periode

$$\Delta t_{insta}(v) = t_{insta}(v) - t_{patch}(v)$$

kann der User aktiv Einfluss nehmen, in dem er den Zeitpunkt zur Installation des Patches wählt.

In welcher der Phasen kann nun der Security Code Clone Detector zur Sicherheit von Software beitragen? Der Security Code Clone Detector nutzt bereits veröffentlichte Sicherheitslücken der CVE. Somit schützt er vor Schwachstellen der Risikogruppe *post-disclosure*, sowie der *post-patch*-Risikogruppe. Bei letzterer ist der Security Code Clone Detector auch in der Lage - sofern diese Daten im Security Repository vorliegen - dem Benutzer das Code Fragment eines möglichen Patches anzuzeigen. Dieses könnte dann vom Entwickler implementiert und somit die Schwachstelle beseitigt werden.

Sollten einem Unternehmen oder dem Benutzer nun aber auch Informationen über nicht veröffentlichte Sicherheitslücken vorliegen, können diese Daten ebenfalls in das Security Repository eingebunden werden. Somit könnte der Security Code Clone Detector sogar die dritte Risikogruppe (*pre-disclosure risk*) abdecken.

3.1.2 Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures (CVE) ist ein *de facto* Industriestandard, der große Akzeptanz in der Sicherheitsindustrie, im akademischen Umfeld und bei einer Vielzahl von Regierungsorganisationen erreicht hat seit der Gründung 1999, so Frei et al. [9].

Die gemeinnützige MITRE Corporation, Betreiber der CVE, beschreibt die Common Vulnerabilities and Exposures als Liste von einheitlichen Kennungen öffentlich bekannter Cybersecurity Schwachstellen.[3] Vor der Gründung 1999 gab es kein einheitliches System zur Kennzeichnung von Schwachstellen. Es war bis dahin nicht ersichtlich, ob die verschiedenen Datenbanken das gleiche Problem adressierten. Das hatte zur Folge, dass die Informationsbeschaffung zu einer bestimmten Sicherheitslücke stark erschwert wurde.

Eine Lösung hierfür stellt die CVE Liste bereit, in dem Sicherheitslücken eine eindeutige Kennung (CVE ID) erhalten und zentral in einer öffentlich

[3] `https://cve.mitre.org/about/`, [11.04.2018]

zugänglichen Datenbank beschrieben werden. Eine Risikobewertung von Schwachstellen nimmt die CVE selbst nicht vor. Sie beschreibt sich selbst eher als „a dictionary rather than a database“.

Eine CVE ID hat die Form `CVE-1999-0067`. Bestehend aus `CVE prefix + Year + Arbitrary Digits` in variabler Länge.[4]

Neben CVE IDs nutzt der Security Code Clone Detector noch das Common Vulnerability Scoring System (CVSS). Das ist ein Industriestandard zur Bewertung des Schweregrades (severity level) von Sicherheitslücken. Hierfür kommt das CVSS v2.0 Rating zum Einsatz, dass eine Schwachstelle auf Basis des CVSS-Base-Score in eine der drei Kategorien „Low“, „Medium“ und „High“ einordnet, wie in Tabelle 3.1 dargestellt.

Severity	CVSS-Base-Score
Low	0.0-3.9
Medium	4.0-6.9
High	7.0-10.0

Tabelle 3.1: CVSS v2.0 Ratings

Der Security Code Clone Detector lädt die CVE Daten nicht eigenständig aus dem Internet herunter. Die Daten liegen im Security Repository vor und werden beim Tokenizing als Metadaten ausgelesen und gespeichert. Weitere Informationen dazu sind in Kapitel 4 zu finden.

3.1.3 Exploit

Ein Exploit ist ein Stück Software, ein Virus, eine Menge von Daten oder eine Sequenz von Kommandos, die eine Sicherheitslücke ausnutzen mit dem Ziel, unbeabsichtigtes oder unvorhergesehenes Verhalten der Software oder eines embedded devices zu provozieren, sagen Frei et al. [9].

Bei der Definition des Begriffs Schwachstelle wurde das Beispiel der SQL Injection (3.1) eingeführt. Ein möglicher Exploit hierfür sieht wie folgt aus:

```
login("Mallory' OR 1=1 --", "");
```

Dem Parameter `user` wurde hier ein SQL Kommando übergeben, statt nur eines Benutzernamens. Der Passwortparameter `pw` wurde leer gelassen. Somit steht in der String-Variable `query` (Zeile 4 im Code-Listing 3.1) das folgende SQL Kommando

```
SELECT id FROM Users WHERE user='Mallory' OR 1=1 -- AND pw=''
```

Beim Ausführen dieses Kommandos ist es ohne Belang, ob ein Eintrag mit dem Benutzernamen „Mallory“ in der Datenbank vorhanden ist, da die `WHERE`-Klausel durch das `OR 1=1` immer wahr ist. Die `AND`-Bedingung wurde durch

[4] `https://cve.mitre.org/cve/identifiers/syntaxchange.html`, [11.04.2018]

die Doppelstriche (--) auskommentiert. Ein Angreifer würde durch diesen Aufruf vom System erfolgreich authentifiziert werden.

3.1.4 Patch

Nicastro unterscheidet in Ihrem Buch „Security Patch Management“ [22] drei Typen von Patches. Den *Feature* Patch, den *Functionality* Patch und den *Security* Patch. Der Functionality Patch adressiert eine vorhandene Funktion in einer Software und zielt dabei nicht auf die Behebung einer Schwachstelle ab. In einigen Fällen auch „bug fix“ genannt, so Nicastro.

Der Feature Patch führt hingegen ein neues Feature oder eine neue Funktion ein und erweitert eine vorhandene Software entsprechend. Auf den ersten Blick scheinen sie dem Functionality Patch zu ähneln, sie führen jedoch ein neues Feature etwas bereits Existierendem hinzu, meint Nicastro.

Wenn im Rahmen dieser Arbeit von Patch oder auch synonym dafür von *Fix* gesprochen wird, ist immer ein Security Patch gemeint. Security Patches dienen immer der Behebung bzw. Beseitigung von bekannten Schwachstellen.

Die Schwachstelle im eingangs eingeführten Beispiel (3.1), kann durch einen (Security) Patch behoben werden. Zu sehen ist dieser Patch in Code-Listing 3.2.

```
1  public boolean login(String user, String pw) throws SQLException
2  {
3    PreparedStatement sm = coneection.prepareStatement(
4        "SELECT id FROM Users WHERE user=? AND pw=?");
5    sm.setString(1, user);
6    sm.setString(2, pw);
7    ResultSet result = sm.executeQuery();
8    boolean isLoggedIn = result.next();
9
10   result.close();
11   sm.close();
12
13   return isLoggedIn;
14 }
```

Code-Listing 3.2: Fix gegen SQL Injection

Anstatt der `Statement`-Klasse wird hier die Klasse `PreparedStatement` verwendet (Zeile 3). Ein Prepared Statement bietet neben der Tatsache, dass es bereits „precompiled“ zum Database Management System (DBMS) gesendet wird und damit dort direkt ausgeführt werden kann den großen Vorteil, dass die zu übergebenden Parameter *escaped* werden und so vor SQL Injections schützen können. Vorausgesetzt man nutzt die setter-Methoden für die Parameter. Das ist in diesem Fix der Fall (vgl. Zeile 5-6). Der im Abschnitt 3.1.3 verwendete Exploit würde hier nicht funktionieren, da der Wert für den `user`-Parameter „`Mallory' OR 1=1 --`“ als Ganzes interpretiert werden würde. Somit würde das System hier keine erfolgreiche Authentifizierung zulassen.

3.2 Code Clone Detection

Die in diesem Abschnitt verwendeten Erläuterungen und Definitionen stützen sich auf die Arbeiten von Sajnani et al. [28] und Sheneamer et al. [29].

Wenn ein Programmierer Quellcode per Copy & Paste wiederverwendet nennen Sheneamer et al. das *code cloning*. Dabei ist es unerheblich, ob kleinere oder auch größere Änderungen am Code vorgenommen wurden. Code Clones bringen Schwierigkeiten mit sich, so Sheneamer weiter, denn sie können zu Fehlerverbreitungen führen. Angenommen es existieren viele identische oder nahezu identische Copy & Paste Quellcodefragmente und in einem wird ein Fehler gefunden, so muss dieser Fehler in allen Code Clones ausfindig gemacht und beseitigt werden.

Code Clone Detection beschäftigt sich mit der Suche nach Code Clones und lokalisiert identische oder ähnliche Codeteile. Dabei kann Code Clone Detection für ganz unterschiedliche Einsatzzwecke genutzt werden. Da wären zum einen die Vereinfachung und Wartbarkeit von Code. Zum anderen aber auch die Plagiatssuche, die Suche nach Copyright-Verstößen wie auch die Suche nach Schwachstellen im Code. Letztere bildet den Kern dieser Arbeit. Die Begriffe Clone Detection und Clone-Suche werden in dieser Arbeit synonym verwendet.

Im Folgenden werden nun einige Begriffe im Kontext der Code Clone Detection erklärt.

3.2.1 Code Fragment

Ein Code Fragment (CF) ist ein Teil eines Quellcodes, das zum Ausführen eines Programms benötigt wird. Es besteht aus mehreren Statements und kann eine Funktion bzw. Methode enthalten, `begin-` und `end`-Blöcke sowie eine ganze Sequenz von Statements. Code-Listing 3.2 stellt ein solches Code Fragment dar, ebenso wie beispielsweise auch nur die enthaltenen Zeilen 5-6.

3.2.2 Token

Ein Token ist die kleinste Einheit, die von einem *Compiler* verstanden werden kann. Als Beispiel sei das Statement in Code-Listing 3.3 gegeben.

```
1 int n = 0;
```

Code-Listing 3.3: Token Beispiel-Statement

Es besteht aus fünf Tokens: `int`, `n`, `=`, `0` und `;`.

3.2.3 Code Clone/Clone Pair

Wenn ein Code Fragment CF_1 ähnlich zu einem anderen Code Fragment CF_2 ist (syntaktisch oder semantisch), ist das eine ein Code Clone vom anderen. Wenn beide in Relation zueinander stehen, d.h. beide gleich oder ähnlich sind, spricht man vom Clone Pair (CF_1, CF_2).

3.2.4 Security Code Clone

In dieser Arbeit werden Code Clones immer im Sicherheitskontext betrachtet. Das heißt, der Begriff *Security Code Clone* stellt definitionsgemäß immer einen Code Clone dar. Die Begriffe Security Code Clone, Code Clone und Clone werden im Rahmen dieser Arbeit als Synonyme verwendet.

3.2.5 Code Clone-Typen

Sheneamer et al. [29] unterteilen Clones in zwei Gruppen. In der ersten Gruppe finden sich Code Fragmente, die basierend auf ihrem Text gleich sind. Folgende drei Clone-Typen gehören der ersten Gruppe an. Die verschiedenen Clone-Typen sind zum besseren Verständnis in Abbildung 3.2 dargestellt.

Typ-1 Clone

Zwei Code Fragmente sind exakte Kopien voneinander. Ausgenommen sind Whitespaces, Leerzeichen und Kommentare.

Typ-2 Clone

Zwei Code Fragmente sind ähnlich und unterscheiden sich zusätzlich zu Typ-1 Clones durch die Namen ihrer Variablen, Literale und Funktionen.

Typ-3 Clone

Zwei Code Fragmente sind syntaktisch ähnlich, allerdings beinhalten sie Modifikationen, wie hinzugefügte oder entfernte Statements, unter Berücksichtigung der Typ-1 und Typ-2 Clone Definitionen.

Typ-4 Clone

In die zweite Gruppe fallen die Typ-4 Clones. Diese werden auch semantische Clones genannt, da sie syntaktisch keinerlei Ähnlichkeiten aufweisen, sondern die Code Fragmente sich semantisch ähnlich sind.

Ursprüngliches Code Fragment CF_0

```
for(i = 0; i < 10; i++)
{
    // foo 2
    if (i % 2 == 0)
        a = b + i;
    else
        // foo 1
        a = b - i;
}
```

CF_1 – Typ-1 Clone

```
for(i = 0; i < 10; i++)
{
    if (i % 2 == 0)
        a = b + i; // comment 1
    else
        b = b - i; // comment 2
}
```

CF_2 – Typ-2 Clone

```
for(j = 0; j < 10; j++)
{
    if (j % 2 == 0)
        y = x + j; // comment 1
    else
        y = x - j; // comment 2
}
```

CF_3 – Typ-3 Clone

```
for(i = 0; i < 10; i++)
{
    // new statement
    a = 10 * b;
    if (i % 2 == 0)
        a = b + i; // comment 1
    else
        a = b - i; // comment 2
}
```

CF_4 – Typ-4 Clone

```
while(i < 10)
{
    // a comment
    a = (i % 2 == 0) ? b + i : b - i;
    i++;
}
```

Abbildung 3.2: Clone-Typen 1 bis 4

Beispiel der verschiedenen Clone-Typen

Die Abbildung 3.2 zeigt links oben das ursprüngliche Code Fragment CF_0. Rechts daneben ist der Typ-1 Clone im CF_1 dargestellt. Hier wurden die Kommentare geändert und an anderer Stelle im Code eingefügt. Auf unterschiedliches Einrücken wurde hier zu Gunsten besserer Lesbarkeit verzichtet. Im CF_2 - dem Typ-2 Clone vom CF_0 - wurden alle Variablen umbenannt, gemäß Definition (3.2.5). Der Typ-3 Clone, hier CF_3 enthält ein zusätzliches Statement gegenüber CF_0 und im semantisch ähnlichen Typ-4 Clone wurde die `for`-Schleife gegen eine `while`-Schleife ausgetauscht. Zudem wurde das `if then else` Statment gegen sein semantisches Äquivalent ausgetauscht. Gut zu sehen ist hier, dass sich die beiden Code Fragmente CF_0 und CF_4 auf den ersten Blick nicht ähneln. Erst bei genauerer Betrachtung erkennt man, dass beide semantisch äquivalent sind.

3.2.6 Clone Detection-Techniken

Beim Detection-Verfahren, mit dem Clone-Detektoren arbeiten, differenzieren Sheneamer et al. zwischen vier verschiedenen Ansätzen. Über diese Ansätze soll nun im Folgenden ein kurzer Überblick auf Grundlage von [29] gegeben werden. Die einzelnen Ansätze sollen dabei nicht erschöpfend diskutiert werden, sondern im Fokus steht, dem Leser einen Überblick über die verschiedene Techniken zu verschaffen.

1. Textueller Ansatz

Textbasierte Ansätze sind auf Typ-1 Clones beschränkt. Sie vergleichen zwei Code Fragmente auf Ähnlichkeit ihrer Code Strings und sind sowohl einfach zu implementieren, wie auch unabhängig von der verwendeten Sprache.

2. Lexikalischer Ansatz (Token-basiert)

Beim lexikalischen Ansatz werden die Sourcecode-Zeilen in Tokens eingeteilt. Daher wird dieser Ansatz auch Token-basierter Ansatz genannt. Der Vorteil besteht hier, dass dieses Verfahren mehrere Typen von Clones abdeckt im Gegensatz zum textuellen Ansatz und bessere Werte für Precision und Recall aufweist. Gleichzeitig erfordert der Token-basierte Ansatz mehr Speicherplatz und ebenfalls eine höhere Laufzeit als der textuelle Ansatz.

Mit dem Begriff *Token* ist in dieser Arbeit eine Zeichenkette (String) gemeint. Dabei handelt es sich um eine Zahl, einen Bezeichner (Identifier), ein Schlüsselwort oder ein Literal. Der zum Erzeugen der Tokens nötige *Tokenizer* muss nicht zwangsläufig an eine bestimmte (Programmier-)Sprache gebunden sein.

3. Syntaktischer Ansatz

Den syntaktischen Ansatz unterteilen Sheneamer et al. in zwei Kategorien. Die erste Gruppe ist die *Tree-based Clone Detection-Technik*. Hierbei wird der Sourcecode in einen Abstract Syntax Tree (AST) geparsed und anschließend mit tree-matching Algorithmen nach Clones gesucht. Die zweite Gruppe ist die *Metric-based Clone Detection-Technik*. Die Suche nach Code Clones erfolgt hier anhand verschiedener Metriken. Welche Metriken das im Detail sind, hängt vom verwendet Clone Detector ab. Sheneamer et al. geben dazu in [29] einen Überblick. Den Autoren nach haben die Tree-basierten Techniken den Nachteil, dass keine Identifier und Literale berücksichtigt werden. Es können somit keine Änderungen der Statement-Reihenfolge als Clones erkannt werden. Die Metrik-basierten Techniken benötigen einen Parser oder einen *Program Dependency Graph* (PDG), um an die Metrikwerte zu gelangen. Zudem sind zwei Code Fragmente mit den gleichen Metrikwerten nicht automatisch ähnliche Code Fragmente.

4. Semantischer Ansatz

Beim semantischen Ansatz können Code Fragmente gefunden werden, die die gleichen Berechnungen anstellen. Also semantische Äquivalenz aufweisen. Das geschieht beispielsweise mit Hilfe von PDGs. Dieser Ansatz eignet sich für das Aufspüren von Typ-4 Clones. Des Weiteren gehören auch hybride Verfahren dazu, so Sheneamer et al. Damit sind Verfahren gemeint, die unterschiedliche Ansätze miteinander kombinieren, um damit Probleme einzelner Verfahren zu überwinden.

Nun folgt Kapitel 4 und stellt das Konzept des Security Code Clone Detectors vor.

Kapitel 4

Konzept des SCC Detectors

Dieses Kapitel beschreibt das Konzept des Security Code Clone Detectors (SCC Detector). Dafür wird in Abschnitt 4.1 der Clone Detector als eine der Kernkomponenten des entwickelten Plugins vorgestellt. Es wird die Arbeitsweise des Clone Detectors erläutert, ebenso wie der eigens dafür entwickelte Tokenizer. Darüber hinaus wird die Entscheidung diskutiert, warum genau dieser Clone Detector als Basis genutzt wird.

Anschließend wird der Security Code Clone Detector mit seinen Komponenten in Abschnitt 4.4 näher beleuchtet und dessen Arbeitsweise erörtert. Zum Abschluss dieses Kapitels wird in Abschnitt 4.5 der „Security Code Exporter für Github“ vorgestellt. Dieser dient dazu, dass benötigte Security Code Repository mit Daten zu existierenden Schwachstellen zu versorgen.

4.1 Clone Detector - SourcererCC

SourcererCC[1] ist ein von Sajnani et al. in [28] vorgestellter Token-basierter Clone Detector. Die Motivation der Autoren war, einen Clone Detector zu entwickeln, der möglichst viele verschiedene Clone-Typen abdeckt und bei großen Projekt-Repositories skalierbar ist.

Die Unterstützung von Typ-1 bis Typ-3 Clones sowie die Skalierbarkeit und Ausführungsgeschwindigkeit, gerade bei großen Datenmengen, zeichnen SourcererCC als Komponente für den Security Code Clone Detector aus. Darüber hinaus erreicht SourcererCC sehr hohe Werte für Precision und Recall im Gegensatz zu anderen Clone-Detektoren, wie Sajnani et al. in ihrer Evaluation (Kapitel 4, [28]) zeigen. Ein weiterer Punkt ist, dass SourcererCC komplett in Java geschrieben ist und hier keine Schnittstelle zu einer anderen Programmiersprache realisiert werden musste. Zusätzlich benötigt der Clone Detector keine speziell aufbereiteten Daten zur Suche nach Code Clones, wie es andere Ansätze erfordern (siehe Kapitel 7). Anforderung [R4] „Bestehende

[1] `http://sourcerer.ics.uci.edu/`

Clone Detection Ansätze analysieren und ggf. kombinieren“ ist somit erfüllt, wobei die Kombination verschiedener Ansätze hier nicht nötig war.

Wie arbeitet der Clone Detector SourcererCC? Diese Frage möchte ich im Folgenden beantworten, indem ich einen kurzen Überblick über die Funktionsweise gebe. Es soll hier keine ausführliche Detailbeschreibung erfolgen, sondern ich möchte dem Leser lediglich einen Überblick für das weitere Verständnis verschaffen. Detailliertere Informationen sind „SourcererCC: Scaling Code Clone Detection to Big Code“ ([28]) zu entnehmen.

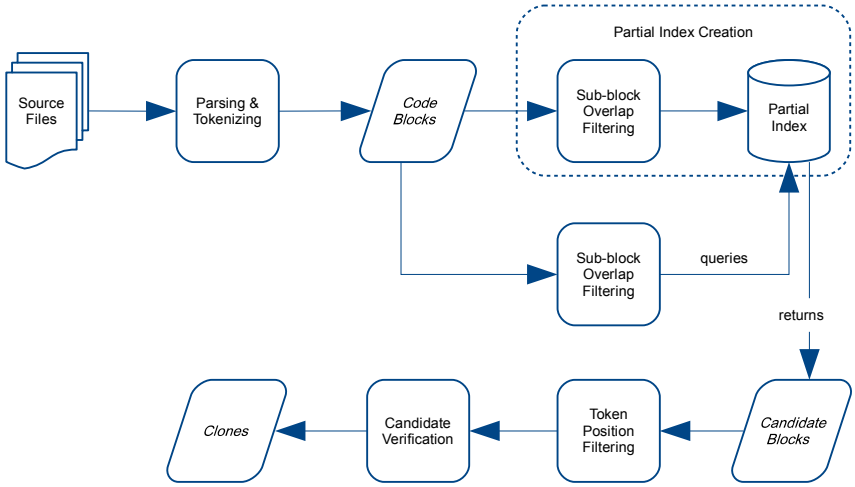

Abbildung 4.1: SourcererCC's Clone Detection-Prozess in Anlehnung an [28]

Abbildung 4.1 stellt den Clone Detection-Prozess von SourcererCC dar. Zuerst verarbeitet der Tokenizer (4.2) die Quellcodedateien und erzeugt die *Code Blocks*. Daraus wird im nächsten Schritt ein *Inverted Index* erzeugt. Einen Inverted Index kann man sich als eine Art Datenbank vorstellen, in der die verschiedenen Tokens gelistet sind. Zudem wird zu jedem Token ein Verweis hinterlegt, wo dieses überall im Quellcode vorkommt (vgl. Kapitel 1, [18]). SourcererCC erzeugt keinen vollständigen Inverted Index, sondern nutzt hierzu eine Filterheuristik (4.1.1). Das verringert die Anzahl potenzieller Clone-Kandidaten und führt zu einem erheblichen Geschwindigkeitsvorteil. Nachdem die *Candidate Blocks* identifiziert und sortiert wurden, kommt eine weitere Filterheuristik (4.1.2) zum Einsatz. Nach abschließender *Candidate Verification* sind mögliche Clones im Quellcode identifiziert. SourcererCC nutzt zur Bestimmung, ob zwei Code Blocks Clones sind, eine *Similarity-Funktion*. Im Folgenden auch Vergleichsfunktion genannt. Diese misst den Grad der Ähnlichkeit zwischen den Code Blocks und gibt einen nicht negativen Wert zurück. Je höher der Wert, desto höher die Ähnlichkeit. Für die Vergleichsfunktion gibt es einen Schwellenwert θ, auch Threshold

genannt. Ist der Wert der Vergleichsfunktion zweier Code Blocks größer als das θ, handelt es sich um Clones. Formal ausgedrückt bedeutet dies:

Seien P_x und P_y zwei Projekte, f eine Similarity-Funktion und θ der Threshold. Das Ziel ist alle Code Blockpaare $P_{x,B}$ und $P_{y,B}$ zu finden, so dass gilt $\mathrm{f}(P_{x,B}, P_{y,B}) \geq \lceil \theta * max(|P_{x,B}|, |P_{y,B}|) \rceil$.

Als Similarity-Funktion nutzen Sajnani et al. *Overlap*. Gegeben seien die zwei Code Blocks B_x und B_y, dann wird die Overlap Similarity $O(B_x, B_y)$ berechnet als die Anzahl an Tokens, die B_x und B_y gemeinsam haben. Alternativ, nach Sajnani et al., können aber auch Jaccard oder Cosine Similarity-Funktionen genutzt werden.

Nachfolgend werden die weiter oben erwähnten Filterheuristiken beschrieben.

4.1.1 Sub-block Overlap Filtering

Die Filterheuristik des Sub-block Overlap Filtering folgt der Intuition: Wenn zwei große Mengen viele Übereinstimmungen aufweisen, so weisen auch ihre kleineren Untermengen Übereinstimmungen auf, so Sajnani et al. [28]. Formal drücken Sajnani et al. das durch die folgende Eigenschaft aus:

Property 1: *Given blocks B_x and B_y consisting of t tokens each in some predefined order, if $|B_x \cap B_y| \geq i$, then the sub-blocks SB_x and SB_y of B_x and B_y respectively, consisting of first $t - i + 1$ tokens, must match at least one token.*

Ein Beispiel zum besseren Verständnis des Sub-block Overlap Filtering ist in Kapitel 6 im Abschnitt 6.1.2 zu finden. Bei der Sortierung der Tokens innerhalb der Code Blocks nutzen Sajnani et al. den Vorteil, dass Programmiersprachen Ähnlichkeiten zu natürlichen Sprachen aufweisen. Es gibt sehr häufig auftretende Wörter, wie z.B. Schlüsselwörter und nicht so häufig auftretende Wörter bzw. Tokens. Durch diese Sortierung können bereits viele *False Positives* während der Clone-Suche eliminiert werden.

4.1.2 Token Position Filtering

Die zweite Filterheuristik ist das Position Filtering. Diese nutzt die Sortierung der Tokens aus, um eine obere und untere Grenze der Similarity-Werte zwischen Query und Candidate Blocks zu bestimmen. Ist die obere Grenze unterhalb von θ, wird der Kandidat verworfen. Akzeptiert wird ein Kandidat, dessen untere Grenze größer als θ ist. Formal ausgedrückt, nach Sajnani et al., ergibt sich daraus die Eigenschaft:

Property 2: *Let blocks B_x and B_y be ordered and $\exists$ token t at index i in B_x, s.t. B_x is divided into two parts, where $B_x(first) =$*

$B_x[1 \ldots (i-1)]$ *and* $B_x(second) = B_x[i \ldots |B_x|]$. *Now if* $|B_x \cap B_y| \geq \lceil \theta * max(|B_x|, |B_y|) \rceil$, *then* $\forall\ t \in B_x \cap B_y, |B_x(first)| \cap |B_y(first)| + min(|B_x(second)|, |By(second)|) \geq \lceil \theta * max(|B_x(first)|, |B_y(first)|) \rceil$.

Ein Beispiel ist der Arbeit von Sajnani et al. in Abschnitt 3.3.2 aus ([28]) zu entnehmen.

Die Quellcodedateien müssen wie eingangs erwähnt in Tokens überführt werden. Das geschieht mit Hilfe des Tokenizers, der im nächsten Abschnitt (4.2) vorgestellt wird.

4.2 Tokenizer

SourcererCC bringt für den zu entwickelnden Security Code Clone Detector keinen für meine Zwecke geeigneten Tokenizer mit. Sajnani et al. stellen ausführbare JAR-Dateien zur Verfügung [26], damit Ihre Experimente mit SourcererCC repliziert werden können. Darin enthalten ist ebenfalls ein rudimentärer Tokenizer, jedoch ohne Quellcode. Im Github Repository[2] sind Tokenizer zu finden, die in der Sprache *Python* verfasst wurden. Hier wären aber so viele Anpassungen und die Portierung zur Programmiersprache Java notwendig, dass ich einen eigenen Tokenizer für meine Adapation des SourcererCC von Grund auf neu entwickelt habe. Hierdurch bieten sich zusätzlich die Möglichkeiten, benötigte Metainformation über Quellcode und CVE-Daten während des Tokenizings zu verarbeiten und zu speichern.

Der Tokenizer benötigt als Eingabe eine einzelne Quellcode-Datei oder eine Pfadangabe. Diese Pfadangabe wird vom Tokenizer rekursiv nach Quellcodedateien durchsucht. Es ist möglich Datei-Patterns in den Tokenizer-Einstellungen anzugeben. Damit wird erreicht, dass beispielsweise nur Java-Dateien (*.java) vom Tokenizer verarbeitet sowie auch weitere Programmiersprachen bzw. deren Quellcodedateien hinzugefügt werden können. Im Abschnitt 5.4.1 wird näher auf die Einstellungsmöglichkeiten des Tokenizers eingegangen.

Beim Tokenizing werden zwei Ausgabedateien erzeugt. Die eine Datei enthält die Tokens der Quellcodedatei. Im Folgenden Tokens-Datei genannt. Die andere Datei enthält die Metadaten, auch *Book Keeping Information* genannt, zur erzeugten Tokens-Datei.

Zum besseren Verständnis des Tokenizers wird seine Funktionsweise nun anhand eines Beispiels vermittelt. In Code-Listing 4.1 ist der Java-Code abgebildet, der in diesem Beispiel vom Tokenizer verarbeitet wird.

[2] `https://github.com/Mondego/SourcererCC`

```java
public class TokenizerExample
{
  public int sumEvenNumbers(int[] numbers)
  {
    int sum = 0;

    for (int n : numbers)
      if (this.isEven(n))
        sum += n;
    return sum;
  }

  private boolean isEven(int n)
  {
    return n % 2 == 0;
  }
}
```

Code-Listing 4.1: Java-Code für Tokenizer-Beispiel

Die aus dem Java-Code (Code-Listing 4.1) erzeugten Tokens, sind in Code-Listing 4.2 abgebildet. Zeile 1 enthält die Tokens für die Methode `sumEvenNumbers`. Zeile 2 entsprechend die Tokens für Methode `isEven`. Ein Token des Schlüsselworts `class` oder der Name der Klasse `TokenizerExample` taucht nicht auf.

```
{0,public,sumEvenNumbers,isEven,numbers,for,this,sum,if,int,n,return}
{0,private,2,boolean,isEven,int,n,return}
```

Code-Listing 4.2: Erzeugte Tokens für 4.1

Der Tokenizer ist so konzipiert, dass er verschiedene Granularitätslevel beim Tokenizing unterstützt. Das hier verwendete Granularitätslevel ist Methoden-basiert. Das heißt, der Tokenizer verarbeitet ausschließlich die Methoden respektive Konstruktoren innerhalb einer Klasse. Andere Daten wie Klassennamen oder *Imports* werden nicht berücksichtigt. Eine Implementation weiterer Granularitätslevel ist im Rahmen dieser Arbeit nicht erfolgt, da sich die möglichen Schwachstellen im Java-Code innerhalb von Methoden befinden. Das Design-Konzept der Programmiersprache Java sieht vor, dass auszuführender Code immer in einer Methode enthalten sein muss. Weitere Granularitätslevel wären beispielsweise Datei-basiert. Der Tokenizer erzeugt hier Tokens der gesamten Datei, also auch Imports, Klassen etc. Darüber hinaus ist ein Statement-basiertes Granularitätslevel vorstellbar, sowie ein Funktions-basiertes. Letzteres bietet sich für Programmiersprachen wie C/C++ an, wo Code auch außerhalb von Klassen und deren Methoden existiert.

```
1 @#@@@::@@1,public@@::@@1,sumEvenNumbers@@::@@1,isEven@@::@@1,numbers@@::@@2,for@@::
      @@1,this@@::@@1,sum@@::@@3,if@@::@@1,int@@::@@4,n@@::@@3,return@@::@@1
2 @#@@@::@@1,private@@::@@1,2@@::@@1,boolean@@::@@1,isEven@@::@@1,int@@::@@1,n@@::@@2,
      return@@::@@1
```

Code-Listing 4.3: Gekürzte Tokens-Datei für 4.1

Nachdem nun in Code-Listing 4.2 die vom Tokenizer erzeugten Tokens, isoliert zur besseren Lesbarkeit, abgebildet sind, enthält Code-Listing 4.3 die gekürzte Tokens-Datei, die während des Tokenizing erzeugt worden ist. Ausgelassen wurden Kopfdaten dieser Datei, ebenfalls zur besseren Lesbarkeit. Sie werden im Anschluss gesondert betrachtet.

Eine Tokens-Datei enthält pro Zeile immer nur die Tokens eines Granularitätslevels. In diesem Fall eine Methode. Zur Trennung erfolgt nach den Kopfdaten - hier ausgelassen - die Zeichenkette `@#@`. Sie ist pro Zeile nur **einmal** zulässig. Jeder *Token Entry* ist durch ein Komma ',' getrennt. Das Ende einer Zeile endet **nicht** mit einem Komma. Ein Token Entry hat die Form:

```
<Token>@@::@@<Quantity>
```

`<Token>` steht hierbei für das entsprechende Token und `<Quantity>` ist eine positive Ganzzahl, die angibt, wie häufig dieses Token auftritt. `@@::@@` dient als Trennzeichen innerhalb eines Token Entries. Beispielsweise tritt das Token `sum` in der Methode `sumEvenNumbers` (4.1) drei Mal auf. Dementsprechend wird der Token Entry `sum@@::@@3` erzeugt.

Wie sehen nun die Kopfdaten einer Tokens-Datei aus und wofür werden diese benötigt? Die Kopfdaten dienen der eindeutigen Identifizierung der Tokens mit einer Methode und den Metadaten. Wie eingangs erwähnt, werden die Metadaten auch synonym als *Book Keeping Information* bezeichnet. Diese werden im Anschluss erläutert. Die Kopfdaten einer Tokens-Datei sind Werte, die ebenfalls durch Kommata getrennt sind und befinden sich am Anfang jeder Zeile in der Datei. Tabelle 4.1 listet die Kopfdaten auf.

Zeilen fünf bis neun sind für zukünftige Erweiterungen und enthalten bislang keine relevanten Daten. Zeile zehn gibt die Gesamtzahl aller Tokens an, die in dieser Zeile enthalten sind. Zeile zwei ist die eindeutige Kennung der Methode. Jede Methode ist darüber eindeutig zu identifizieren. Zeile eins ist die eindeutige Kennung eines Code-Blocks. Beinhaltet eine Methode beispielsweise eine anonyme Klasse, teilen Sie sich eine gemeinsame Block ID. Die Unique Tokenization ID teilen Sie sich jedoch nicht. Die Project ID aus Zeile drei dient dazu, Quellcodedateien eindeutig einem Projekt zuzuordnen. Diese ID ist für die Vermeidung von Intraprojekt-Clones (siehe Abschnitt 4.3) notwendig. Die Unique Resource ID in Zeile vier ist die Kennung der Quellcode-Ressource. Das kann beispielsweise eine Java-Datei oder auch ein in einer Datenbank enthaltenes Code Fragment sein. Weitere Details zu

	Wert	Beschreibung
1.	Block ID	Kennung eines Code-Blocks
2.	Unique Tokenization ID	Kennung einer Methode
3.	Project ID	Kennung eines Projekts
4.	Unique Resource ID	Kennung der Code Ressource
5.	Reserved	Reserviert für zukünftige Verw.
6.	Reserved	Reserviert für zukünftige Verw.
7.	Reserved	Reserviert für zukünftige Verw.
8.	Reserved	Reserviert für zukünftige Verw.
9.	Reserved	Reserviert für zukünftige Verw.
10.	Total Tokens	Gesamtanzahl aller Tokens

Tabelle 4.1: Kopfdaten einer Tokens-Datei

den Kopfdaten können der Datei „TokenInformation.java“ des Java-Projekts „SCCDetectorTokenizer“ entnommen werden.

	Wert	Beschreibung
1.	Block ID	Kennung eines Code-Blocks
2.	Unique Resource ID	Kennung der Code Ressource
3.	Unique Tokenization ID	Kennung einer Methode
4.	Start	Zeilennummer Start
5.	End	Zeilennummer Ende
6.	Granularity Level	siehe Erläuterung
7.	Language	siehe Erläuterung
8.	Resource Type	siehe Erläuterung
9.	Reserved	Reserviert für zukünftige Verw.
10.	Reserved	Reserviert für zukünftige Verw.
11.	Reserved	Reserviert für zukünftige Verw.
12.	RSD	siehe Erläuterung
13.	Resource Name	Name der Ressource.
14.	Name	Name der Methode

Tabelle 4.2: Book Keeping Information

Neben der Tokens-Datei erzeugt der Tokenizer noch die zugehörige Datei mit den Book Keeping Information. Diese speichert für jede Zeile der Tokens-Datei die entsprechenden Metadaten. Die einzelnen Daten einer Zeile in der Book Keeping-Datei sind durch ein Komma ’,’ getrennt. Tabelle 4.2 bildet die Daten ab. Hierbei sind die ersten drei Werte (Zeile eins bis drei) identisch mit den entsprechenden Werten der Tokens-Datei (Block ID, Unique Resource ID und Unique Tokenization ID). Anschließend folgt die Zeilennummer in der Quellcodedatei, in der die verarbeitete Methode anfängt, gefolgt von der Zeilennummer, die das Ende der Methode angibt

(Zeile vier und fünf). Die Zeilen sechs bis sieben sind ganzzahlige Werte. In Zeile sechs ist das Granularitätslevel der erzeugten Tokens hinterlegt. Zeile sieben speichert die Sprache des Quellcodes, von dem die Tokens erzeugt wurden und Zeile acht den Typ der Quellcode-Ressource. Zeile neun bis elf sind für eventuelle spätere Verwendungen reserviert. Zeile 13 enthält den Namen der Quellcode-Ressource, z.B. den Pfad zur Quellcodedatei. Zeile 14 speichert den Methodennamen. Die Abkürzung RSD in Zeile 12 steht für *Resource Security Data* und es sind die in Tabelle 4.3 dargestellten Werte enthalten:

1.	File ID
2.	Vulnerable ID
3.	CVE ID
4.	Typ
5.	Severity Score

Tabelle 4.3: Resource Security Data

Die ersten beiden ID-Angaben in den Zeilen eins und zwei beziehen sich auf die Datenbank, die der Security Code Exporter für Github (4.5) erzeugt. Hierbei verweist die File ID auf die ID Spalte der Tabelle „vulnerability_file“ und die Vulnerable ID auf die ID Spalte der „vulnerability_code“ Tabelle. Mit Hilfe dieser IDs können entsprechende Informationen zu einer Sicherheitslücke bei Bedarf dynamisch geladen werden. Zeile drei enthält die CVE ID der Sicherheitslücke. Der Typ in Zeile vier kann die Werte „UNKNOWN“, „VULNERABLE_CODE“ oder „EXPLOIT_CODE“ annehmen. Das sind Daten, die ebenfalls der Security Code Exporter für Github erzeugt. Zeile fünf hinterlegt den Schweregrad für die entsprechende Sicherheitslücke. Sollten zu einem Code Fragment im Security Repository keine weiteren Daten vorliegen, sind die Resource Security Data-Informationen leer. Weitere Details sind hier den Quellcodedateien „BookKeepingInformation.java“ des Java-Projekts „SCCDetectorTokenizer“ und „ResourceSecurityData.java“ des Java-Projekts „SCCDetectorAPI“ zu entnehmen.

Der nun nachfolgende Abschnitt (4.3) erörtert die notwendigen Anpassungen der Clone Detector-Komponente.

4.3 Adaption von SourcererCC

Neben der Neuentwicklung eines Tokenizers waren einige Anpassungen und Änderungen notwendig, damit SourcererCC als Clone Detector-Komponente für den Security Code Clone Detector verwendet werden konnte. Die von mir als Grundlage verwendete Codebasis des SourcererCC war nicht in der Lage *Intraprojekt-Clones* zu identifizieren und von der Liste der Clone Candidates zu streichen. Intraprojekt-Clones sind Code Clones, die innerhalb eines Java-

Projekts auftreten, wie zum Beispiel redundanter Code. Das Ziel des Security Code Clone Detectors ist es aber, Security Code Clones in einem Java-Projekt bzw. einer Quellcodedatei zu identifizieren anhand eines Security Repositories. Abbildung 4.2 soll das verdeutlichen. Sei A die Menge von

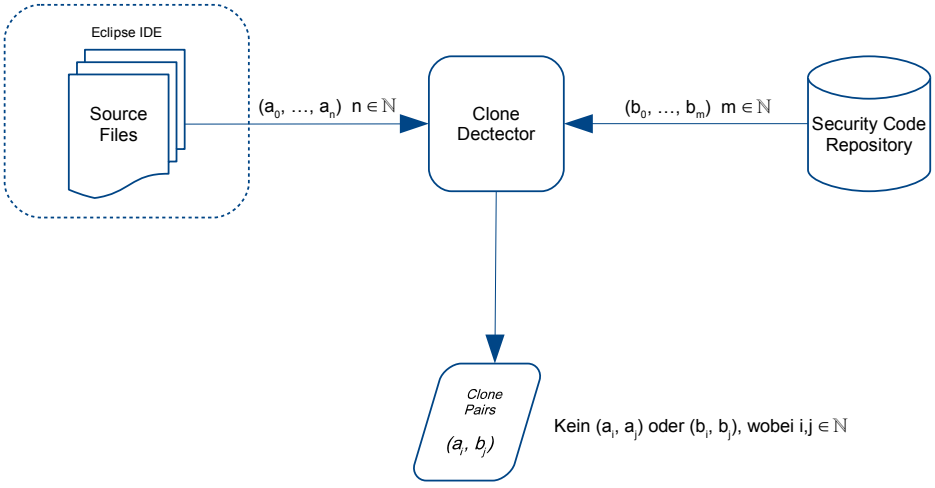

Abbildung 4.2: Vermeidung von Intraprojekt-Clones

Quellcode mit der ein Entwickler innerhalb von Eclipse arbeitet und B die Menge von Quellcode innerhalb des Security Code Repositories. Weiter seien $(a_0, \ldots, a_n)$ und $(b_0, \ldots, b_m)$ mit $n, m \in \mathbb{N}$ Code Fragmente, wobei $a_i \in A$ und $b_j \in B$ mit $i, j \in \mathbb{N}$ gilt. Der Clone Detector prüft nun die Code Fragmente a_i und b_j auf Security Code Clones. Sollten Clones gefunden werden, werden diese der Menge der Clone Pairs hinzugefügt. Ein Intraprojekt-Clone Pair hätte die Form (a_i, a_j). Wie eingangs schon erwähnt, soll aber nicht nach dieser Art der Clones gesucht werden, sondern bei dieser Aufgabenstellung sind nur Clone Pairs aus Eclipse-Quellcode a_i und Security Repository-Quellcode b_j von Interesse, da wir nach bekannten Schwachstellen suchen. Formal also an einem Clone Pair der Form (a_i, b_j). Dazu wurde die *Candidate Verification*-Phase (siehe Abbildung 4.1) so angepasst, das hier mögliche Intraprojekt-Clones erkannt und eliminiert werden.

Zudem habe ich SourcererCC an das eigens entwickelte Tokens-Format adaptiert mit entsprechenden Änderungen aller nötigen Datenstrukturen. Darüber hinaus sind die folgenden Weiterentwicklungen von mir vorgenommen worden: Anpassen und Hinzufügen von Einstellungsmöglichkeiten (siehe Abschnitt 5.4.1), Unterstützung für Datei-Patterns sowohl bei Tokens- als auch Quellcodedateien, verschiedene Phasen des Clone Detection-Prozesses mit Hilfe eines *Controllers* gebündelt und die Verarbeitung der Book Keeping Informationen hinzugefügt.

Nachdem das Konzept des Clone Detectors sowie des Tokenizers disku-

tiert wurden, erfolgt nun die Vorstellung des Security Code Clone Detectors.

4.4 Security Code Clone Detector

Der Security Code Clone Detector setzt sich aus drei Komponenten zusammen: Der Benutzeroberfläche, dem Tokenizer und dem Clone Detector. Die Suche nach Schwachstellen läuft hierbei vollautomatisch im Hintergrund der Eclipse-IDE ab. Die Arbeitsweise des Security Code Clone Detectors ist in Abbildung 4.3 veranschaulicht.

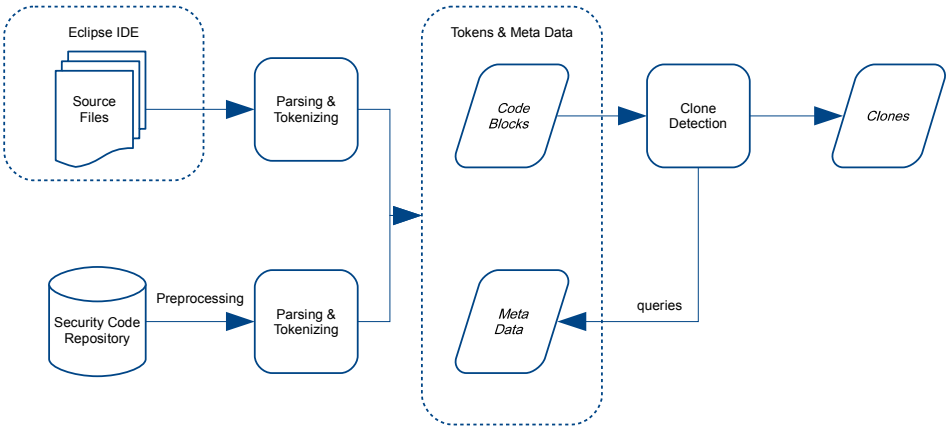

Abbildung 4.3: Arbeitsweise des Security Code Clone Detectors

Wie in Abschnitt 4.3 besprochen soll der Security Code Clone Detector bekannte Schwachstellen innerhalb des in der Eclipse-IDE geschriebenen Java-Codes finden. Dazu wird Quellcode benötigt, der bekannte Schwachstellen enthält. Dieser ist im Security Code Repository (4.5) hinterlegt und wird vom Tokenizer verarbeitet. Hierbei werden die benötigten Tokens erzeugt und die Metadaten mit Informationen zu den einzelnen Schwachstellen hinterlegt. Schreibt der Entwickler nun Code innerhalb der Eclipse-IDE, wird dieser ebenfalls vom Tokenizer verarbeitet. Der nächste Schritt ist in Abbildung 4.3 als *Clone Detection* gekennzeichnet. Dabei wird von den Code Blocks aus Security Repository und Eclipse-IDE ein gemeinsamer Index erzeugt, auf dem anschließend die Clone-Suche erfolgt. Zu den gefundenen Clone Pairs werden die Metadaten angefordert und das Ergebnis der Clone-Suche, hier als *Clones* dargestellt, ausgegeben.

Die Benutzeroberfläche für den Security Code Clone Detector ist so konzipiert, dass der Benutzer wählen kann, ob eine Clone-Suche automatisch oder erst durch aktives Handeln des Benutzers erfolgen soll. Zudem lässt sich der Security Code Clone Detector an die jeweiligen Bedürfnisse des Benutzers anpassen (5.3.5). Folglich sind die Anforderungen [R2.1], [R2.2], [R2.4] und

[R7] erfüllt.

Ein großer Vorteil des Security Code Clone Detectors ist, dass er den Entwickler von Anfang an des Entwicklungsprozesses bei der Suche nach Schwachstellen unterstützt. Dafür braucht der Java-Code sogar noch nicht kompilier- und/oder lauffähig sein. Das heißt auch Teilimplementationen von Klassen und/oder Interfaces sowie Code Fragmente mit noch fehlenden Datenstrukturen können bereits auf Sicherheitslücken geprüft werden. Der Java-Code muss lediglich syntaktisch korrekt formuliert sein. Das wird dadurch erreicht, dass der Tokenizer zum Erzeugen der Tokens den Java-Code nur parsed und nicht weitergehend analysiert bzw. kompiliert.

Die Frage, woher die Daten für das Security Code Repository kommen und was sich dahinter verbirgt, soll nun im folgenden Abschnitt 4.5 geklärt werden.

4.5 Security Code Exporter für Github

Damit der Security Code Clone Detector Java-Code auf Schwachstellen prüfen kann, muss er „wissen“, was eine Schwachstelle ist oder präziser formuliert, benötigt er Quellcodedaten zu einer (bekannten) Schwachstelle. Dafür wird das Security Code Repository benötigt. Dort liegen alle Quellcodedateien, die bekannte Schwachstellen enthalten. Somit sind die Anforderungen [R2.3] „Bestehendes Wissen vorhandener (Code-) Datenbanken nutzen“ und [NR5] „Ein vorhandenes Security Repository soll genutzt werden“ ebenfalls erfüllt. Security Code Repository und Security Repository werden in dieser Arbeit synonym verwendet.

Woher kommt aber nun der Quellcode mit den Sicherheitslücken? Hierfür wurde im Rahmen dieser Masterarbeit der Security Code Exporter für Github genutzt. Dieser ist innerhalb der Bachelorarbeit von Christian Müller [20] entwickelt worden. „Der Security Code Exporter sucht automatisiert auf Github, einem der größten Hosts für Softwareprojekte, nach security relevantem Code und lädt diesen herunter“, sagt Müller. „Die gefundenen Ergebnisse, bestehend aus Metadaten und Dateien, werden in einer Datenbank gespeichert und können angezeigt und durchsucht werden“, so Müller weiter. Diese Daten, bestehend aus Quellcode und Informationen über die Schwachstelle, werden vom Security Code Clone Detector zur Schwachstellensuche genutzt.

Nachdem nun das Konzept des Security Code Clone Detectors besprochen wurde, wird im nächsten Kapitel (5) dessen Implementation vorgestellt.

Kapitel 5

Implementation

Zu Beginn dieses Kapitels wird in Abschnitt 5.1 die Architektur des Security Code Clone Detectors vorgestellt. Im Anschluss daran wird in Abschnitt 5.2 das Application Programming Interface (API) näher beleuchtet. Es folgt die Betrachtung der Benutzeroberfläche (GUI) des entwickelten Eclipse Plugins in Abschnitt 5.3. Zum Ende dieses Kapitels werden die Einstellungsmöglichkeiten für das Plugin (5.3.5) und den Clone Detector (5.4.1) vorgestellt. Diese sind bewusst im Kapitel „Implementation“ untergebracht, da hier gleichzeitig auf Implementationsdetails eingegangen wird. Abschließend wird in Abschnitt 5.5 die Implementation des Tokenizers erörtert.

5.1 Architektur

Die Architektur des Security Code Clone Detectors besteht aus drei Schichten und dessen Implementation ist am Model View Controller (MVC) Architekturmuster angelehnt. Hierbei steuert ein Controller den Programmablauf. Er kommuniziert über eine API (5.2) mit den einzelnen Schichten und steuert den Clone Detector sowie den Tokenizer. Zudem ist er dafür zuständig, dass die einzelnen Sichten (Views) auf UI-Ebene aktualisiert werden. Das Model besteht hierbei aus dem Security Code Repository und den Quellcodedaten innerhalb der Eclipse-UI. Es wird vom Controller nicht eigenständig verändert, da der Quellcode vom Entwickler geschrieben und das Security Code Repository durch den Security Code Exporter (4.5) verändert wird.

Wie sehen nun die einzelnen Schichten des Security Code Clone Detectors aus? Abbildung 5.1 stellt diese dar. Die oberste Schicht besteht aus der GUI, dem *Eclipse Plugin*. Es folgt die *Clone Detector API*-Schicht und die unterste Schicht ist die des *Clone Detectors* samt *Tokenizer*. Jede dieser Schichten ist als separates Java-Projekt implementiert, wobei Clone Detector und Tokenizer ebenfalls zwei getrennte Module sind, wie im Schichtenmodell zu sehen. Daraus ergeben sich insgesamt vier verschiedene Java-Projekte

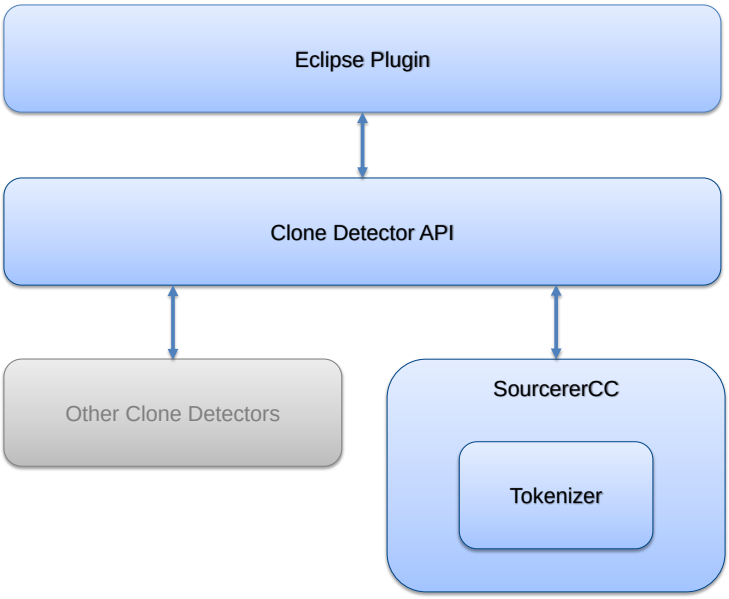

Abbildung 5.1: Schichten des Security Code Clone Detectors

bzw. Module. Das Plugin befindet sich im Projekt „SCCDetectorPlugin“. Die API-Schicht ist im Projekt „SCCDetectorAPI“ implementiert, der Tokenizer im „SCCDetectorTokenizer“-Projekt und das Clone Detector-Projekt heißt „SCCDetector-v2-MA“. Damit ist die Anforderung [R5.1] „Modularer Aufbau der Software“ erfüllt.

Der nun folgende Abschnitt beleuchtet das Application Programming Interface (API) näher.

5.2 Clone Detector API

Das Plugin, genauer gesagt der Controller kommuniziert über eine von mir entwickelte API mit dem Clone Detector, so dass die Anforderung [NR2] „Die Software soll erweiterbar sein“ ebenfalls dahingehend erfüllt ist, dass andere Clone Detection-Ansätze darüber eingebunden werden können, ohne dass Änderungen am Plugin vorgenommen werden müssen. Zukünftige Clone Detection-Ansätze brauchen dafür nur die entsprechenden Interfaces implementieren. Abbildung 5.2 zeigt zunächst das UML-Paketdiagramm der Clone Detector API. Dieses besteht aus vier Paketen (Packages). Im Package *de.luh.se.sccd.api* befinden sich die Interfaces, die zur Kommunikation mit dem Clone Detector genutzt werden. Das Package *de.luh.se.sccd.util* beinhaltet die Klasse `Tuple`. Der Datenbank-Controller mit Klassennamen `SecurityRepositoryController` befindet sich im Package

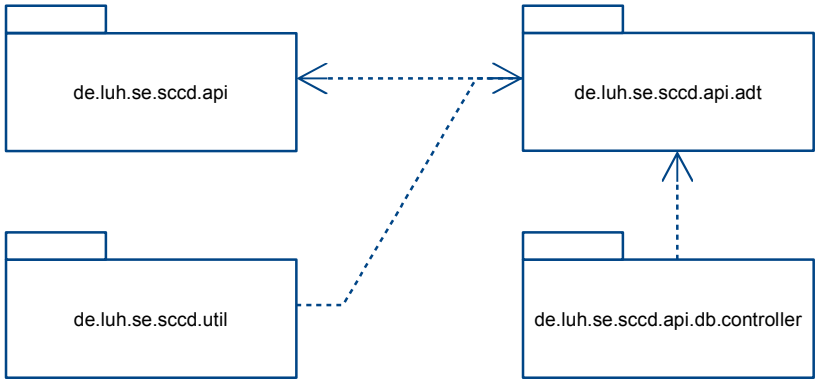

Abbildung 5.2: UML-Paketdiagramm der API

de.luh.se.sccd.api.db.controller. Dieser liest die Metadaten aus der CVE- und Repository-Datenbank aus, die vom Security Code Exporter erzeugt werden. Weiterführende Informationen zu den Datenbankschemata sind der Arbeit [20] zu entnehmen.

Die Klassen mit den verschiedenen API-Datentypen sind im Package *de.luh.se.sccd.api.adt* enthalten. Ein UML-Klassendiagramm ist im Anhang in Abbildung A.2 dargestellt. Für Informationen zu den einzelnen Klassen und ihrer Bedeutung sei hier auf den Quellcode verwiesen. Hier habe ich mich gemäß [R5] „Gut strukturierter und kommentierter Programmcode“ bemüht, ausreichend Informationen zu den einzelnen Klassen und deren Anwendungen in Form von Kommentaren zu geben. Darüber hinaus habe ich den Code so strukturiert, dass sich dieser gut nachvollziehen lässt. Das Paket `de.luh.se.sccd.api` mit seinen Interfaces soll nun etwas eingehender betrachtet werden. Dazu ist in Abbildung 5.3 eine UML-ähnliche Darstellung der darin enthaltenen Interfaces dargestellt. Zwecks Übersichtlichkeit und besserer Lesbarkeit sind keine Rückgabetypen (außer `void`) und Parameter angegeben. Ein vollständiges UML-Diagramm der Interfaces ist der Abbildung A.1 im Anhang zu entnehmen.

```
1 public class OtherCloneDetector implements ICloneDetector
2 {
3   public void tokenize() { /* empty */ }
4
5     public void index() { /* empty */ }
6 }
```

Code-Listing 5.1: Beispiel leerer Implementation Body für ICloneDetector

Soll zukünftig ein zusätzlicher bzw. anderer Clone Detector genutzt werden,

Abbildung 5.3: UML-ähnliches Klassendiagramm, Package de.luh.se.sccd.api

so muss dieser das Interface `ICloneDetector` implementieren. Wenn dieser Clone Detector nicht Token-basiert ist oder keinen Indexierungsprozess verwendet, können die Methoden `tokenize` und `index` mit leerem Implementation Body implementiert werden, wie Code-Listing 5.1 zeigt.

Die Methode `setProperties` übergibt einer Clone Detector-Instanz die detektorspezifischen Einstellungen. Die müssen das Interface `ICloneDetectorProperties` implementieren. Die Konfiguration eines Clone Detectors erfolgt mit Hilfe des `ICloneDetectorConfiguration`-Interfaces. Dafür sind die Methoden `loadProperties`, `restoreDefaultProperties` `getPropertiesFileName` zu implementiern. Letztere darf nur den Dateinamen ohne zugehörigen Pfad zurückgeben. Die Methode `initializeCloneDetector` ist dafür gedacht, falls ein Clone Detector vor seiner ersten Verwendung eine Initialisierung benötigt, wie zum Beispiel Laden zusätzlicher Daten oder Instanziierung von Datenstrukturen. Des Weiteren erfolgt über das `ICloneDetectorConfiguration`-Interface auch das Preprocessing des Security Repositories. Dieser Vorgang wird über die `preProcessSecurityRepository`-Methode initiiert und abschließend wird die Methode `postProcessSecurityRepository` aufgerufen. Die Interfaces `IBookKeepingInformation` und `IVulnerabiltyDetails` werden von den Datenstrukturen `BookKeepingInformation` `VulnerabiltyDetailsBase` aus dem Package `de.luh.se.sccd.api.adt` implementiert. Sollten Datenstrukturen benötigt werden, die zusätzliche Book Keeping Information speichern oder zusätzliche Informationen zu Sicherheitslücken benötigen, müssen

diese direkt von den Klassen `BookKeepingInformation` respektive `VulnerabiltyDetailsBase` erben.

Mit Hilfe der Clone Detector API-Schicht ist die Anforderung [NR2.1] „Durch eine einheitliche API sollen andere Clone Detection-Ansätze integriert werden können“ erfüllt. Der nun folgende Abschnitt beschreibt die Implementation des Eclipse Plugins.

5.3 GUI - Eclipse Plugin

Damit die Anforderungen [R1] „Die Software soll als Eclipse Plugin realisiert werden“ und [R1.1] „Das Plugin soll in der Sprache Java geschrieben werden“ erfüllt sind, wurde das Eclipse Plugin für den Security Code Clone Detector mit der Eclipse Version Oxygen.3 Release (4.7.3) unter Verwendung von Java in der Version 1.8.0_171 entwickelt. Das Plugin wurde auf Basis der neuen Eclipse 4 Application Platform[1] (e4 Application) implementiert. Dabei habe ich mich an den Eclipse User Interface Guidelines in der Version 2.1 [8] orientiert.

Das Plugin und seine Implementation wird nun schrittweise in den nachfolgenden Abschnitten vorgestellt. Hierfür wird als erstes in Abschnitt 5.3.1 auf das Hauptmenü des Plugins eingegangen. Anschließend wird die Implementation für die Suche nach Security Code Clones in Abschnitt 5.3.2 erklärt. Danach folgt ein Überblick über die verschiedenen Sichten des Plugins in Abschnitt 5.3.3. Abschließend stellt der Abschnitt 5.3.4 weitere Methoden vor, wie das Security Code Clone Detector Plugin dem Benutzer Schwachstellen visualisiert.

5.3.1 SCC Detector Menü

Das Menü des Security Code Clone Detectors ist in Abbildung 5.4 dargestellt und wird in der Eclipse-IDE in der Hauptmenüzeile angezeigt. Darüber hat der Benutzer die Möglichkeit den Security Code Clone Detector zu steuern. Der Benutzer kann so ein ganzes Java-Projekt nach Clones durchsuchen lassen, die automatische Clone-Suche aktivieren bzw. deaktivieren, eine Suche nach Clones in der aktiven Datei manuell starten. Ebenso hat der Benutzer über das Menü Zugriff auf die Clone Detector-Einstellungen. Das Preprocessing des Security Code Repositories erfolgt ebenfalls über das Menü.

Wie erfolgt nun eine Suche nach Security Code Clones? Diese kann entweder manuell erfolgen oder vollautomatisch, wie der jetzt folgende Abschnitt 5.3.2 erläutert.

[1] `https://eclipsesource.com/de/technologie/e4/`

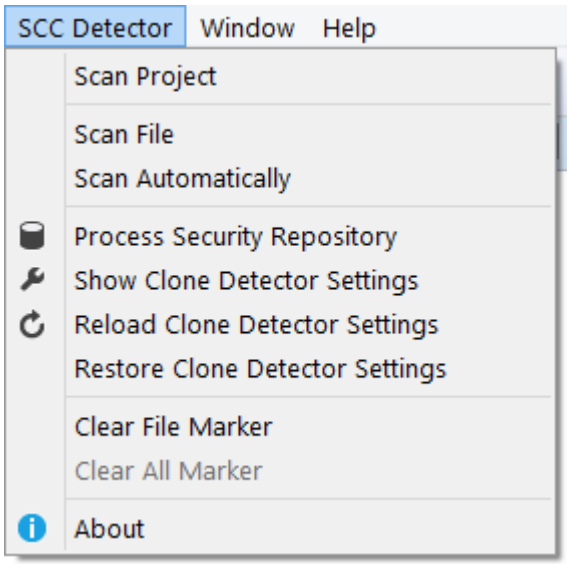

Abbildung 5.4: Security Code Clone Detector Menü

5.3.2 Suche nach Security Code Clones

Wie das Menü in Abbildung 5.4 zeigt, gibt es hier die Möglichkeit eine manuelle Suche nach Security Code Clones zu starten. Zum einen kann diese für ein derzeit aktives Java-Projekt erfolgen, zum anderen für die aktuelle Quellcodedatei, mit der der Entwickler gerade arbeitet. Ist der Haken im Menüpunkt „Scan Automatically“ gesetzt, wird die aktuelle Quellcodedatei vollautomatisch nach Schwachstellen überprüft. Damit Anforderung [NR1] „Die Clone-Suche darf die Eclipse-UI nicht blockieren“ erfüllt ist, kommt die Eclipse Jobs-API hier zum Einsatz. Diese ermöglicht es zeitintensive Prozesse, wie in diesem Fall die Suche nach Security Code Clones, in einen Hintergrundprozess auszulagern. Dieses hat den Vorteil, dass die Eclipse-UI nicht blockiert und der Benutzer weiterhin ungehindert arbeiten kann. Gleichzeitig wird der Fortschritt dieses Hintergrundprozesses dem Benutzer angezeigt.

Die vollautomatische Suche nach Security Code Clones wurde entsprechend Anforderung [NR3] „Die Clone-Suche soll auf Änderungen im Programmcode reagieren“ wie folgt umgesetzt. Wenn der Entwickler Änderungen an einer Quellcodedatei vornimmt und anschließend speichert, wird das Security Code Clone Detector Plugin durch Implementation eines Resource Change Listeners vom Eclipse Framework über die Änderungen informiert. Nun aber sofort eine neue Clone-Suche zu initiieren würde möglicherweise einen Overhead erzeugen, da der Entwickler vielleicht nur zwei Zeichen im Code geändert hat. An dieser Stelle habe ich die Prüfung einer unteren Grenze für Änderungen an Dateien in Form von veränderten

Bytes in Bezug auf die Dateigröße implementiert. Praktisch bedeutet dies, dass mindestens n Bytes an einer Datei verändert werden müssen, bevor eine neue Suche nach Security Code Clones gestartet wird. Die Anzahl der Bytes-Änderung ist frei einstellbar (siehe 5.3.5). Was ist aber mit Entwicklern, die nur sehr selten speichern? Angenommen ein Entwickler hat die Angewohnheit, nur alle 1000 bis 2000 Zeichen seinen Quellcode zu speichern. Das könnte dazu führen, dass sehr viele Schwachstellen bei einer Clone-Suche gefunden werden und der Entwickler somit jetzt wieder sehr viel Code ändern muss. Besser wäre es, den Entwickler frühzeitiger über mögliche Schwachstellen zu informieren. Auch hier bietet der Security Code Clone Detector eine Lösung an. Es ist möglich, die automatische Suche nach Clones nicht nur nach einem Save Event an einer Quellcodedatei zu starten, sondern das Plugin bietet hier alternativ die Möglichkeit, nach einem benutzerdefinierten Zeitintervall eine Clone-Suche zu starten. Realisiert habe ich das über einen periodisch laufenden Hintergrundprozess, der alle n Minuten schaut, ob genügend Bytes der Quellcodedatei geändert wurden, um eine neue Suche nach Security Code Clones zu starten. Das Zeitintervall ist frei einstellbar.

Die vollautomatische Suche nach Clones erfolgt immer für die aktuelle Quellcodedatei, mit der ein Entwickler gerade arbeitet. Da es im Grunde genommen auszuschließen ist, dass ein Entwickler sämtliche Quellcodedateien eines ganzen Java-Projekts auf einmal ändert, wird die projektbezogene Suche nach Schwachstellen aktiv durch den Benutzer initiiert. Das erfolgt über den entsprechenden Menüpunkt in Abbildung 5.4.

Der nächste Abschnitt (5.3.3) befasst sich mit der Darstellung der Ergebnisse einer Clone-Suche.

5.3.3 Views

Der eingangs in Abschnitt 5.1 erwähnte Controller startet eine Clone-Suche auf Befehl des Benutzers oder automatisiert über ein Save Event respektive periodisch laufenden Hintergrundprozess. Anschließend ist er dafür zuständig die Sichten, auch Views genannt, zu aktualisieren und somit die Ergebnisse der Suche dem Benutzer zu visualisieren.

Der Security Code Clone Detector verfügt über zwei eigene Views. Diese sind entsprechend Eclipse User Interface Guidelines 7.5 ([8]), über das Menü „Window → Show View → Other“ zu finden. Daraufhin erscheint ein Dialog mit den verfügbaren Views (siehe Abbildung 5.5). Die Views sind in der Kategorie „SCC Detector“ untergebracht. Der Security Code Clone Detector verfügt zusätzlich über eine eigene Konsole, die in die Eclipse-eigene Console View integriert ist. Implementiert ist die Konsole als `MessageConsole` über das Eclipse Interface `IConsoleManager`. Die Konsole wird zur Ausgabe von Statusinformationen und möglichen Fehlern genutzt. Abbildung 5.6 zeigt die Konsolenausgabe nach einer Schwachstellensuche.

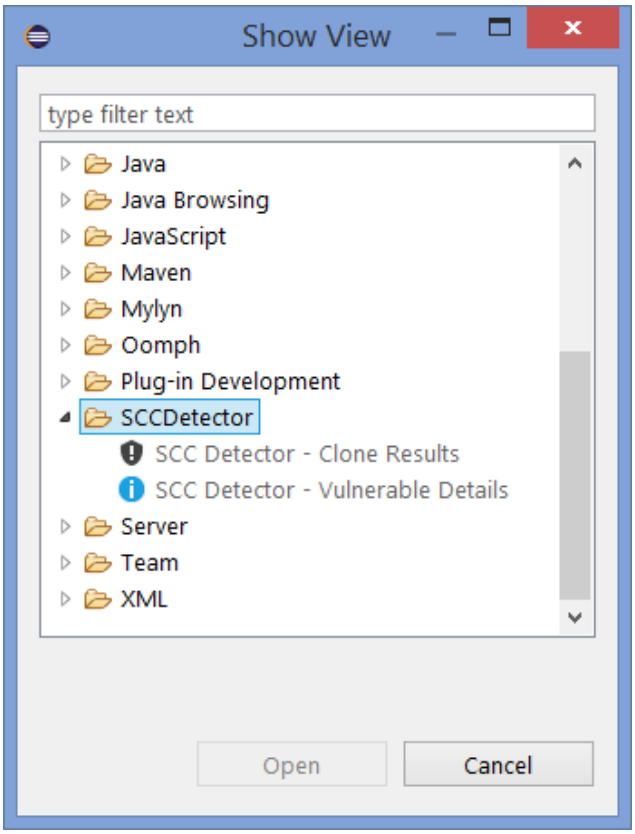

Abbildung 5.5: Security Code Clone Detector Views

Der „Clone Results“-View (Abbildung 5.7) bereitet die Suchergebnisse in einer Listenansicht auf. Hierbei gibt die erste Spalte den Schweregrad einer Sicherheitslücke an, bestehend aus Symbol und Scoring. Die zweite Spalte enthält die CVE, sofern vorhanden. Der dritten Spalte ist der Name der Quellcodedatei zu entnehmen, die die Sicherheitslücke enthält. Gefolgt von der Spalte „Method“, die den Methodennamen angibt und die letzte Spalte gibt die Zeilennummern der Methode in der Form `<Start>-<Ende>` an. Der „Clone Results“-View verfügt zudem über ein Kontextmenü, das in Abbildung 5.8 dargestellt ist. Über den Befehl „Goto Source“ gelangt man direkt zur Methode im Quellcode der aktuell selektierten Schwachstelle. Dieses kann ebenfalls über einen Doppelklick auf die Selektion erfolgen. „More Details“ öffnet den „Vulnerable Details“-View. „Clear All“ löscht die gesamte Liste. Darüber hinaus lässt sich die Ergebnisliste nach Severity Scoring oder CVE sortieren, jeweils auf- bzw. absteigend. Die verwendeten Icons zu den entsprechenden Schweregraden einer Schwachstelle mit samt Scoring-Intervallen sind Tabelle 5.1 zu entnehmen. Der „Vulnerable Details“-View ist in Abbildung 5.9 dargestellt. Hier werden Detailinformationen zu

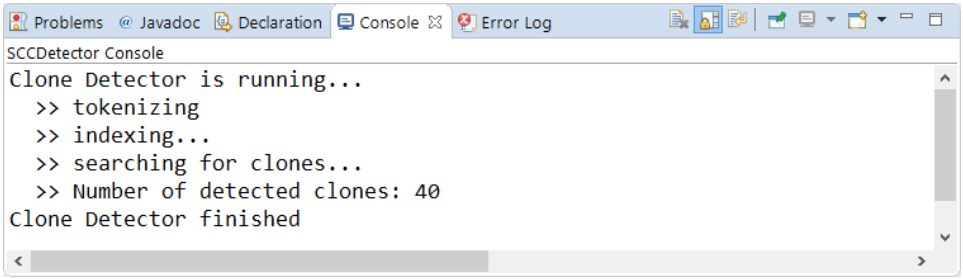

Abbildung 5.6: Security Code Clone Detector Console Output

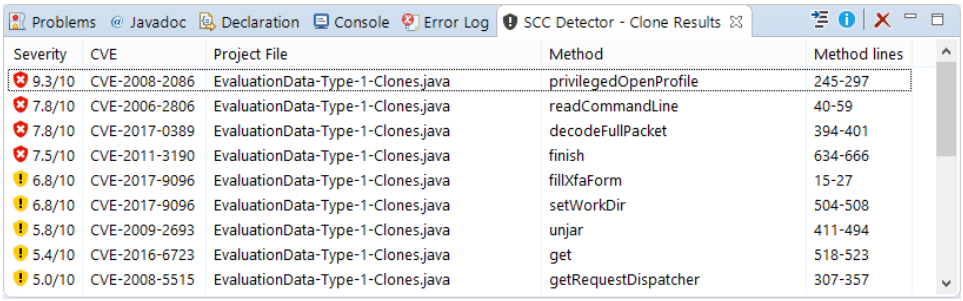

Abbildung 5.7: Clone Results View

einer aus der Clone Results View gewählten Sicherheitslücke visualisiert. Neben dem Severity Icon ist die CVE als Link realisiert. Ein Klick darauf führt zur *National Vulnerability Database*[2] (NVD) des *National Institute of Standards and Technology* (NIST). Hier kann der Benutzer eine Vielzahl von Details zur Schwachstelle einsehen wie z.B. Auswirkungen, betroffene Produkte usw.

Im View ist eine Zusammenfassung zur Schwachstelle zu sehen. Diese ist direkt mit der entsprechenden CVE verknüpft und stammt aus der

[2] https://nvd.nist.gov

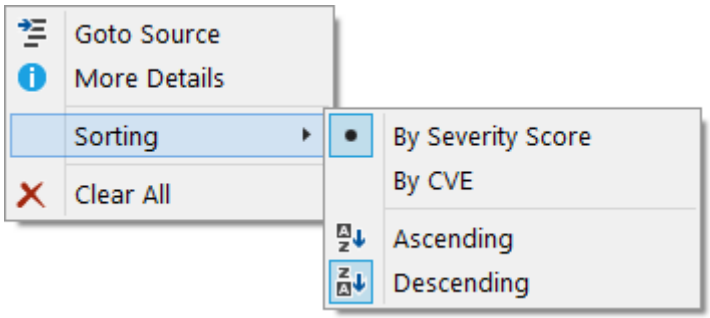

Abbildung 5.8: Clone Results Context Menu

Icon	Severity	Base Score Range
	Low	0,0 - 3,9
	Medium	4,0 - 6,9
	High	7,0 - 10,0

Tabelle 5.1: Severity Score Icons

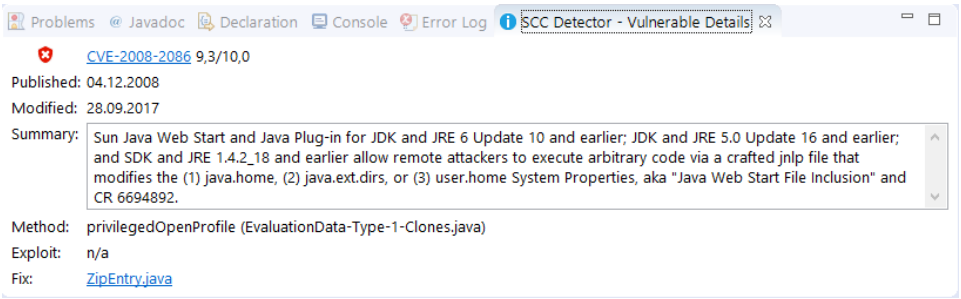

Abbildung 5.9: Vulnerable Details View

Datenbank des Security Code Repository. Sollte zu einer Schwachstelle ein Exploit oder Fix im Security Code Repository hinterlegt sein, sind diese per Klick auf den entsprechenden Link einzusehen. In der Abbildung 5.9 ist beispielsweise ein Link zu einem Fix in der Datei „ZinEntry.java“ angegeben.

Neben den gerade vorgestellten Sichten, verfügt das Security Code Clone Detector Plugin noch über weitere Optionen, dem Benutzer die Schwachstellen zu visualisieren. Diese werden im folgenden Abschnitt 5.3.4 vorgestellt.

5.3.4 Warnungen im Project Explorer und Code Editor

Das Plugin ist so konzipiert, dass es den Entwickler möglichst nicht von seiner Tätigkeit ablenkt. Dazu gehört auch, dass nicht davon ausgegangen werden kann, dass der Entwickler nach jeder Schwachstellensuche gewissenhaft die Ergebnisliste studiert. Das würde bedeuten, dass er jedes Mal das Schreiben von Code unterbrechen müsste. Darüber hinaus soll gemäß Anforderung [NR8] ein Security Code Clone farblich hervorgehoben werden. Somit braucht der Entwickler seine Tätigkeit nicht zu unterbrechen und sieht sofort innerhalb des Eclipse-eigenen Java Editor, dass eine Schwachstelle gefunden wurde und vor allem auch wo. Mit Hilfe der Implementation von Datei-Markern wird die entsprechende Stelle im Code farblich hervorgehoben, wie in Abbilung 5.10 zu sehen ist. Die farbliche Hervorhebung unterscheidet sich hier je nach Schweregrad (Severity Level) der gefundenen Schwachstelle. Für das Severity Level *Low* ist der Quellcode hellgrau hinterlegt, *Medium* ist hellgelb und *High* hellrot hinterlegt. Am linken Rand des Code Editors wird zusätzlich das dem Schweregrad entsprechende Icon eingeblendet.

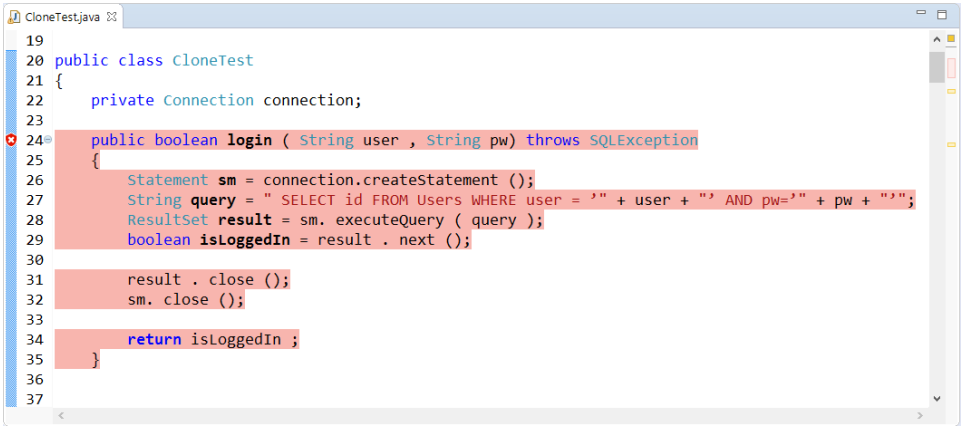

Abbildung 5.10: Code Highlighting für Schwachstellen

Zusätzlich werden die entsprechenden Quellcodedateien im *Project Explorer* von Eclipse annotiert. Dargestellt ist das in Abbildung 5.11. Die Datei „CloneTest.java“ hat die nachgestellte Annotation *[Vuln. found]* und ein kleines *Overlay Icon* rechts oben. „Vuln. found“ steht für Vulnerable found und wurde abgekürzt, damit die gesamte Annotation auch bei kleinerem *Project Explorer*-Fenster sichtbar ist. In der Abbildung zwecks besserer Lesbarkeit mit einem roten Rahmen versehen. Die Realisierung erfolgte über einen *Resource Decorator* unter Verwendung des `ILightweightLabelDecorator`-Interfaces.

Durch die Implementationen der Views in Abschnitt 5.3.3 und der in Abschnitt 5.3.4 vorgestellten Warnungen im Package Explorer sowie der farblichen Hervorhebung der Schwachstellen im Code Editor (siehe auch Abb. 5.10) sind die Anforderungen [R3] „Die Software meldet dem Benutzer, wenn Clones gefunden wurden“ und [NR8] „Code Clones werden direkt in der UI farblich hervorgehoben“ umgesetzt worden. Darüber hinaus ist sogar die als **optional** gekennzeichnete Anforderung [NR7] „Optional: Der Benutzer erhält per Mausklick Detailinformationen zu Schwachstellen“ durch die „Vulnerable Details“-View (Abb. 5.9) erfüllt.

Als nächstes folgt ein Einblick in Einstellungen des Plugins.

5.3.5 Einstellungen Plugin

Die Einstellungen für das Plugin wurden als *Eclipse Preference Pages* implementiert und sind über „Window → Preferences → General → SCC Detector“ zu erreichen. Der Einstellungsdialog ist im Anhang in Abbildung A.7 dargestellt. Eine kurze Erläuterung dazu erfolgt ebenfalls im Anhang.

Die einzelnen Farben für die *Marker* zur Hervorhebung der Schwachstellen im Code Editor sind ebenfalls anpassbar. Erreichbar unter „Window → Preferences → General → Editors → Text Editors → Annotations“. Siehe

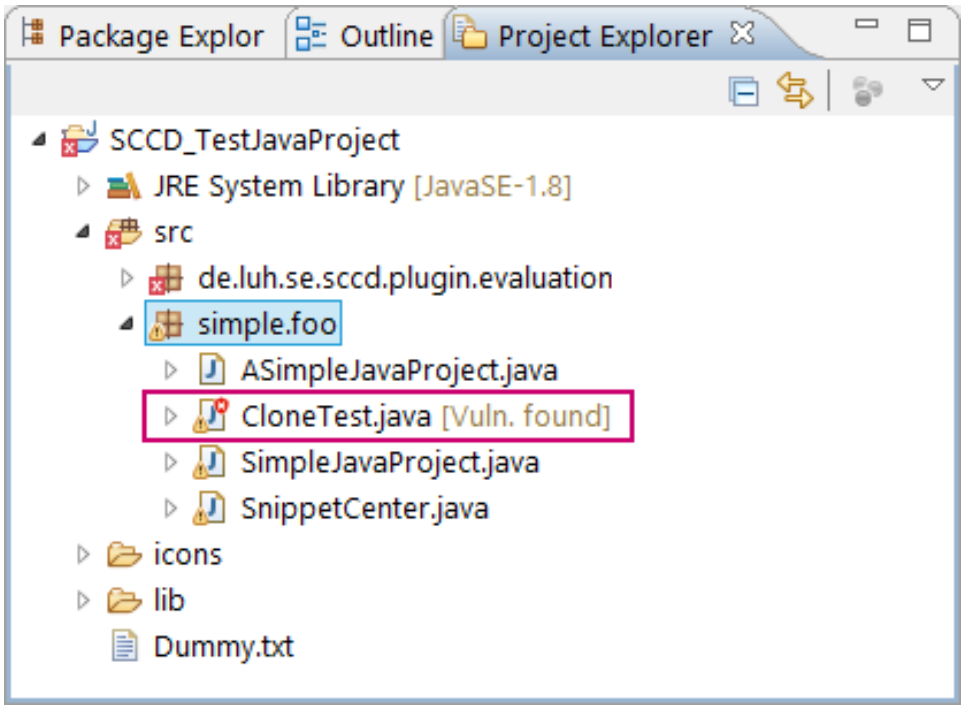

Abbildung 5.11: Warnungen im Project Explorer

Abbildung A.8 im Anhang dazu. Die Auswahl der Marker-Farben erfolgt nicht über den Einstellungsdialog (A.7), weil Eclipse die Annotationen für den Text Editor an zentraler Stelle verwaltet.

Nun folgt abschließend noch die Erweiterbarkeit des Plugins in Abschnitt 5.3.6.

5.3.6 Plugin Erweiterbarkeit

Im Abschnitt 5.1 ist bereits der modulare Aufbau und die Erweiterbarkeit des Security Code Clone Detectors diskutiert worden. Das Plugin als eigenständiges Modul ist ebenfalls mit zusätzlichen Views einfach erweiterbar. Eclipse verwendet dafür XML-Beschreibungen, womit die Abhängigkeiten für ein Plugin festgelegt werden können. Neue Views - im e4 Application-Modell *Parts* genannt - sind der „fragment.e4xmi“-Datei hinzuzufügen. Jeder Part erhält seine eindeutige ID und einen Namen. Er wird einer bestimmten Kategorie zugeordnet (hier „SCC Detector“) und über eine Java-Klasse implementiert. Neben den Parts werden in der fragment.e4xmi auch Menüs, Symbolleisten, sowie deren Handler und Commands definiert.

Nachdem die Implementation des Plugins im Detail erläutert wurde, beschäftigt sich der folgende Abschnitt 5.4 mit der Implementation des Clone Detectors.

5.4 Clone Detector

Der Clone Detector verfügt ebenfalls wie das Plugin über einen Controller. Dieser implementiert zum einen das `ICloneDetector`-Interface (Abb. 5.3), zum anderen ist er zuständig für die Koordination der einzelnen Schritte im Clone Detection-Prozess (Abb. 4.1). Der Controller ist in der Klasse `SourcererController` implementiert und fungiert als Schnittstelle nach außen. Das Interface `ICloneDetectorConfiguration` ist in der Klasse `CloneDetectorConfigurator` implementiert und kümmert sich um die Verwaltung der Clone Detector-Einstellungen (siehe Abschnitt 5.4.1), sowie Pre-/Postprocessing des Security Code Repository. Die XML-Serialisierung bzw. Deserialisierung der Clone Detector-Einstellungen habe ich über die *Java API for XML Processing* [23] in der Klasse `PropertiesXmlManager` implementiert.

Wie in Abschnitt 4.3 zu lesen, habe ich die *Candidate Verification* von SourcererCC so angepasst, dass Intraprojekt-Clones eliminiert werden. Darüber hinaus habe ich neue Datenstrukturen dem SourcererCC hinzugefügt. Dieses war notwendig zum Verarbeiten des Formats der Tokens sowie der Metadaten.

Zum Protokollieren von möglichen Problemen und Fehlern verwendet der Clone Detector den `java.util.logging.Logger`. Diesen habe ich in der Klasse `SourcererLogger` implementiert. Der Logger kann Ereignisse der Kategorien *Info*, *Warning* und *Error* protokollieren. Genutzt werden zur Zeit *Info* und *Error*. Die Ausgabe erfolgt in eine XML-Datei „sourcerer.log“ im „log“-Verzeichnis des Ausgabeverzeichnisses des Clone Detectors (siehe Einstellungen Plugin Abschnitt 5.3.5).

Ein Paket- und ein Klassendiagramm des Clone Detectors sind im Anhang in den Abbildungen A.5 und A.6 zu sehen.

5.4.1 Einstellungen Clone Detector

Die Einstellungen für den Clone Detector und den Tokenizer sind in einer XML-Datei hinterlegt. Die Einstellungen habe ich bewusst nicht als *Eclipse Preference Page* implementiert, da sonst u.a. die Erweiterbarkeit nicht gegeben wäre. Jeder Clone Detector-Ansatz wird vermutlich über ein unterschiedliches Einstellungs-Set verfügen, so dass diese nicht als statische Preference Page implementiert werden können. Gleichzeitig soll eine Abhängigkeit vom Clone Detector-Modul auf das Plugin Modul bzw. Eclipse verhindert werden, da dieses als eigenständiges Modul implementiert ist. Die Einstellungen können über den Menüpunkt „Show Clone Detector Settings“ (Abbildung 5.4) aufgerufen werden. Die XML-Datei wird dann innerhalb der Eclipse-IDE angezeigt und kann dort bearbeitet werden. Nach dem Bearbeiten und Speichern reicht ein Klick auf den Menüeintrag „Reload Clone Detector Settings“, um die geänderten Einstellungen zu laden.

Das manuelle Neuladen ist notwendig, da Eclipse keine Möglichkeit bietet, Resource Listener für externe Dateien zu implementieren. Eine Alternative wäre somit nur, das Dateisystem des Betriebssystems auf Änderungen der Einstellungsdatei zu überwachen. Davon habe ich allerdings abgesehen, da es unnötige Rechenleistung vonseiten des Plugins erfordern würde.

Eine Einstellungsdatei ist im Anhang im Abschnitt A.3 zu sehen, wie auch eine Erläuterung dazu ebenfalls dem Anhang zu entnehmen ist.

Der nun folgende Abschnitt (5.5) widmet sich der Implementation des Tokenizers.

5.5 Tokenizer

Für die Implementation des Tokenizers habe ich das *Tee-and-join-Pipeline*-Architekturmuster gewählt. Es ist eine Modifikation des *Pipes and Filters*-Architekturmusters und ermöglicht einem Filter mehr als eine Ein-/Ausgabe zu nutzen (vgl. „Pattern-Orientated Software Architecture“ von Buschmann et al. [3]). Abbildung 5.12 stellt das Architekturmuster für den Tokenizer dar. Die aktuelle Implementation des Tokenizers erzeugt Tokens aus Java-

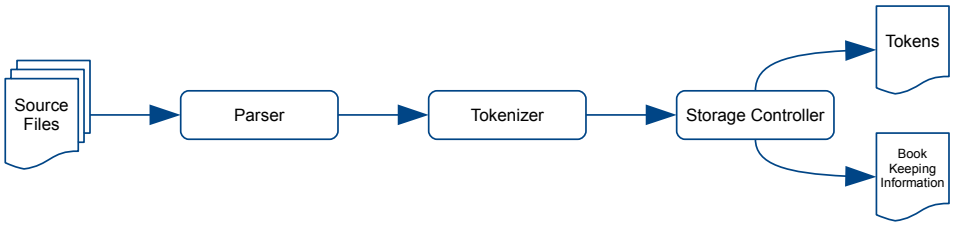

Abbildung 5.12: Tokenizer Tee-and-join-Pipeline

Code. Der *Parser* ist mit Hilfe der *JavaParser*-Komponente[3] realisiert. Der *Tokenizer* erhält vom Parser die Daten zu den gefundenen Methoden und Konstruktoren. Diese werden analysiert und in einzelne Tokens zerlegt. Whitespaces, Operatoren und Kommentare werden hierbei nicht verarbeitet. Zusätzlich lädt der Tokenizer beim Preprocessing die benötigten Metadaten aus der Datenbank des Security Repository. Anschließend übergibt der Tokenizer die Daten an den *Storage Controller*, der die Daten speichert. Der Storage Controller ist als Interface realisiert. So sind auch hier spätere Erweiterungen problemlos möglich. Die aktuelle Implementation des Storage Controllers ist dateibasiert. D.h. die Tokens und Book Keeping Information werden in zwei Dateien gespeichert. Es ist also auch problemlos möglich, einen Storage Controller zu implementieren, der Tokens und Book Keeping Information beispielsweise in einer Datenbank ablegt.

[3] `http://javaparser.org`

Das Paket- sowie das Klassendiagramm des Tokenizers sind im Anhang dargestellt. Abbildung A.3 zeigt das Paketdiagramm und Abbildung A.4 das Klassendiagramm.

Wie lässt sich nun eine weitere Programmiersprache für den Tokenizer implementieren? Diese Frage wird im folgenden Abschnitt 5.5.1 erklärt.

5.5.1 Prototypische Implementation einer weiteren Sprache

Bei der Entwicklung des Tokenizer habe ich stets die Erweiterbarkeit bedacht. So ist es, wie im Abschnitt 5.5 beschrieben, nicht nur möglich den Storage Controller zu erweitern, sondern auch Tokens einer anderen Programmiersprache erzeugen zu lassen. Dafür muss eine andere *Parser*-Komponente genutzt werden sowie eine Implementation der abstrakten Klasse `TokenizerBase` für die entsprechende Sprache erfolgen. Dieses lässt sich problemlos mit den verschiedenen Granularitätsleveln (4.2) kombinieren.

Prototypisch habe ich das für die Programmiersprache C# getan und somit die **optionale** Anforderung [R2.5] „Optional: Prototypische Implementation für eine weitere Sprache“ umgesetzt. Zur Erzeugung des Parsers habe ich den Parser Generator *ANTLR*[4] (ANother Tool for Language Recognition) genutzt. Dieser benötigt zur Generierung der Java-Dateien eine Grammatik. In diesem Fall für die Sprache C#, dessen Grammatik ich von Github heruntergeladen habe [5]. Mit dieser Grammatik erzeugt ANTLR einen Lexer und Parser in Java-Code aus dem ich einen C#-Parser für den Tokenizer programmiert habe. Zudem habe ich eine konkrete Klasse der `TokenizerBase`-Klasse implementiert und damit einen Tokenizer für die Programmiersprache C# umgesetzt. Der Code für den C# Parser befindet sich in einem gesonderten Java-Projekt mit Namen „CSharpParser“. Der von mir entwickelte Parser-Prototyp ist bereits zu Demonstrationszwecken in den Tokenizer integriert.

Eine Anpassung des Clone Detectors ist nicht notwendig, da dieser Token-basiert arbeitet und sprachunabhängig ist. Der letzte Abschnitt 5.6 des Implementationskapitels beschäftigt sich mit der Portabilität.

5.6 Portabilität

Eine der nicht funktionalen Anforderungen ist [NR9] „Die Software soll portabel sein“. Der Security Code Clone Detector arbeitet viel mit Dateien und Pfaden, so dass ich u.a. darauf geachtet habe, `File.separator` als Trennzeichen für Pfade zu nutzen, damit es bei Linux-Systemen nicht zu Problemen kommt. Dort ist ein Pfadtrennzeichen ein Slash ('/'), bei Windows i.d.R ein Backslash ('\'). Die Pfadangaben aus den Datenbanken im Security Repository werden ebenfalls normalisiert. Darüber hinaus wird in Dateien

[4] `http://www.antlr.org`

generell mit Hilfe des `java.io.OutputStreamWriter` und *UTF-8* Encoding geschrieben. Die Verwendung des Klasse `java.io.FileWriter` wäre nicht portabel. Hier wird das Standard-Encoding des Betriebssystems genutzt. Ein Konstruktor zur Angabe eines Encodings bietet Java hier leider nicht an. Das Lesen aus Dateien wird analog mit der `java.io.InputStreamReader`-Klasse und UTF-8 Encoding umgesetzt.

Überprüft habe ich die Portabilität des Security Code Clone Detectors auf einem Windows System (Spezifikation siehe 6.17), einem Ubuntu Linux und einem Mac OS. Auf allen drei Systemen läuft der Security Code Clone Detector als Eclipse Plugin und sucht erfolgreich nach Schwachstellen. Die Spezifikationen für die beiden Unix-Derivate sind in Tabelle 5.2 aufgeführt.

Software	**Version**
Ubuntu Linux	16.04 LTS
Java SE Development Kit 8	1.8.0_171
Eclipse Oxygen	Oxygen.3a Release (4.7.3a)
macOS High Sierra	10.13.4
Java SE Development Kit 8	1.8.0_171-b11
Eclipse Oxygen	Oxygen.3a Release (4.7.3a)

Tabelle 5.2: Portabilität: Spezifikation Software

Das Kapitel Implementation des Security Code Clone Detectors ist damit abgeschlossen. Ich habe alle Anforderungen (siehe Kapitel 2) umgesetzt und darüber hinaus auch die optionalen Zusatzanforderungen erfüllen können. Im Anschluss folgt nun die Evaluation (6).

Kapitel 6

Evaluation

Im Rahmen dieser Arbeit wurde der Security Code Clone Detector (SCC Detector) zum Auffinden von bekannten Schwachstellen in Java-Code entwickelt. In diesem Kapitel soll dieser mit Hilfe von bekannten Schwachstellen evaluiert werden. Dabei soll die Performanz hinsichtlich der Schwachstellenerkennung und in Bezug auf die Ausführungsgeschwindigkeit des Security Code Clone Detectors untersucht werden.

6.1 Performanz der Schwachstellenerkennung

6.1.1 Retrieval

Dieser Abschnitt untersucht, wie viele Schwachstellen vom Security Code Clone Detector aus einer Menge bekannter Schwachstellen mit existierenden CVE Einträgen gefunden, wie viele nicht gefunden werden bzw. wie viele Fehlfunde es gibt. Dieser Abschnitt stellt die verwendete Methodik vor. In 6.1.2 werden die Daten ausgewertet und anschließend in 6.1.3 ein Fazit gezogen.

Methodik

Zu Beginn wird auf die Auswahl von Schwachstellen eingegangen, sowie deren Modifikationen für die verschiedenen Clone-Typen (Typ 1 bis 3). Anschließend werden die verwendeten Metriken näher beleuchtet.

Auswahl der Schwachstellen

Die Auswahl der Schwachstellen erfolgte mit dem Security Code Exporter für Github, welches in Abschnitt 4.5 bereits diskutiert wurde. Anhand dieser Daten wurde eine Menge von Schwachstellen ausgewählt, für die CVE Einträge existieren. Dabei wurden insgesamt 102 durch den Security Code Exporter gefundene Sicherheitslücken gesichtet und auf ihre Eignung für

die Evaluation untersucht. Nicht geeignet haben sich Schwachstellen, die lediglich eine andere Bibliothek importieren. Ebenfalls für ungeeignet im Rahmen dieser Evaluation wurden Schwachstellen befunden, die Abhängigkeiten dynamisch laden bzw. dies mit Hilfe von Versionsangaben in Literalen tun. Des Weiteren fanden Schwachstellen, die wesentliche Code-Änderungen in einer Vielzahl von Methoden enthielten, keine Berücksichtigung. Hier wurden häufig nur Literale ausgetauscht, Exception Handling verändert oder ähnliches. Diese Änderungen enthielten nicht ausreichend Daten, um die Anpassung an die verschiedenen Clone-Typen möglichst realistisch zu gestalten und damit möglichst viel Code zu „verändern“.

In Tabelle 6.1 sind die 20 gewählten Schwachstellen aufgeführt mit Angabe der CVE, des *Scorings* sowie dem betroffenem Produkt. Die Code-Fragmente jeder einzelnen Schwachstelle wurde zudem so angepasst, dass die Sicherheitslücke weiterhin existiert, also das eigentliche Problem der Schwachstelle nicht behoben wurde, das Code-Fragment nun aber einen Typ-2 Clone respektive Typ-3 Clone darstellt.

CVE	**Scoring**	**Produkt**
CVE-2006-2806	7.8	Apache Java Mail Enterprise Server
CVE-2007-5461	3.5	Apache Tomcat
CVE-2007-6203	4.3	Apache HTTP Server 2.0.x and 2.2.x
CVE-2008-2086	9.3	Sun JDK
CVE-2008-5515	5.0	Apache Tomcat
CVE-2009-2693	5.8	Apache Tomcat
CVE-2009-2901	4.3	Apache Tomcat
CVE-2010-4172	4.3	Apache Tomcat
CVE-2011-1475	5.0	Apache Tomcat
CVE-2011-3190	7.5	Apache Tomcat
CVE-2011-3377	4.3	Ubuntu Linux, Opensuse, Redhat Icedtea-Web
CVE-2012-1621	4.3	Apache Open For Business Project
CVE-2012-2459	5.0	Bitcoind und Bitcoin-Qt
CVE-2016-3897	4.3	Google Android
CVE-2016-6723	5.4	Google Android
CVE-2017-0389	7.8	Google Android
CVE-2017-0846	5.0	Google Android
CVE-2017-1217	4.3	IBM WebSphere Portal
CVE-2017-1591	4.3	IBM DataPower Gateway
CVE-2017-9096	6.8	iTextpdf iText

Tabelle 6.1: Verwendete Schwachstellen

Ein Code Fragment einer solch existierenden Schwachstelle ist im Code-Listing 6.1 gegeben.

```
1 @Override
2 public String toString()
3 {
4   StringBuffer sb = new StringBuffer();
5   for (String key : mFields.keySet())
6     sb.append(key).append(" ").append(mFields.get(key)).append("\n");
7   return sb.toString();
8 }
```

Code-Listing 6.1: Schwachstelle CVE-2016-3897

Das Problem dieser Sicherheitslücke ist, dass ebenfalls ein Passwort im Klartext beim Aufrufen der `toString`-Methode zurückgegeben wird. Im Original lautet die Beschreibung dieser Schwachstelle

> „The WifiEnterpriseConfig class in net/wifi/WifiEnterpriseConfig.java in Wi-Fi in Android 4.x before 4.4.4, 5.0.x before 5.0.2, 5.1.x before 5.1.1, and 6.x before 2016-09-01 includes a password in the return value of a toString method call, which allows attackers to obtain sensitive information via a crafted application, aka internal bug 25624963.“ [1]

Der entsprechende Fix für die Schwachstelle ist im Code-Listing 6.2 dargestellt. Er soll an dieser Stelle nur zur Veranschaulichung dienen und zeigen, dass der Security Code Clone Detector dem Benutzer mögliche Fixes anbieten kann, sofern diese verfügbar sind. Es sei hier angemerkt, dass für die Evaluation keine Fixes oder Exploits von Sicherheitslücken genutzt bzw. betrachtet werden, da im Rahmen dieser Arbeit ausschließlich das Auffinden von bekannten Schwachstellen im Fokus steht.

```
1 @Override
2 public String toString()
3 {
4   StringBuffer sb = new StringBuffer();
5   for (String key : mFields.keySet())
6   {
7     // Don't display password in toString().
8     String value = PASSWORD_KEY.equals(key) ? "<removed>" : mFields.get(
      key);
9     sb.append(key).append(" ").append(value).append("\n");
10  }
11  return sb.toString();
12 }
```

Code-Listing 6.2: Fix für Schwachstelle CVE-2016-3897

[1] `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3897`, [29.03.2018]

Anpassen der Schwachstellen an die Clone-Typen

Um zu überprüfen, dass der Security Code Clone Detector nicht nur Typ-1 Clones findet, sondern auch Typ-2 und Typ-3 Clones, wurden die Code-Fragmente der gewählten Schwachstellen aus Tabelle 6.1 so angepasst, dass sie ebenfalls Typ-2 respektive Typ-3 Clones darstellen. Wie bereits weiter oben beschrieben, zeigt Abbildung 6.1 das Code-Fragment zur Schwachstelle mit CVE-2016-3897. Es stellt zugleich den Typ-1 Clone dieser Schwachstelle dar.

Das angepasste Typ-2 Clone Code-Fragment ist in Code-Listing 6.3 dargestellt. Hierfür wurde der Variablenname `sb` der StringBuffer-Instanz in `str` umbenannt (Zeile 5), ebenso wie die Variable `key` in `k` in Zeile 6. Zudem wurden in den Zeilen 4 und 8 noch Kommentare hinzugefügt.

```
1  @Override
2  public String toString()
3  {
4    // instantiate a StringBuffer
5      StringBuffer str = new StringBuffer();
6      for (String k : mFields.keySet())
7        str.append(k).append(" ").append(mFields.get(k)).append("\n");
8      // return the StringBuffer as string
9      return str.toString();
10 }
```

Code-Listing 6.3: Schwachstelle CVE-2016-3897 als Typ-2 Clone

Code-Listing 6.4 zeigt den Typ-3 Clone. Hier wurde die ursprüngliche StringBuffer-Instanzvariable `sb` aus 6.1 in `strBuffer` umbenannt (Zeile 5). Des Weiteren wurde die String-Konkatenation aus Zeile 6 von 6.1 hier in einzelne Statements (Zeile 24-27) überführt. Semantisch wurde das Code-Fragment also nicht verändert.

```java
@Override
public String toString()
{
  StringBuffer strBuffer = new StringBuffer();

  for (String k : mFields.keySet())
  {
      String value = PASSWORD_KEY.equals(k) ? "<removed>" : mFields.get(k)
    ;

      // seperate the statement into multiple lines
      strBuffer.append(k);
      strBuffer.append(" ");
      strBuffer.append(value);
      strBuffer.append("\n");
  }

  return strBuffer.toString();
}
```

Code-Listing 6.4: Schwachstelle CVE-2016-3897 als Typ-3 Clone

Metriken

Die Evaluierung erfolgt anhand von Methoden des *Information Retrievals*, wie Christopher D. Manning et al. sie in [18] beschreiben. Tabelle 6.2 zeigt eine Konfusionsmatrix, die die Berechnung von *Precision* und *Recall* veranschaulichen soll.

	Relevant	Nonrelevant
Retrieved	True Positives (TP)	False Positives (FP)
Not retrieved	False Negatives (FN)	True Negatives (TN)

Tabelle 6.2: Konfusionsmatrix in Anlehnung an [18], S.155

In Anlehnung an [18] ist der Recall in Bezug auf den Security Code Clone Detector definiert als: „$Recall(R)$ ist der Anteil von gefundenen Schwachstellen“ und wird durch folgende Formel ausgedrückt

$$Recall = \frac{TP}{TP + FN}$$

Die $Precision(P)$ ist der Anteil von Schwachstellen, die relevant sind.

$$Precision = \frac{TP}{TP + FP}$$

Tabelle 6.3 enthält zur Veranschaulichung von Precision und Recall ein fiktives Beispiel in Bezug zum Security Code Clone Detector. Das Beispiel enthält insgesamt 20 bekannte Schwachstellen (CVE vorhanden).

	Relevant (CVE vorhanden)	Nonrelevant (ohne CVE)
Retrieved	20 TP	5 FP
Not retrieved	0 FN	0 TN

Tabelle 6.3: Fiktives Beispiel Konfusionsmatrix

Die vom Clone Detector erkannten Schwachstellen werden als *True Positives* (TP) und nicht erkannte Schwachstellen als *False Negatives* (FN) bezeichnet. Neben 20 von 20 durch den Clone Detector erkannten Schwachstellen wurden 5 weitere Clones gefunden und fälschlicherweise als Schwachstelle ausgegeben. Bezeichnet werden diese als *False Positives* (FP). Die Anzahl der nicht relevanten Schwachstellen für z.B. diese Software sind 0. Bezeichnet werde diese als *True Negatives* (TN).

Der Recall in diesem Beispiel liegt bei

$$Recall = \frac{20\ TP}{20\ TP + 0\ FN} = 1$$

Der Security Code Clone Detector hat also 100% der Schwachstellen erkannt. Ein Recall von 0 würde bedeuten, dass keine der Schwachstellen gefunden worden wären. Die gefundenen 100% lassen auf einen sehr hohen Nutzen schließen. Allerdings sind sie für sich allein genommen aber noch nicht aussagekräftig. Schließlich wurden auch noch 5 irrelevante Schwachstellen gefunden. Also muss noch der Anteil bestimmt werden, der die für die Software relevanten Sicherheitslücken enthält. Dies wird über die Precision berechnet.

$$Precision = \frac{20\ TP}{20\ TP + 5\ FP} = \frac{4}{5}$$

Eine Precision von 1 bedeutet, dass alle Schwachstellen relevant sind. Eine Precision von 0 hingegen bedeutet, dass keine der Schwachstellen relevant ist. Im Beispiel aus 6.3 beträgt die Precision $\frac{4}{5}$ bzw. 80%.

Um eine möglichst hohe Precision zu erreichen, liegt häufig ein schlechterer Recall vor, wie folgender Sonderfall verdeutlichen soll. Angenommen der Security Code Clone Detector findet genau eine Schwachstelle von insgesamt 100 vorhandenen Schwachstellen, die auch relevant ist. Er würde hier eine Precision von 100% erreichen. Allerdings auf Kosten eines Recalls von nur 1%. Ein weiterer Sonderfall tritt ein, wenn keine Schwachstelle vom Security Code Clone Detector gefunden wird, trotz der Tatsache, das Sicherheitslücken vorhanden sind. Hier würde sich eine Precision von 1 (100%) ergeben, da sowohl Zähler als auch Nenner der Precision 0 sind, denn in der leeren Menge sind keine nicht relevanten Schwachstellen enthalten.

Daher stellt sich die Frage, wie sollte der Security Code Clone Detector Precision und Recall gewichten? Es sollten natürlich möglichst alle Schwachstellen gefunden werden und somit ein möglichst hoher Recall erzielt

werden. Wenn dieses Ziel aber auf Kosten einer geringen Precision erreicht wird, so dass der Entwickler viele Falschmeldungen erhält, könnte das im schlimmsten Fall dazu führen, dass der Entwickler den Security Code Clone Detector nicht mehr nutzt. Umgekehrt - also hohe Precision bei geringem Recall - würde ebenso wenig für Akzeptanz des Plugins sorgen. Denn wenn für den Entwickler zwar nur relevante Sicherheitslücken erkannt, aber nur insgesamt ein Bruchteil der Vorhandenen gefunden werden, muss der Entwickler entweder selber nach Schwachstellen suchen oder existierende Schwachstellen bleiben erhalten. Das Plugin soll ihm aber gerade diese Arbeit abnehmen.

Wie lässt sich nun die Performanz der Schwachstellenerkennung messen? Oder anders ausgedrückt, wie könne Precision und Recall ins Verhältnis gesetzt werden, damit eine Zahl daraus resultiert? Ein erster intuitive Ansatz wäre, das arithmetische Mittel von Precision und Recall zu bilden. Ein Blick auf folgenden Sonderfall sollte aber schnell deutlich machen, dass das auch nicht zielführend ist. Man stelle sich vor, der Security Code Clone Detector findet nicht eine einzige Sicherheitslücke. Somit würde der Recall bei 0 liegen, die Precision aber bei 1. Das arithmetische Mittel wäre in diesem Fall 50%.

Besser eignet sich hier das gewichtete harmonische Mittel, genannt *F measure*. Das resultierende Mittel liegt näher am kleineren Wert. F ist nach [18] wie folgt definiert

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R} \quad \text{, wobei } \beta^2 = \frac{1-\alpha}{\alpha}$$

wobei gilt $\alpha \in [0,1]$ und folglich $\beta \in [0,\infty]$. In dieser Arbeit sollen Precision und Recall gleich gewichtet werden, dann gilt $\alpha = \frac{1}{2}$ oder $\beta = 1$. Daraus ergibt sich der F_1 *measure*. F_1 ist die Kurzform für $F_{\beta=1}$. Dadurch vereinfacht sich die Formel zu

$$F_{\beta=1} = \frac{2PR}{P+R}$$

Für eben genannten Sonderfall würde sich somit ein F_1 von 0% einstellen und nicht 50%.

Anmerkung zu Precision und Recall bei der Evaluation

Der ***Recall*** wird anhand der ausgewählten Schwachstellen in Tabelle 6.1 unter Berücksichtigung der verschiedenen Clone-Typen gemessen.

Da in dieser Arbeit exakt bekannt ist, wie viele Schwachstellen (Clones) vorhanden sind, kann eine präzise Messung des Recalls erfolgen. Würde zum Beispiel nicht nur nach bekannten Security Code Clones - also bekannten Schwachstellen - gesucht werden, sondern einfach nur nach Code Clones innerhalb von Software-Projekten, wäre die Recall-Messung deutlich komplizierter, da nicht bekannt ist, wie viele Clones es tatsächlich gibt. Eine

manuelle Inspizierung der Daten wäre in solch einem Fall sehr aufwendig. Alternativ dazu legt Hitesh Sajnani et al in Abschnitt 4.3 von [28] dar, wie hierfür der Recall zur Evaluation der verschiedenen Clone-Detektoren gemessen wurde.

Die ***Precision***-Messung erfolgt hier anhand einer manuellen Validierung, d.h. es wird „händisch“ geschaut, ob die vom Security Code Clone Detector erkannten Clones mit denen aus Tabelle 6.1 übereinstimmen.

Die verschiedenen Konfigurationen des Security Code Clone Detectors sind in Tabelle 6.4 angegeben. Jede Schwachstelle (CVE) wird mit diesen Konfigurationen untersucht. Der Threshold-Wert ist für die Similarity-Funktion des Clone Detectors und wird im Folgenden auch wieder mit θ bezeichnet.

Iteration	Clone Type	Threshold (θ)	Granularity	Language
1	1	3,0	Method level	Java
2	1	5,5	Method level	Java
3	1	8,0	Method level	Java
4	2	3,0	Method level	Java
5	2	5,5	Method level	Java
6	2	8,0	Method level	Java
7	3	3,0	Method level	Java
8	4	5,5	Method level	Java
9	4	8,0	Method level	Java

Tabelle 6.4: Konfigurationen des Clone Detectors

6.1.2 Auswertung

Die Auswertung erfolgt anhand der Konfiguration in Tabelle 6.4. Jede Iteration durchläuft alle Schwachstellen, wobei der Clone Detector gemäß Tabelle 6.4 konfiguriert wird. Es werden jeweils Precision, Recall und F_1-Measure gemessen.

	Relevant (CVE vorhanden)	Nonrelevant (ohne CVE)
Retrieved	20 TP	0 FP
Not Retrieved	0 FN	0 TN
Summe	20	0

Tabelle 6.5: Konfusionsmatrix für Typ-1 Clones, θ 3.0, 5.5, 8.0

Tabelle 6.5 zeigt die Konfusionsmatrix für die Iterationen eins bis drei der Typ-1 Clones. Mit jedem der verschiedenen Threshold-Parameter wurden alle Schwachstellen gefunden. Dadurch ergeben sich für Recall, Precision und F_1 jeweils 100% wie in Tabelle 6.6 dargestellt. Die erste Spalte

gibt die Anzahl der gefundenen Schwachstellen und die Gesamtanzahl der Schwachstellen an.

Schwachstellen	θ	Recall	Precision	F_1
20/20	3,0	100	100	100
20/20	5,5	100	100	100
20/20	8,0	100	100	100

Tabelle 6.6: Ergebnis für Typ-1 Clones

Bei der Iteration vier und fünf wurden sowohl für den Threshold-Wert 3,0 sowie 5,5 ebenfalls alle Schwachstellen als Typ-2 Clones gefunden wie die Konfusionsmatrix in Tabelle 6.7 veranschaulicht.

	Relevant (CVE vorhanden)	Nonrelevant (ohne CVE)
Retrieved	20 TP	0 FP
Not Retrieved	0 FN	0 TN
Summe	20	0

Tabelle 6.7: Konfusionsmatrix für Typ-2 Clones, θ 3.0, 5.5

Iteration sechs, mit 8,0 als Threshold-Parameter, weist 12 False Negatives auf. Also 12 nicht entdeckte Schwachstellen. Von den acht gefundenen Schwachstellen gab es keine False Positives (Tabelle 6.8). Somit ergibt sich eine Precision von 100%. Der Wert für den Recall liegt bei 40%. Somit folgt ein F_1 von 57,14, dargestellt in der Ergebnistabelle 6.9 für die Typ-2 Clones. Die acht erkannten Sicherheitslücken sind in Tabelle 6.10 aufgelistet.

	Relevant (CVE vorhanden)	Nonrelevant (ohne CVE)
Retrieved	8 TP	0 FP
Not Retrieved	12 FN	0 TN
Summe	20	0

Tabelle 6.8: Konfusionsmatrix für Typ-2 Clones, $\theta = 8,0$

Das der Security Code Clone Detector hier 12 der 20 Schwachstellen für ein θ von $8,0$ nicht erkannt hat, liegt an der Arbeitsweise des verwendeten Clone Detectors. Im Folgenden soll das hier exemplarisch für die nicht gefundene Schwachstelle CVE-2017-9096 dargestellt werden. Code-Listing 6.5 enthält das ursprüngliche Code Fragment dazu.

Schwachstellen	θ	Recall	Precision	F_1
20/20	3,0	100	100	100
20/20	5,5	100	100	100
8/20	8,0	40	100	57,14

Tabelle 6.9: Ergebnis für Typ-2 Clones

CVE	Severity
CVE-2006-2806	7,8
CVE-2009-2693	5,8
CVE-2010-4172	4,3
CVE-2011-3190	7,5
CVE-2016-2723	5,4
CVE-2017-0389	7,8
CVE-2017-1217	4,3
CVE-2017-1591	4,3

Tabelle 6.10: Gefundene Typ-2 Schwachstellen-Clones für $\theta = 8,0$

```
1 public void fillXfaForm(InputSource is)
2     throws ParserConfigurationException, SAXException, IOException
3 {
4     DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
5     DocumentBuilder db = dbf.newDocumentBuilder();
6     Document newdoc = db.parse(is);
7     fillXfaForm(newdoc.getDocumentElement());
8 }
```

Code-Listing 6.5: Schwachstelle CVE-2017-9096

Der Tokenizer erzeugt daraus insgesamt 19 Tokens. Der Code des Typ-2 Clones, hier in Code-Listing 6.6 dargestellt, enthält ebenfalls 19 Tokens. Es wurden gemäß der Definition für Typ-2 Clones die Variablen umbenannt.

```
1 public void fillXfaForm(InputSource is)
2     throws ParserConfigurationException, SAXException, IOException
3 {
4     DocumentBuilderFactory a = DocumentBuilderFactory.newInstance();
5     DocumentBuilder b = a.newDocumentBuilder();
6     Document c = b.parse(is);
7     fillXfaForm(c.getDocumentElement());
8
9 }
```

Code-Listing 6.6: Schwachstelle CVE-2017-9096 Typ-2 Clone

Der verwendete Clone Detector nutzt nun u.a. das *Sub-block Overlap*

Filtering (siehe Kapitel 4.1) zur Clone-Suche. Das heißt, es gibt in diesem Fall zwei Code Blöcke B_x und B_y mit insgesamt 19 Tokens ($t = 19$). B_x bezieht sich auf Code-Listing 6.5 und B_y auf 6.6. Da ein θ von $8,0$ vorliegt, können $\lceil 0.8 * 19 \rceil = 16$ Tokens als Clones in Betracht gezogen werden. Somit ergibt sich ($i = 16$). Um nun zu schauen, ob es sich bei B_x und B_y um Clones handelt, müssen gemäß *Property 1* die ersten $t - i + 1 = 4$ Tokens ihrer *sub-blocks* sich mindestens in einem Token gleichen. Die Tokens der Blöcke B_x und B_y sind in Code-Listing 6.7 aufgeführt. Zur besseren Übersicht sind nur die ersten vier Tokens angegeben.

```
1 // Erste 4 Tokens Code-Block B_x
2 {DocumentBuilderFactory,Document,SAXException,getDocumentElement,...}
3
4 // Erste 4 Tokens Code-Block B_y
5 {a,InputSource,b,void,...}
```

Code-Listing 6.7: Tokens von 6.5 und 6.6

Es ist keine Übereinstimmung bei den ersten 4 Tokens zu finden. Also sind B_x und B_y nicht als Clones zu identifizieren und somit wird die Schwachstelle hier nicht als Security Code Clone identifiziert. Wenn aber nun das θ um 0,5 auf $7,5$ wieder reduziert wird, ergibt sich ein i von $\lceil 0.75 * 19 \rceil = 15$. Also müssen jetzt die ersten $t - i + 1 = 5$ Tokens eines sub-blocks übereinstimmen. In Code-Listing 6.8 ist das nun der Fall. Hier stimmt das Token „DocumentBuilderFactory“ überein. Folglich wird diese Schwachstelle bei einem θ von $7,5$ vom Security Code Clone Detector als Clone erkannt. Damit konnte hier exemplarisch durch das Sub-block Overlap Filtering gezeigt werden, dass nicht immer alle Clones bei den verschiedenen Konfigurationen gefunden werden können. Nachfolgend erfolgt die Auswertung der Typ-3 Clones.

```
1 // Erste 5 Tokens Code-Block B_x
2 {DocumentBuilderFactory,Document,SAXException,getDocumentElement,is,...}
3
4 // Erste 5 Tokens Code-Block B_y
5 {a,InputSource,b,void,DocumentBuilderFactory,...}
```

Code-Listing 6.8: Tokens von 6.5 und 6.6

Die Schwachstellensuche nach Typ-3 Clones wurde in den Iterationen sieben bis neun durchgeführt. Bei einem kleineren θ (hier 3,0) werden sämtliche Schwachstellen der Typ-3 Clones gefunden. Die Konfusionsmatrix dazu ist in Tabelle 6.11 zu finden. Steigt das θ wie in Iteration acht auf

	Relevant (CVE vorhanden)	Nonrelevant (ohne CVE)
Retrieved	20 TP	0 FP
Not Retrieved	0 FN	0 TN
Summe	20	0

Tabelle 6.11: Konfusionsmatrix für Typ-3 Clones, $\theta = 3,0$

5,5 werden nur noch 12 der insgesamt 20 Schwachstellen erkannt (Tabelle 6.12) und für $\theta = 8,0$ findet der Security Code Clone Detector keine der Schwachstellen mehr. Siehe Konfusionsmatrix 6.13.

	Relevant (CVE vorhanden)	Nonrelevant (ohne CVE)
Retrieved	12 TP	0 FP
Not Retrieved	8 FN	0 TN
Summe	20	0

Tabelle 6.12: Konfusionsmatrix für Typ-3 Clones, $\theta = 5,5$

	Relevant (CVE vorhanden)	Nonrelevant (ohne CVE)
Retrieved	0 TP	0 FP
Not Retrieved	20 FN	0 TN
Summe	20	0

Tabelle 6.13: Konfusionsmatrix für Typ-3 Clones, $\theta = 8,0$

Die Werte für Recall, Precision und F_1 sind für $\theta = 3,0$ 100%. Für $\theta = 5,5$ wurden 12 Schwachstellen gefunden. Daher ein Recall von 60% erreicht und eine Precision von 100%. Der F_1 liegt hier bei 75%. Da bei der neunten Iteration keine der Schwachstellen mehr gefunden wurde, liegt der Recall bei 0%. Die Precision liegt hier bei 100%. Die leere Menge enthält keine nicht relevante Sicherheitslücke. Der F_1 liegt bei 0%. Tabelle 6.14 zeigt diese Ergebnisse. Die 12 gefundenen Schwachstellen für $\theta = 5,5$ sind in Tabelle 6.15 aufgelistet.

6.1.3 Fazit

Der Security Code Clone Detector findet, gemäß der Anforderungen [R6] „Unterstützung der Clone-Typen 1 bis 3“, Schwachstellen-Clones der Typen eins bis drei. Dabei erreicht er für alle Clone-Typen ein Recall von 100%

Schwachstellen	θ	Recall	Precision	F_1
20/20	3,0	100	100	100
12/20	5,5	60	100	75
0/20	8,0	0	100	0

Tabelle 6.14: Ergebnis für Typ-3 Clones

CVE	Severity
CVE-2008-2086	9.3
CVE-2008-5515	5.0
CVE-2009-2693	5.8
CVE-2009-2901	4.3
CVE-2010-4172	4.3
CVE-2011-3190	7.5
CVE-2011-3377	4.3
CVE-2012-1621	4.3
CVE-2012-2459	5.0
CVE-2016-3897	4.3
CVE-2017-0389	7.8
CVE-2017-1217	4.3

Tabelle 6.15: Gefundene Typ-3 Schwachstellen-Clones für $\theta = 5,5$

und in dieser Evaluation eine Precision von ebenfalls 100%. Es hängt hier allerdings von der Wahl des θ ab, welcher Recall für den entsprechenden Clone-Typ erzielt werden soll. Welche Bedeutung θ im Hinblick auf die Ausführungsgeschwindigkeit hat, wird nun im folgenden Abschnitt 6.2 untersucht.

6.2 Performanz der Ausführungsgeschwindigkeit

Als weiterer Teil der Evaluation soll hier die Performanz im Hinblick auf die Ausführungsgeschwindigkeit des Security Code Clone Detectors betrachtet werden.

6.2.1 Methodik

Tabelle 6.16 listet noch einmal die gewählten Schwachstellen mit der jeweiligen Größe in Bytes und der Gesamtsumme aller Dateigrößen auf. Diese Daten liegen im Security Repository und werden vom Tokenizer vorverarbeitet (*Preprocessing*), damit dies nicht bei jeder Clone-Suche geschehen muss. Das ist möglich, da sich das Security Repository nicht ständig ändert, sondern nur nach Aktualisierungen durch den Benutzer erneut vorverarbeitet werden muss. Dieses Preprocessing erfolgt automatisiert, nachdem der

Benutzer im Menü des Security Code Clone Detectors den Befehl dazu gegeben hat.

CVE	Datei	Größe
CVE-2006-2806	SMTPHandler_pre_commit.java	16.765
CVE-2007-5461	WebdavServlet_pre_commit.java	108.888
CVE-2007-6203	Cookie_pre_commit.java	14.577
CVE-2008-2086	ICC_Profile_pre_commit.java	66.057
CVE-2008-5515	Request_pre_commit.java	75.366
CVE-2009-2693	JarUtils_pre_commit.java	13.933
CVE-2009-2901	WebappClassLoader_pre_commit.java	97.696
CVE-2010-4172	JspHelper_pre_commit.java	7.921
CVE-2011-1475	CoyoteAdapter_pre_commit.java	44.769
CVE-2011-3190	AbstractAjpProcessor_pre_commit.java	36.590
CVE-2011-3377	JNLPSecurityManager_pre_commit.java	7.921
CVE-2012-1621	SimpleMethod_pre_commit.java	48.612
CVE-2012-2459	MerkleTree_pre_commit.java	2.829
CVE-2016-3897	WifiEnterpriseConfig_pre_commit.java	21.856
CVE-2016-6723	PacManager_pre_commit.java	13.581
CVE-2017-0389	DhcpPacket_pre_commit.java	43.528
CVE-2017-0846	ClipboardService_pre_commit.java	16.712
CVE-2017-1217	PackagePaths_pre_commit.java	4.967
CVE-2017-1591	StringUtils_pre_commit.java	43.309
CVE-2017-9096	XfaForm_pre_commit.java	41.004
	Summe in Bytes	739.095

Tabelle 6.16: Verwendete Schwachstellen mit Dateigrößen in Bytes

Die Messung des Preprocessing-Vorgangs erfolgt zehn Mal hintereinander, um etwaige Lastspitzen - Lese-/Schreibaktivitäten, Scheduling etc. - des Betriebssystems zu filtern.

Des Weiteren werden die Zeiten jeder Iteration aus Tabelle 6.4 gemessen. Eine Iteration besteht hier aus dem Tokenizing der jeweiligen Java-Datei (mit der der Entwickler gerade arbeitet). In diesem Fall sind das die Clones der Schwachstellen, im Folgenden als Arbeitsdatei bezeichnet. Als nächstes folgt die Indexierung von Security Repository und Arbeitsdatei und anschließend der Clone-Suche im zuvor erzeugten Index.

Das System für die Geschwindigkeitsmessungen ist in Tabelle 6.17 angegeben.

6.2.2 Auswertung

Tabelle 6.18 listet die Daten der Preprocessing-Messung auf. Die erste Spalte gibt die Iteration an. Die zweite Spalte die absolut gemessene Zeit

Betriebssystem	Windows 8.1 64-Bit
CPU	Intel Core i5 M450 @ 2,4 GHz
RAM	4GB
Festplatte	Samsung 850 Evo
Eclipse-Version	Oxygen.3 Release (4.7.3)
Java-Version	1.8.0_171
Queue Threads	1

Tabelle 6.17: Testumgebung Geschwindigkeitsmessung

in Sekunden.

Durchlauf	**Zeit [in s]**
1	4,574
2	2,533
3	2,449
4	2,303
5	2,297
6	2,237
7	2,222
8	1,985
9	2,382
10	1,606

Tabelle 6.18: Messung Preprocessing

Abbildung 6.1 stellt die gemessenen Zeiten als Box Plot dar. Die „Box“ sind hier die mittleren 50% der Zeitmessung. Das untere Quartil entspricht den kleinsten 25%. Hier markiert durch den Anfang der Box. Das Ende der Box ist das 75%-Quartil. Der Strich innerhalb der Box ist der Median. Die Antennen links und rechts der Box werden auch Whisker genannt. Sie stehen für Minimum, respektive Maximum. Die Werte außerhalb der Antennen werden Ausreißer genannt. Weiterführende Informationen sind dem Buch

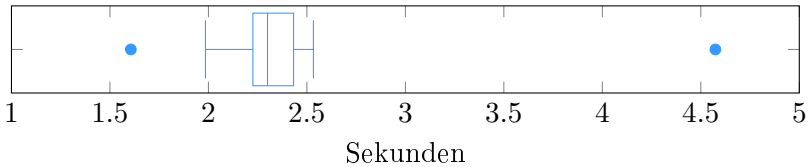

Abbildung 6.1: Zeiten des Preprocessings

„Beschreibende Statistik und Wirtschaftsstatistik“ von Mosler und Schmidt zu entnehmen [21], ab Seite 33ff.

Bei den gemessenen Zeiten für das Preprocessing liegt das Minimum

bei 1,606 Sekunden. Das Maximum bei 4,574 Sekunden, das 1. Quartil bei 2,162 Sekunden. Der Median liegt bei 2,3 Sekunden und das 3. Quartil bei 2,47 Sekunden. Auffallend ist das Maximum mit 4,574 Sekunden. Das die nachfolgenden Werte sich um 2 Sekunden einpendeln, ist möglicherweise auf Caching-Vorgänge des Betriebssystem zurückzuführen oder lag am aktuellen Scheduling des Betriebssystems.

Vorgang	Zeit in Millisekunden		
	$\theta = 3,0$	$\theta = 5,5$	$\theta = 8,0$
Tokenizing	109	234	256
Indexing	485	547	469
Searching	8563	3375	2438
Summe	9175	4156	3163

Tabelle 6.19: Geschwindigkeitsmessung Typ-1 Clones

Die Zeitmessungen für die Schwachstellensuche wurde, wie eingangs beschrieben, nur einmal pro Iteration durchgeführt. Die Zeiten für die Typ-1 Clone-Suche sind in der Tabelle 6.19 aufgeführt. Zu beobachten ist, dass die Zeiten für das Tokenizing und Indexing für die verschiedenen θ sich nur geringfügig voneinander unterscheiden. Der Vorgang der Clone-Suche dagegen unterscheidet sich deutlich, abhängig vom gewählten Threshold θ. Eine mögliche Erklärung ist hier das Sub-Block Overlap Filterung des Clone Detectors. Je kleiner θ, desto weniger Vergleiche müssen in den Sub-Blocks durchgeführt werden. Steigt θ, steigt auch die Anzahl der Vergleiche.

Die Ergebnisse der Zeitmessungen für die Typ-2 und Typ-3 Clones sind in den Tabellen 6.20, 6.21 dargestellt. Unabhängig vom Clone-Typ hängt die Suchzeit maßgeblich von der Wahl des θ ab.

Vorgang	Zeit in Millisekunden		
	$\theta = 3,0$	$\theta = 5,5$	$\theta = 8,0$
Tokenizing	156	217	121
Indexing	484	428	575
Searching	8268	3058	2574
Summe	8908	3703	3361

Tabelle 6.20: Geschwindigkeitsmessung Typ-2 Clones

6.2.3 Fazit

Die Anforderung [NR4] „Die Clone-Suche soll möglichst performant sein“ hat der Security Code Clone Detector erfüllt. Die Größe der Quellcodedatei für die Typ-1 Clones beträgt 45,4 KB mit 1061 LOC. Die Typ-2 Clone-Quelldatei umfasst 965 LOC und hat eine Dateigröße von 43,3 KB. 1071 LOC und

Vorgang	Zeit in Millisekunden		
	$\theta = 3,0$	$\theta = 5,5$	$\theta = 8,0$
Tokenizing	140	234	203
Indexing	438	453	547
Searching	7954	3157	2610
Summe	8532	3844	5767

Tabelle 6.21: Geschwindigkeitsmessung Typ-3 Clones

eine Größe von 47,3 KB hat die Quellcodedatei für die Typ-3 Clones. Die Geschwindigkeiten für Tokenizing und Indexing sind in allen neun Iterationen sehr ähnlich. Das Searching beansprucht hier den größten Zeitbedarf und ist stark abhängig von der Wahl des θ. Der Benutzer hat hier die Möglichkeit, nach eigenen Vorlieben eine geeignete Parameterkonfiguration zu wählen. Zusätzlich bieten die Clone Detector-Einstellungen noch die Option, die Anzahl der verwendeten Queue Threads zu erhöhen, was auf leistungsstarken Multi Core Workstations einen weiteren Geschwindigkeitszuwachs bringen kann.

Das nun folgende Kapitel 7 erörtert verwandte Arbeiten zum Security Code Clone Detector.

Kapitel 7

Verwandte Arbeiten

Dieses Kapitel soll einen Überblick über verwandte Arbeiten zum Thema Security Code Clone Detection geben.

7.1 CBCD

Entwickler übernehmen häufig Code Fragmente per Copy & Paste, um diese wiederzuverwenden oder zu bearbeiten. Dabei pflanzen sich mögliche Fehler aus dem kopierten bzw. modifizierten Code fort, sagen Li et al. und stellen in „CBCD: Cloned buggy code detector“ [16] das Tool CBCD (Cloned Buggy Code Detector) vor, das die Verbreitung von *Buggy Code* verhindern soll. Bei Li et al. geht es nicht direkt um die Suche nach bekannten Schwachstellen, sondern im Fokus steht vielmehr, die Verbreitung von Buggy Code zu verhindern. Zur Anwendung kommen PDG-basierte Code Clone Detection-Ansätze. PDG steht für *Program Dependency Graph*. Diese ermöglichen ebenfalls Typ-4 Clones zu erkennen. Li et al. beschreiben Ihren Ansatz in drei Schritten. In Schritt 1 werden aus dem Buggy Code und dem System, das nach Code durchsucht werden soll, je ein PDG generiert. Schritt 2 unterteilt den System-PDG zwecks Komplexitäts- und Kostenminimierung auf. In Schritt 3 wird schließlich geprüft, ob der Bug-PDG ein Untergraph des System-PDGs ist. Li et al. kommen zu dem Schluss, dass CBCD gute Erkennungsraten für Buggy Code Clones aufweist, jedoch der verwendete PDG-Generator zu langsam ist.

7.2 CLORIFI

Mittels Code Clone Verification (CLORIFI) sollen Schwachstellen in „real world programs“ gefunden werden. Da statische Code Analyse nicht ausreichend für die Suche nach Sicherheitslücken ist, so Li et al. [12], nutzen sie mit COLRIFI die Vorteile von statischer und dynamischer Code Analyse bei der Suche nach Schwachstellen. Dabei nutzt CLORIFI *Concolic Testing*,

um False Positives bei der Schwachstellensuche zu reduzieren. Concolic Testing ist eine Testvariante die sich aus *Con*crete und Symb*olic* Testing zusammensetzt. CLORIFI spürt Sicherheitslücken bei sehr niedriger False Positive-Rate innerhalb angemessener Zeiten auf, so Li et al.

7.3 ReDeBug

Jang et al. stellen mit *ReDeBug* ein syntaxbasiertes Code Clone Detection-System vor, das ungepatchte Schwachstellen aufspürt (vgl. „ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions“ [13]). ReDeBug ist sprachunabhängig, zudem schnell und besitzt eine geringe False Positive-Rate, sagen Jang et al. Ferner steht bei ReDeBug Skalierbarkeit im Fokus im Hinblick auf große Codearchive, so Jang et al. weiter. Das Erreichen einer niedrigen False Positive-Rate schafft ReDeBug durch nahezu exaktes Matching. Dadurch findet er zwar in Summe weniger Clones, erreicht aber gleichzeitig eine geringe False Positive-Rate, erklären Jang et al. Bei ReDeBug kommt kein Parser zum Einsatz. Der Quellcode wird „normalisiert“. Hierbei werden redundante Whitespaces (bis auf Zeilenumbrüche), Kommentare und nicht ASCII-Zeichen entfernt, geschweifte Klammern ignoriert und alle Zeichen in Kleinbuchstaben konvertiert. Daraus werden Tokens anhand von Zeilenumbrüchen und „*regex substrings*“ erzeugt. Ein Vergleich der Tokens erfolgt über eine Similarity-Funktion. Abschließend kommen Jang et al. zu dem Schluss, dass ReDeBug eine produktiv einsetzbare Lösung ist, um Entwickler bei Ihrer täglichen Arbeit zu unterstützen.

7.4 SCVD

Mit SCVD liefern Zou et al. in ihrer Arbeit „SCVD: A New Semantics-Based Approach for Cloned Vulnerable Code Detection“ [6] einen semantischen Clone Detection-Ansatz, um nach Schwachstellen zu suchen. Hierzu wird zu Beginn ein *Program Dependency Graph* (PDG) erzeugt. Um das NP-vollständige Problem der Suche in Teilgraphen zu umgehen, konvertieren Zou et al. den PDG in einen *Program Feature Tree*, wobei die Integrität der semantischen Informationen vollständig gewährleistet bleibt. Ihr Ansatz funktioniert aktuell nur für die Programmiersprache C/C++. SCVD liefert geringe False Negative-Raten. Das wird erreicht, da SCVD sowohl die syntaktischen als auch semantischen Informationen nutzen kann, so Zou et al. Die False Positive-Raten sind geringer als bei CBCD (7.1), jedoch höher als bei ReDeBug (7.3). Zou et al. heben die Baumsuche anstatt der NP-vollständigen Graphensuche von SCVD hervor, wobei sämtliche semantischen Informationen erhalten bleiben. Darüber hinaus sagen sie, dass die Erstellung des Program Feature Trees den PDG bei der Security Code Clone Detection vollständig ersetzen kann.

7.5 VFDETECT

Liu et al. stellen in Ihrer Arbeit „VFDETECT: A Vulnerable Code Clone Detection System Based on Vulnerability Fingerprint“ [17] ein System zur Schwachstellenerkennung vor, das auf *Fingerprints* basiert. Vorhandene Methoden der Security Code Clone Detection liefern eine große Anzahl an False Negatives bei Codeänderungen, so Liu et al. Daher haben sie den Ansatz gewählt, Security Code Clones mit Hilfe von Fingerprints aufzuspüren. Dabei folgen Sie der Annahme, dass die *Diff*-Dateien innerhalb eines Patches eine Schwachstelle charakterisiert. VFDETECT berechnet dann einen Fingerprint der Schwachstelle, basierend auf den Diff-Dateien und einem Preprocessing, dass sie ebenfalls in [17] beschreiben. Laut Liu et al. ist VFDETECT in der Lage die Clone-Typen 1 bis 4 zu erkennen. Durch die Verwendung von Fingerprints, die bereits im Vorfeld berechnet werden, ist VFDETECT auch für die Verwendung großer Codedaten qualifiziert bei guter Erkennungsrate, so Liu et al.

7.6 VUDDY

Bei VUDDY verfolgen Kim et al. einen effizienten und treffsicheren Ansatz der skalierbaren Erkennung von Schwachstellen-Clones in großen Software-Programmen (siehe „VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery“ [15]). Vuddy steht für VUlnerable coDe clone DiscoverY. Kim et al. erklären, dass VUDDY zweistufig arbeitet. Als erstes erfolgt das *Preprocessing*. Es besteht aus den drei Phasen *Function retrieval*, *Abstraction and normalization* und *Fingerprint generation*. In Phase 1 erfasst VUDDY mit Hilfe eines Parsers die Funktionen eines Programms. Während der Abstraction and normalization-Phase wenden Kim et al. vier verschiedene Abstraktionslevel an. Würde sofort ein Fingerprint berechnet werden, würden zum Beispiel Typ-2 Clones nicht erkannt werden und es würde zu False Negatives führen, so Kim et al. weiter. Der Fingerprint besteht aus einem 2-Tuple. Erstes Element ist die Länge des Code Fragments und das zweite Element der Hash. Gespeichert wird das Tuple in einem Dictionary, wobei die Länge als Schlüssel fungiert und der Hash als Wert. Die nächste Stufe ist die *Clone Detection*. Sie besteht aus den Phasen *Key Lookup* und *Hash Lookup*. Beim Key Lookup wird geschaut, ob ein passender Schlüssel zur prüfenden Funktion im Dictionary existiert. Ist dies nicht der Fall, liegt kein Clone vor. Sollte ein entsprechender Schlüssel vorhanden sein, greift Phase Hash Lookup. Hier wird nach dem passenden Hash geschaut. VUDDY ist sehr effektiv und effizient, so Kim et al. In der Evaluation wies er keine False Positive-Werte und sehr geringe False Negative-Werte auf bei gleichzeitiger hoher Geschwindigkeit.

7.7 VulPecker

Der *Vulnerability Pecker* (VulPecker) überprüft Quellcode auf Schwachstellen. Li et al. beschreiten hierbei einen etwas anderen Weg, den sie in „VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis“ [32] veröffentlicht haben. Li et al. sehen sich bei ihrem Ansatz des Security Code Clone Detections mit zwei Problemen konfrontiert. Erstens ist nicht jeder Code-Similarity Algorithmus für jede Art von Schwachstelle geeignet, so Li et al. und zweitens gibt es zwar Schwachstellendatenbanken wie die National Vulnerability Database (NVD), die jedoch nur die Schwachstellen beschreiben, aber nicht immer mit Code Patches verknüpft sind bzw. die benötigten Daten für VulPecker liefern. Infolgedessen haben Li et al. für das zweite Problem eine eigene Schwachstellendatenbank auf Grundlage der NVD aufgebaut. Bisher für C/C++ Code von Open Source-Produkten. Für die Lösung des ersten Problems wählen sie den vielversprechendsten Algorithmus zur Security Code Clone Detection aus, der während einer Anlernphase des VulPeckers am besten für den jeweiligen Schwachstellentyp geeignet ist. Li et al. konnten in Ihrer Arbeit demonstrieren, dass VulPecker sogar in der Lage war, 40 bisher nicht in der NVD veröffentlichte Schwachstellen zu finden. Darüber hinaus konnten sie sogar zeigen, dass eine von Mozilla für Firefox adressierte Schwachstelle ebenfalls in Thunderbird existierte.

7.8 Abgrenzung

Die in diesem Kapitel beschriebenen Arbeiten verfolgen unterschiedliche Ansätze bei der Suche nach Schwachstellen im Quellcode und verfahren dabei ebenfalls anders als SourcererCC, der beim Security Code Clone Detector zur Anwendung kommt. Die Autoren von SourcererCC verfolgten nicht das primäre Ziel Security Code Clones zu finden, sondern allgemein Code Clones ohne Bezug zu Sicherheitslücken. Von den vorgestellten Arbeiten machen VFDETECT, VUDDY und VulPecker, was die Erkennungsraten von Schwachstellen angeht, einen guten Eindruck. Für die Praxis ungeeignet halte ich ReDeBug, aufgrund des Design-Kriteriums weniger Schwachstellen zu finden, dafür aber präziser zu sein. Es macht aus meiner Sicht mehr Sinn, eher eine Schwachstelle zu viel zu finden, bei der sich hinterher herausstellt, dass es keine Schwachstelle ist, als eine zu übersehen.

Keine der genannten Arbeiten beschäftigt sich mit der Integration der Schwachstellensuche in eine Entwicklungsumgebung (IDE), wie ich es bei dem von mir entwickelten Security Code Clone Detector realisiert habe. Sajnani et al. haben zwar für SourcererCC ([28]) ein Eclipse Plugin entwickelt und stellen es auf ihrer Internetseite zum Download bereit (siehe SourcererCC-I [27]), jedoch steht die Suche nach Clones hier nicht im

Security-Kontext. Es geht dabei um das Auffinden von Intraprojekt-Clones.

Ein weiteres Eclipse Plugin, das keinen direkten Bezug zur Security Code Clone-Suche hat ,ist in der Arbeit „An Eclipse Plugin to Support Code Smells Detection“ von Pessoa et al. ([24]) zu finden. Die Autoren stellen hier ein Plugin vor, das nach *Code Smells* sucht. Mit Code Smells wird Code bezeichnet, der fehlerhaft und/oder sogar eine Sicherheitslücke enthalten kann. Darüber hinaus aber auch schlecht formulierter Code, der schwer zu verstehen ist oder einfach nur Code, der gegen Coding-Konventionen verstößt.

Im Eclipse Marketplace[1] sind zu den Suchbegriffen „security code clone“ und „vulnerable code clone“ folgende Plugins zu finden:

- Early Security Vulnerability Detector - ESVD [30]
- Veracode Greenlight [31]
- Code Dx Eclipse [4]

Early Security Vulnerability Detector prüft während des Schreibens von Code auf eine Menge von vordefinierten Schwachstellenarten wie Command Injection, Cross-Site Scripting (XSS), SQL Injection usw. Der Benutzer erhält in einem View einen Überblick über die Schwachstellenfunde. Das Plugin steht unter der Eclipse Public License[2].

Veracode Greenlight sucht ebenfalls nach Sicherheitslücken wie SQL Injection oder Cross Site Scripting und gibt zudem die assoziierte CWE[3] mit aus. Das Plugin ist kostenpflichtig.

Code Dx ist ein „Application Vulnerability Correlation“ und „Management System“. Es ermöglicht die Priorisierung und Verwaltung von Schwachstellen. Diese können zudem im Quellcode hervorgehoben werden. Code Dx ist ebenfalls kostenpflichtig und besteht mit seinen Management-Funktionen aus einer ganzen Tool Suite, die weit über Security Code Clone Detection hinausgeht.

[1] https://marketplace.eclipse.org/
[2] https://opensource.org/licenses/EPL-1.0
[3] https://cwe.mitre.org/

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Der Security Code Clone Detector wurde entwickelt, um Entwickler auf Schwachstellen in ihrem Java-Code hinzuweisen. Durch Code Clones können sich Fehler und Schwachstellen leicht ausbreiten, was ein großes Problem bei Wartung und Fehlerbehebung darstellen kann, da jeder dieser Clones gefunden und gepachtet werden muss. Der Security Code Clone Detector wurde als Eclipse Plugin realisiert und kann somit von Beginn an in den Entwicklungsprozess integriert werden. Das bietet die Möglichkeit, den Entwickler frühzeitig auf Schwachstellen aufmerksam zu machen, so dass dieser rechtzeitig das Problem beheben kann.

Der Security Code Clone Detector nutzt zur Suche nach Schwachstellen im Java-Code einen Clone Detector, der aus einer Menge bekannter Schwachstellen, die in einem Security Code Repository liegen, nach Security Code Clones im aktuell geöffneten Eclipse Workspace sucht. Die Sicherheitslücken im Security Code Repository enthalten den Java-Code sowie entsprechende CVE-Daten. Es ist auch möglich eigenen Code, der Sicherheitslücken enthält, einzupflegen.

Die Suche nach Schwachstellen läuft automatisch im Hintergrund, während der Entwickler Java-Code schreibt. Dabei reagiert die automatische Suche entweder auf Save Events der Codedateien oder läuft periodisch innerhalb eines frei definierbaren Zeitintervalls ab. Die Genauigkeit mit der der Clone Detector nach Schwachstellen sucht, ist ebenfalls einstellbar.

Werden Schwachstellen gefunden, zeichnet es den Security Code Clone Detector aus, dass der Entwickler eine Übersicht aller gefundenen Schwachstellen erhält. Gleichzeitig werden diese direkt im Code Editor von Eclipse farbig hervorgehoben. Der Entwickler hat anschließend die Möglichkeit, sich weitere Informationen zur gefundenen Schwachstelle anzeigen zu lassen

sowie auch mögliche Exploits und Fixes anzeigen zu lassen, sofern diese im Security Code Repository vorhanden sind. Der Security Code Clone Detector unterstützt zudem Typ-1 bis Typ-3 Clones.

8.2 Ausblick

Bei der Evaluation konnte gezeigt werden, wie viele Schwachstellen der Security Code Clone Detector aus einer Menge gegebener Schwachstellen tatsächlich erkennt, wobei jede Schwachstelle entsprechend der Clone-Typdefinitionen aus Abschnitt 3.2.5 modifiziert wurde. In einem nächsten Schritt wäre interessant zu schauen, welche False Positive-Rate der Security Code Clone Detector aufweist, wenn größere Mengen an Code hinzugefügt werden, der keine Sicherheitslücken enthält.

Anzustreben wäre auch, dass der Security Code Clone Detector in Zukunft auch Typ-4 Clones erkennen kann. Dazu müsste ein weiterer Clone Detector hinzugefügt werden, der Typ-4 Clones unterstützt. Eine weitere Optimierung im Hinblick auf die Ausführungsgeschwindigkeit des Preprocessings wie auch der Clone-Suche selbst könnten darin bestehen, die Codedateien im Security Code Repository zu kürzen, so dass diese nur noch die relevanten Methoden und Code Fragmente enthalten, die tatsächlich der Schwachstelle zuzuordnen sind. Die Unterstützung für weitere Programmiersprachen beim Tokenizer könnte zukünftig auch realisiert werden, um somit diese auch auf Security Code Clones untersuchen zu können.

Ein weiterer Weg die Geschwindigkeit zu erhöhen und gleichzeitig die Wartung des Security Repositories zu erleichtern wäre *Clone Detection as a Service*. Da die Größe des Security Code Repositories mit der Zeit immer weiter anwachsen wird und in Unternehmen somit jeder Entwickler eine aktuelle Spiegelung des neuesten Zustands dieses Repositories benötigt, wäre eine Möglichkeit, das Preprocessing und den Prozess der Clone Detection auszulagern. Man könnte dazu das Security Code Repository an zentraler Stelle speichern und nicht an jedem Arbeitsplatz. So gäbe es nur noch ein stets aktuelles Repository. Darüber hinaus würde der Security Code Clone Detector nur noch die lokalen Codedateien vom Tokenizer verarbeiten lassen und diese über einen *Clone Detection Service* an einen leistungsstarken Rechner senden, auf dem dann die Suche nach Schwachstellen erfolgt. Zurückgeliefert wird anschließend das Ergebnis, das der Security Code Clone Detector, wie er es jetzt schon leistet, grafisch aufbereitet.

Darüber hinaus könnten weitere Security Code Clone Detection-Ansätze integriert werden, dafür ist der Security Code Clone Detector dank der API bereits vorbereitet. Im Besonderen sehen hier die Clone Detection-Ansätze (Kapitel 7) VFDETECT, VUDDY und eventuell auch VulPecker aus dem vielversprechend aus.

Anhang A

Anhang

A.1 UML Klassen- und Paketdiagramme

Abbildung A.1: Klassendiagramm Package de.luh.se.sccd.api

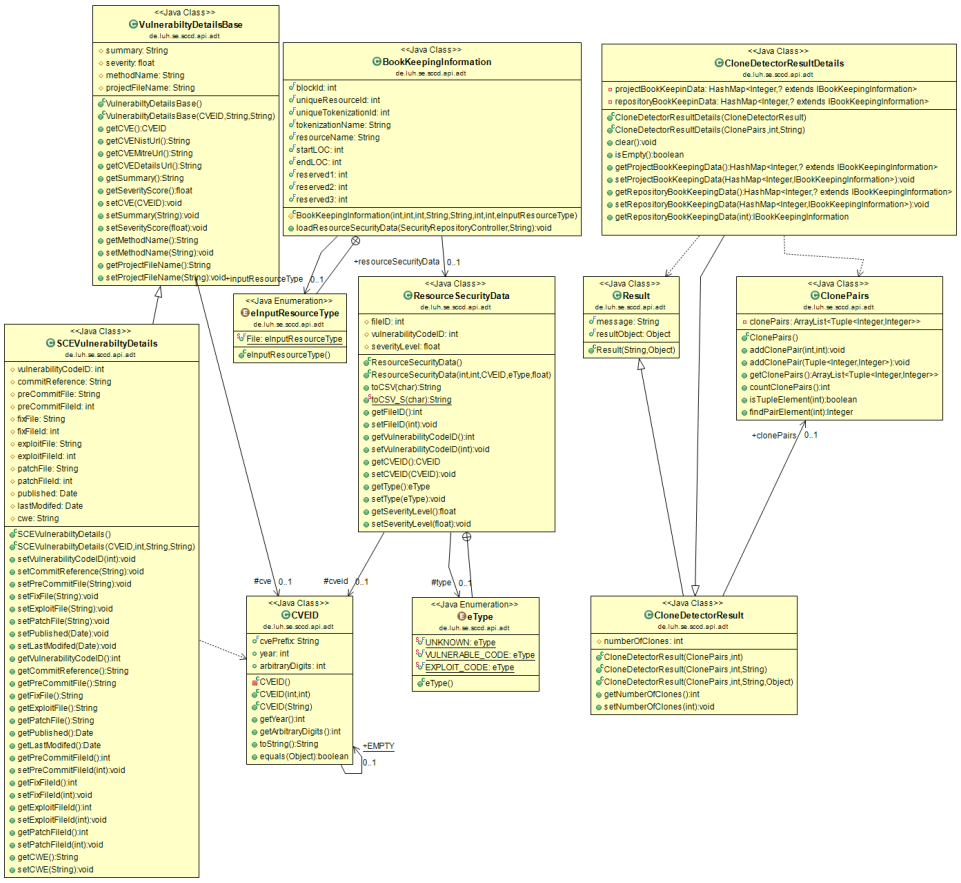

Abbildung A.2: Klassendiagramm Package de.luh.se.sccd.api.adt

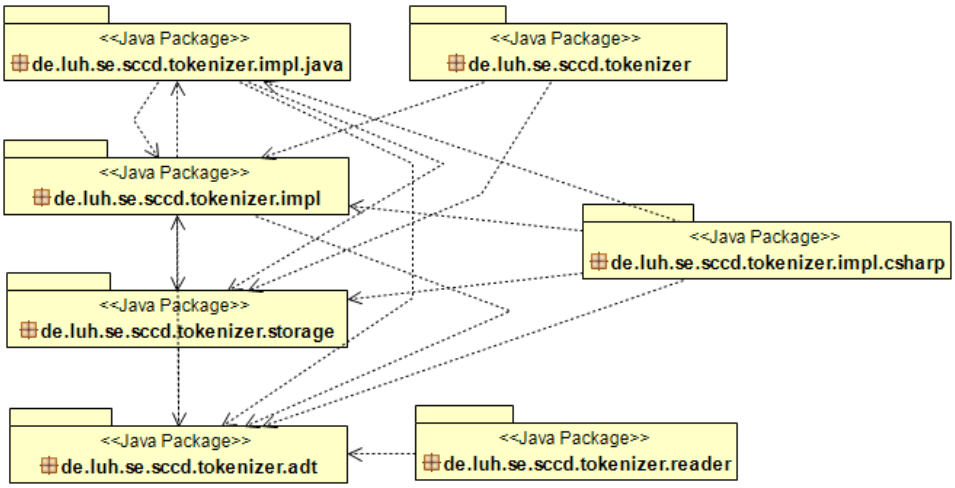

Abbildung A.3: Paketdiagramm Tokenizer

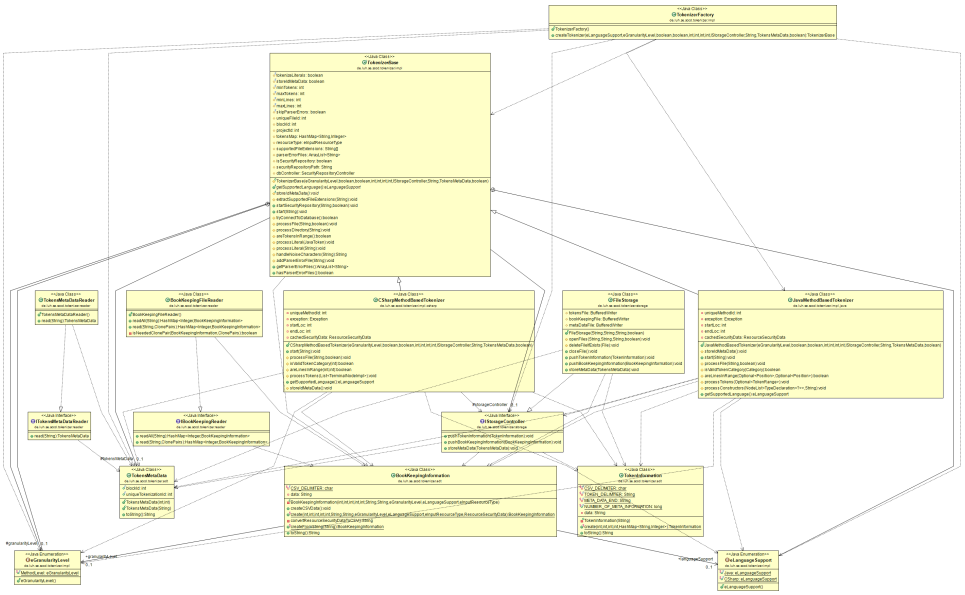

Abbildung A.4: Klassendiagramm Tokenizer

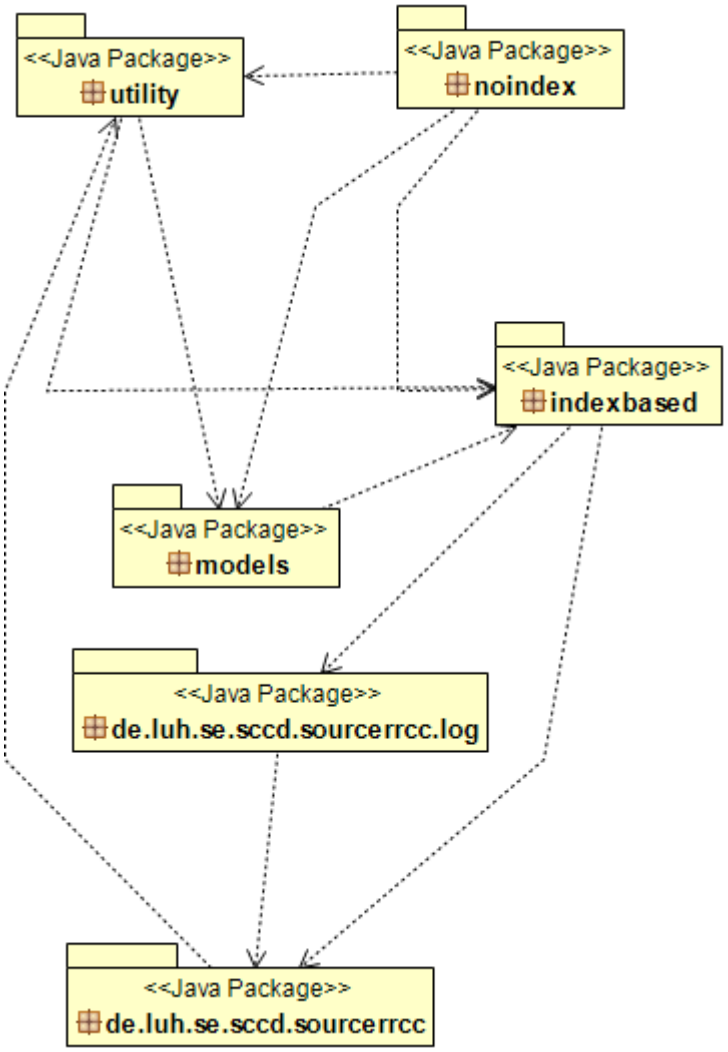

Abbildung A.5: Paketdiagramm Clone Detector

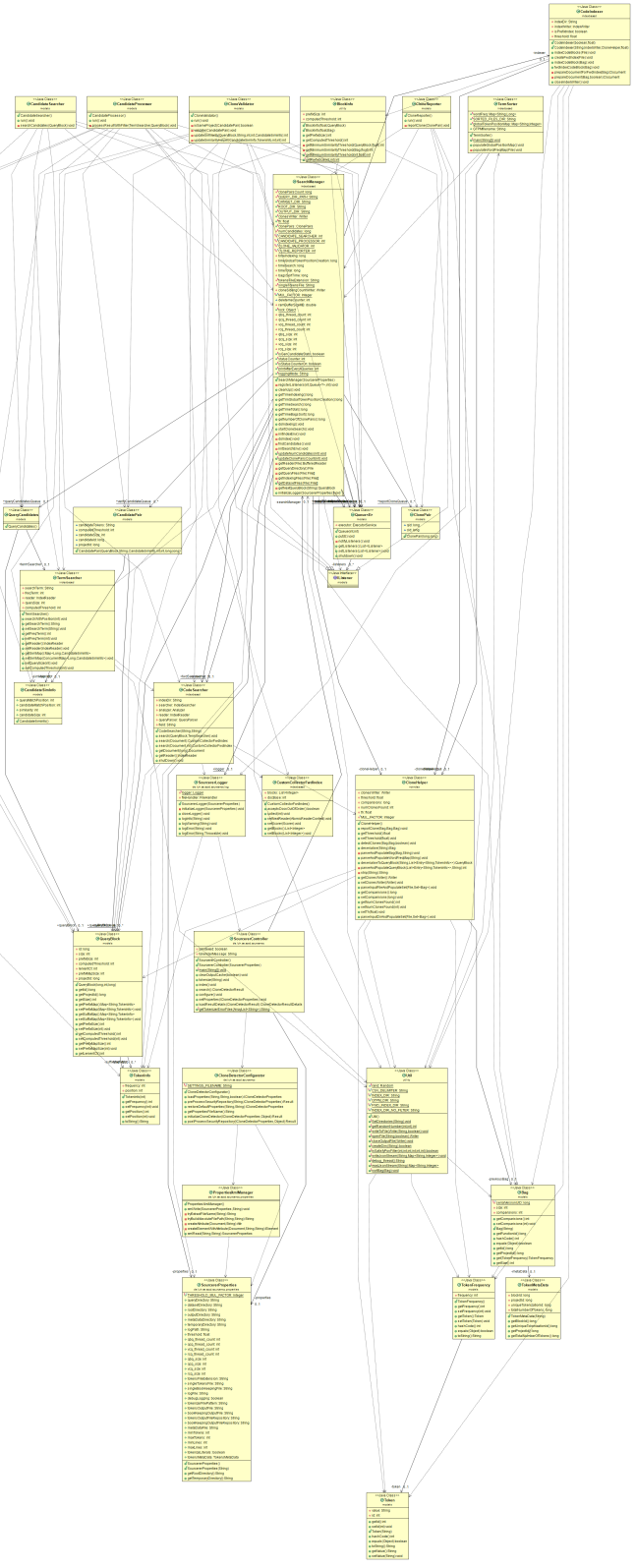

Abbildung A.6: Klassendiagramm Clone Detector

A.2 Plugin Einstellungen

Zur Erläuterung ist der Einstellungsdialog in Abbildung A.7 mit Zahlen versehen. Es folgt eine Erklärung anhand der Nummerierung.

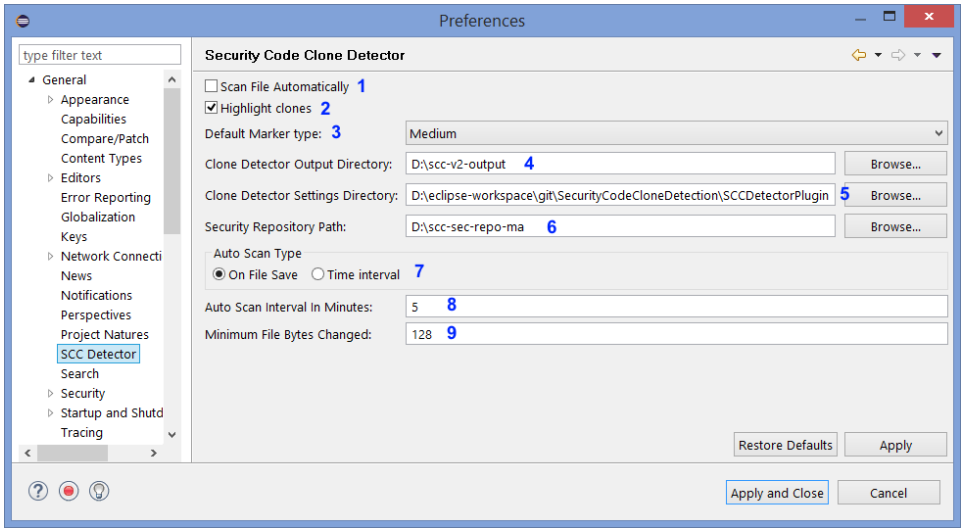

Abbildung A.7: Plugin Preference Page

1. Aktiviert/Deaktiviert die automatische Schwachstellensuche
2. Ist der Haken gesetzt, werden die gefundenen Schwachstellen im Code Editor farblich hervorgehoben
3. Für Schwachstellen ohne CVE-Daten kann eine Standardfarbe zur Hervorhebung im Code Editor eingestellt werden
4. Arbeitsverzeichnis des Clone Detectors. Muss beim ersten Plugin-Start gesetzt werden
5. Pfad an dem die Einstellungen für den Clone Detector abgelegt werden sollen. Muss beim ersten Plugin-Start gesetzt werden
6. Pfad zum Security Code Respository. Muss beim ersten Plugin-Start gesetzt werden
7. Festlegen, ob beim Speichern oder in einem Zeitinvervall nach Schwachstellen gesucht werden soll
8. Zeitintervall für die Schwachstellensuche in Minuten
9. Untere Grenze der Dateigrößenänderung

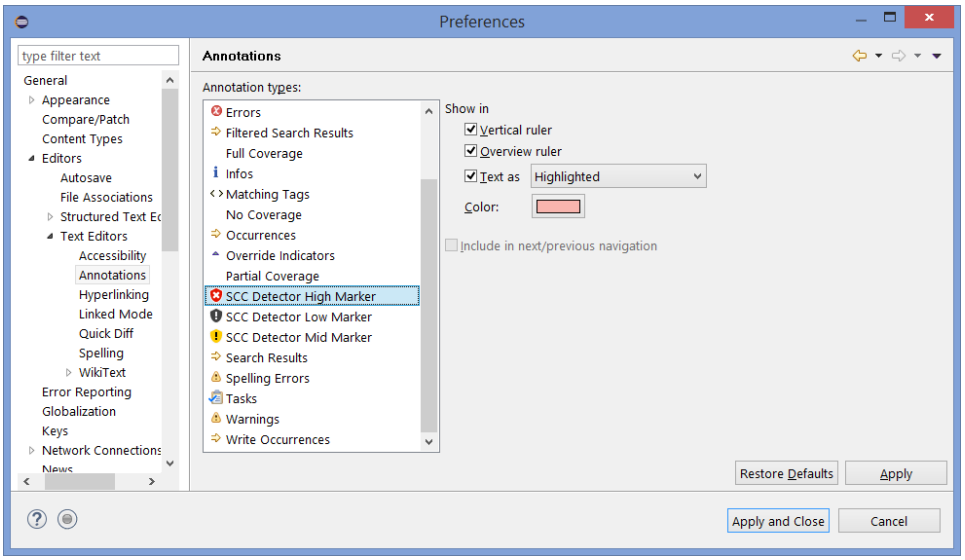

Abbildung A.8: Plugin Preference Page für Marker

A.3 Clone Detector Einstellungen

Code-Listing A.1 zeigt den Inhalt der XML-Datei für die Clone Detector- und Tokenizer Einstellungen. Beide Einstellungen sind in einer Datei untergebracht, da der Tokenizer vom Clone Detector aufgerufen wird und die Werte als Parameter übergibt.

Die Zeilen 3-15 zeigen die Einstellungen für den Clone Detector. An dieser Stelle sei nur der Tag `threshold` erwähnt. Dieser entspricht dem Parameter θ für den Clone Detector. Zur Erläuterung der übrigen Clone Detector Einstellungen sei auf die Datei „SourcererProperties.java“ im Projekt „SCCDetector-v2-MA“ verwiesen. Im Quellcode sind die einzelnen Werte erläutert.

Die Zeilen 16-25 sind für den Tokenizer. Die Einstellung `tokenizerFilePattern` legt das Pattern der Dateiendung fest. Diese Dateitypen werden vom Tokenizer verarbeitet. Alle übrige Dateitypen in einem Verzeichnis werden ignoriert. `minTokens` und `maxTokens` legen die untere und obere Grenze für Anzahl an Tokens fest, die eine Methode enthalten muss, damit diese verarbeitet wird. `minLines` und `maxLines` legt die minimale bzw. maximale Zeilenanzahl einer Methode fest. Die Werte 0 bedeuten, das Minimal- und Maximalangaben ignoriert werden. Die übrigen Einstellungen werden ebenfalls in der Datei „SourcererProperties.java“ erläutert.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<sourcerer-properties>
  <threshold value="3.0"/>
   <qbq_size value="3"/>
   <qcq_size value="3"/>
   <vcq_size value="3"/>
   <rcq_size value="3"/>
   <qbq_thread_count value="1"/>
   <qcq_thread_count value="1"/>
   <vcq_thread_count value="1"/>
   <rcq_thread_count value="1"/>
   <tokensFileExtension value=".tokens"/>
   <singleTokensFile value=""/>
   <singleBookkeepingFile value=""/>
   <debugLogging value="true"/>
   <tokensOutputFile value="plugin.tokens"/>
   <bookkeepingOutputFile value="plugin.bookkeeping"/>
   <tokensOutputFileRepository value="security-repository.tokens"/>
   <bookkeepingOutputFileRepository value="security-repository.bookkeeping
    "/>
   <minTokens value="0"/>
   <maxTokens value="0"/>
   <minLines value="0"/>
   <maxLines value="0"/>
   <tokenizeLiterals value="true"/>
   <tokenizerFilePattern value=".java"/>
</sourcerer-properties>
```

Code-Listing A.1: Clone Detection Einstellungen

Literaturverzeichnis

[1] M. Broy. Challenges in Automotive Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[2] BSI. Botnetze. https://www.bsi-fuer-buerger.de/BSIFB/DE/Risiken/BotNetze/bot_netze.html, -. Letzter Zugriff 09.04.2018.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Orientated Software Architecture*, pages 53–70. John Wiley & Sons, New York, USA, volume 1 edition, 1996.

[4] Code Dx. Code Dx Eclipse. https://marketplace.eclipse.org/content/code-dx-eclipse, 2018. Letzter Zugriff 02.05.2018.

[5] G. Comun. C# Grammar for ANTLR Parser Generator. https://github.com/antlr/grammars-v4/tree/master/csharp, 2015. Letzter Zugriff 02.05.2018.

[6] Z. Deqing, Q. Hanchao, L. Zhen, W. Song, J. Hai, S. Guozhong, W. Sujuan, and Z. Yuyi. SCVD: A New Semantics-Based Approach for Cloned Vulnerable Code Detection. In M. Polychronakis and M. Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 325–344, Cham, 2017. Springer International Publishing.

[7] P. T. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 227–239. ACM, 2000.

[8] N. Edgar, K. Haaland, J. Li, and K. Peter. Eclipse User Interface Guidelines Version 2.1. https://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html, 2004. Letzter Zugriff 05.05.2018.

[9] S. Frei, D. Schatzmann, B. Plattner, and B. Trammell. Modeling the Security Ecosystem - The Dynamics of (In)Security. In *Economics of*

Information Security and Privacy, pages 79–106, Boston, MA, 2010. Springer US.

[10] W. Halfond, J. J. Viegas, A. Orso, et al. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[11] J. A. Hölzing. *Die Kano-Theorie der Kundenzufriedenheitsmessung*. Gabler Verlag | GWV Fachverlage GmbH, Wiesbaden, Deutschland, 2008.

[12] L. Hongzhe, K. Hyuckmin, K. Jonghoon, and L. Heejo. CLORIFI: software vulnerability discovery using code clone verification. *Concurrency and Computation: Practice and Experience*, 28(6):1900–1917, 2015.

[13] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *2012 IEEE Symposium on Security and Privacy*, pages 48–62, May 2012.

[14] R. Jochem. *Was kostet Qualität? - Wirtschaftlichkeit von Qualität ermitteln*, pages 27–54. Carl Hanser Verlag GmbH & Co. KG, München, Deutschland, 2010.

[15] S. Kim, S. Woo, H. Lee, and H. Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017.

[16] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 310–320, June 2012.

[17] Z. Liu, Q. Wei, and Y. Cao. VFDETECT: a vulnerable code clone detection system based on vulnerability fingerprint. In *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*, pages 548–553, Oct 2017.

[18] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, United Kingdom, 2009.

[19] C. P. Mayer. Security and privacy challenges in the internet of things. *Electronic Communications of the EASST*, 17, 2009.

[20] C. Müller. Security Code Exporter für Github. Bachelorarbeit, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2018.

[21] K. Mosler and F. Schmid. *Beschreibende Statistik und Wirtschaftsstatistik*. Springer, Heidelberg, Deutschland, 2006.

[22] F. M. Nicastro. *Security Patch Management*, pages 19–21. CRC Press, Boca Raton, FL, USA, 2011.

[23] Oracle. Java API for XML Processing. `http://www.oracle.com/technetwork/java/intro-140052.html`, 2008. Letzter Zugriff 02.05.2018.

[24] T. Pessoa, F. B. e Abreu, M. P. Monteiro, and S. Bryton. An Eclipse Plugin to Support Code Smells Detection. *CoRR*, abs/1204.6492, 2012.

[25] M. Rohr. Sicherheit im Software-Entwicklungsprozess. `https://www.informatik-aktuell.de/betrieb/sicherheit/sicherheit-im-software-entwicklungsprozess.html`, 2015. Letzter Zugriff 15.03.2018.

[26] H. Sajnani, V. Saini, and C. V. Lopes. Scaling Token-Based Code Clone Detection. `http://mondego.ics.uci.edu/projects/clonedetection/`. Letzter Zugriff 02.04.2018.

[27] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: Scaling Type-3 Clone Detection to Large Software Repositories. `http://mondego.ics.uci.edu/projects/SourcererCC/`. Letzter Zugriff 02.04.2018.

[28] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: Scaling Code Clone Detection to Big-Code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, May 2016.

[29] A. Sheneamer and J. Kalita. A Survey of Software Clone Detection Techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[30] thecodemaster. Early Security Vulnerability Detector - ESVD. `https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd`, 2018. Letzter Zugriff 02.05.2018.

[31] Veracode. Veracode Greenlight. `https://marketplace.eclipse.org/content/veracode-greenlight`, 2018. Letzter Zugriff 02.05.2018.

[32] L. Zhen, Z. Deqing, X. Shouhuai, J. Hai, Q. Hanchao, and H. Jie. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 201–213, New York, NY, USA, 2016. ACM.

Abbildungsverzeichnis

List of Code-Listings