Modelbasierte Analyse von Sicherheitsschwachstellen in objektorientierten Modulen

Proposal für Diplomarbeit am
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Software-Entwurf und -Qualität
Prof. Dr. Ralf Reussner
Fakultät für Informatik
Universität Karlsruhe (TH)

von

cand. inform.
Christina Pildner

Betreuer:
Prof. Dr. Ralf Reussner
Dipl.-Inform. Pierre Parrend

Tag der Anmeldung: 01. Juni 2009
Tag der Abgabe: 30. November 2009

Inhaltsverzeichnis

1. Einleitung

Die Diplomarbeit, die mit diesem Proposal beschrieben wird, soll Sicherheitsschwachstellen in objektorientierten Modulen untersuchen und modellieren. Die Resultate der Analyse sollen als Grundlage für ein Tool werden, das dem Programmierer bei dem Erkenntnis dieser Schwachstellen ihres Codes behilflich ist. In der *Einleitung* dieses Proposal werden die Hintergründe und Ziele der Diplomarbeit dargestellt. Weiter in der *Konzeption* werden die Grundlagen, Architekturmerkmale und Erwartungen aufgelistet. Die anschließende *Durchführung* listet unter andere organisatorische Details, gibt einen Überblick über die verwendete Werkzeuge und Richtlinien und zeit einen vorläufigen Zeitplan für die Diplomarbeit.

1.1 Motivation

Enge Interaktion zwischen Anwendungen ist längst keine Zukunftsvision mehr. Eigenschaften wie schnellerer Zugriff auf Dienste und schnellere Verarbeitung der Daten haben dieses Konzept zur Selbstverständlichkeit gemacht. Allerdings es bestehen Nachteile: Benutzerdaten, involvierte Software und Systeme können zum Ziel mancher Angriffe werden. Diese lassen sich vermeiden durch das Erkennen und Vermeidung von Vulnerabilitäten bereits während der Entwicklungsphase. Desweiteren erfordern steigende Produktivitätsbedürfnisse, dass Testen und Analyse kostengünstiger und deswegen automatisiert werden sollen.

1.2 Ziele und Aufgabenstellung

Ein Teil der Diplomarbeit beschäftigt sich mit der Erstellung eines Tools, die mittels statischer Analyse den Code des Programmierers untersucht und somit aufgedeckte Vulnerabilitäten anzeigt. Dieses Tool soll Programmierern das Auffinden und Verstehen der Vulnerabilitäten aus eigenem Code erleichtern durch eine übersichtliche Auflistung der Ergebnissen und weitere Informationen über dem Befund.

Der zweite Teil der Diplomarbeit besteht aus der Definition und Klassifikation der Vulnerabilitäten in OO-Modulen in Java. Dieses erfordert eine vorgehende Modellbildung, damit die Sicherheitskriterien in kompatible Formalismen ausgedruckt werden können. Eine Sicherheitsschwachstelle wird durch die Problemmuster identifizierbar, die sie verursacht.

1.3 Grundlagen

- Definition Vulnerabilität - Definition Statische Analyse - Bereits vorhandene Forschung - Vorstellung der Vulnerabilitätenkatalog - Bereits vorhandene Kataloge - SISSy (hier oder beim Entwicklungsumgebung?)

Unter „Vulnerabilität“ wird eine Schwäche in einer Rechenanlage verstanden. Diese kann ausgenutzt werden um unautorisierten Zugang zu Informationen zu bekommen oder kritische Bearbeitungen zu stören (die letzte bekannt unter den Namen „DoS“ - Denial of Services). Eine Vulnerabilität kann entweder ein Defekt sein oder eine gefährliche Funktion sein.

Durch Analyse des Quellcodes können solche Vulnerabilitäten entdeckt werden. In dies ist eine sehr forschungsintesives Bereich der Informatik. Es gibt bereits viele verschiedene Analysemethoden und Tools, die das Ziel haben, Vulnerabilitäten in verschiedene Programmiersprachen zu entdecken, allerdings kein Silver Bullet. Das Verständnis einer Vulnerabilität, die Methodik der Analyse, die verschieden Sprachen und die Entdeckung neuer Vulnerabilitäten machen es unmöglich. Die meist bekannten Tools sind zum Beispiel ESC/Java 2, FindBugs [HoPu04], JLint [JLin] und PMD [PMD]. Eine an der Universität Maryland durchgeführte Untersuchung [RuAF04], zeigt jedoch, dass keine von den Tools eine vollständige Abdeckung der Fehler erzielen. Das liegt unter anderen auch daran, dass sie unterschiedliche Ziele haben und unterschiedliche Elemente des Java-Codes untersuchen: FindBugs, JLint und PMD führen eine syntaktische Analyse durch, ESC/Java 2 dagegen eine Modellüberprüfung.

Schwächen in Java lassen sich in der JVM (Java Virtual Machine) und in den Anwendungscode finden. Die Diplomarbeit beschäftigt sich nur mit dem, die aus dem Anwendungscode stammen. Die Charakteristika von Java-Sprachkonstrukte die sich ausbeuten lassen sind zum Beispiel Typ-Sicherheit, `public` Felder, innere Klassen, Serialisierung, Reflektion. Weitere Vulnerabilitäten können bei Fehlersuche, Überwachung und Verwaltung entstehen, indem man additionale Software dafür nutzt. Eine Möglichkeit diese zu vermeiden, ist Code herzustellen, die sich an Richtlinien hält. Diese Richtlinien sollen dem Programmierer behilflich bei der Herstellung sicherer Code sein. SUN stellt solche bereit [Sun 07], MCGraw und Felten bezeichnen ihre Richtlinien als „die 12 Regeln“ für ein sicherer Java-Code. [McFe98].

Die Liste der erforschten Vulnerabilitäten befindet sich in dem technischen Report: „Java Component Vulnerabilities - an Experimental Classification targeted at the OSGI Platform“ [PaFr07]. (OSGi Platformen ermöglichen Verwaltung von mehrere Anwendungen über JVM, die dadurch die Möglichkeit haben, miteinander zu kommunizieren. Die On-the-Fly-Installation der OSGi-Komponente lassen neue Schwachstellen entstehen und momentan kein Mechanismus garantiert, dass der ausgeführte Code kein negatives Verhalten aufweist.) Die Darstellung der Vulnerabilitäten im Katalog ist semiformal und beinhaltet alle Informationen die zum Verständnis und Prävention und Korrektur nützlich sind. Beispiel: (Auszug aus dem Katalog)

D.5.10 Memory Load Injection

Vulnerability Reference

- **Vulnerability Name:** Memory Load Injection

- **Identifier:** Mb.java.10
- **Origin:** MOSGI Ares Research Project (OSGi Platform Monitoring)
- **Location of Exploit Code:** Application Code - Java API
- **Source:** Application Code (No Algorithm Safety - Java)
- **Target:** Platform
- **Consequence Type:** Performance Breakdown
- **Introduction Time:** Development
- **Exploit Time:** Execution

Vulnerability Description

- **Description:** A malicious bundle that consumes most of available memory (61,65 MB in the example)
- **Preconditions:** -
- **Attack Process:** Store a huge amount of data in a byte array
- Consequence Description: Only a limited memory space is available for the execution of programs
- **See Also:** Ramping Memory Load Injection, CPU Load Injection

Protection

- **Existing Mechanisms:** -
- **Enforcement Point:** -
- **Potential Mechanisms:** Code static Analysis ; Resource Control and Isolation - Memory
- **Attack Prevention:** -
- **Reaction:** -

Vulnerability Implementation

- **Code Reference**: Fr.inria.ares.memloadinjector-0.1.jar
- **OSGi** Profile: J2SE-1.5
- **Date:** 2006-08-24
- **Test Coverage:** 100

- **Known Vulnerable Platforms:** Felix; Equinox; Knopflerfish; Concierge; SFelix

Für die Abstraktion des Quellcodes, die für die Analyse notwendig ist, wird das Tool SISSy in Betracht genommen, die am FZI in Rahmen des Projekts QBench entstanden ist. Das Ziel des Projekts ist

> Entwicklung und Einsatz eines ganzheitlichen Ansatzes zur konstruktions- und evolutionsbegleitenden Sicherung der inneren Qualität von objektorientierter Software, um den Aufwand der Softwareentwicklung und -evolution (und damit Kosten) deutlich senken zu können.

[FZI].

-PQL

2. Konzeption

-Taxonomie

2.1 Formale Definition

Wie in dem letzten Kapitel erwähnt wurde, teil der Aufgabenstellung besteht darin, eine formale Darstellung der Vulnerabilitäten zu entwickeln, die eine genaue und eindeutige Beschreibung dieser gewährleistet, frei von subjektiver Interpretation. Damit sollen die Vulnerabilitäten aus dem bereits erwähnten Vulnerabilitätenkatalog darstellbar sein. Desweiteren soll die Darstellung leicht erweiterbar sein, um neu entdeckte Vulnerabilitäten aufgenommen zu werden.

Beispiel einer möglichen formalen Darstellung wäre:

Exposed Fields für Objektinstanzen

$Obj_{C_{A,i}} \overset{set}{\Rightarrow} Obj_{C_{B,j}}.var1$
$\Rightarrow$
$\exists Obj_{C_{C,k}} \neq Obj_{C_{A,i}} \mid Obj_{C_{C,k}}.Obj_{C_{B,j}}.var1 = Obj_{C_{A,i}}.Obj_{C_{B,j}}.var1$

Schreibweise: $Obj_{C_{A,i}}$: Objektinstanz i einer Klasse der Componente A
$Obj_{C_{A,i}}.var1$: var1 von $Obj_{C_{A,i}}$

Die Regel besagt: Eine „Exposed Fields“ Vulnerabilität ist vorhanden falls eine Objektinstanz i einer Klasse aus Komponente A eine Variable var1 der Instanz j einer Klasse aus Komponente B setzen darf und eine weitere Instanz einer Klasse aus einer Komponente C gibt, die die gleiche Variable setzen darf.

2.2 Analyse

Nach der Entwicklung der formelle Darstellung soll diese für die Analyse in einem von dem Tool interpretierbar Format dargestellt werden. Dafür eignen sich Textzeichen basierte Formate sehr gut: sie sind leicht einzulesen und auch von Benutzer lesbar. Das Beispiel weiterführen würde eine mögliche Darstellung so gestaltet:

␣-␣XML␣-␣Darstellung

Der nächste Schritt ist die Analyse des Codes. Dafür werden SISSy und PQL in Betracht genommen. Diese sollen den Code untersuchen und daraus Informationen abstrahieren, die bedeutend für die Analyse sind. Die Analyse besteht dann darin, die abstrahierte Informationen nach dem Muster in den Schritt zwei zu untersuchen. Die Befunde werden persistent gespeichert, damit sie jederzeit von dem Programmierer aufrufbar sein können.

2.3 Architektur

Eclipse ist eine der beliebtesten Open Source Entwicklungsplattform für Java, dessen Framework einfach mit weitere Werkzeuge zu erweitern ist. Daher wurde entschlossen, das Tool als Eclipse Plug-In zu entwickeln. Vorteile eines Eclipse Plug-In sind unter andere

- Einfache Installation und Bedienung,
- Erweiterbarkeit,
- Plattform-unabhängige Nutzung.

2.4 Erwartete Qualitätsmerkmale

Von der Benutzersicht soll das Tool zuverlässig die Vulnerabilitäten erkennen, ideal ohne False Positives oder False Negatives. Die Durchführung der Analyse soll nicht viel Zeit und Ressourcen in Anspruch nehmen und soll keine Unvollständigen Ergebnisse zeigen. Die Resultaten der Analyse und Hinweise sollen verständlich sein auch für Benutzer ohne allzu lange Programmiererfahrung [ChWe07]. Eine Dokumentation der Installation und Nutzung soll der Umgang mit dem Tool erleichtern. Von der Sicht der Entwickler soll das Tool modularisiert erstellt werden, so das spätere Erweiterungen leicht durchzuführen sind. Besonders die Liste der Vulnerabilitäten soll leicht aktualisiert werden.

2.5 Risiko

- SISSy - Darstellung nicht ausreichend für den Ganzen Katalog -> andere Tools zu Hilfe nehmen - Aufwand der Formalisierung, Transformation größer als vorgestellt -> Ein Teil der Katalog Transformieren - Krankheitsfall - Ausreichende Puffer in der Planung.

3. Durchführung

3.1 Organisatorisches

Erstgutachter: Prof. Dr. Ralf H. Reussner
Zweitgutacher:
Betreuer: Dr. Pierre Parrend

Die Diplomarbeit findet am Institut für Programmstrukturen und Datenorganisation (IPD), Lehrstuhl für Software Design und Qualität (SDQ), in Kollaboration mit Forschungszentrum Informatik (FZI).

3.2 Entwicklungsumgebung

Für die Erstellung der Implementierung wird Eclipse als Entwicklungsplattform und Java als Programmiersprache verwendet. Um eine übersichtliche Code-Erstellung während der Diplomarbeit zu gewährleisten wird das Maven Eclipse Plug-in eingesetzt. Zusammen mit CheckStyle und Javadoc soll dazu führen, die SEQ-Code-Konventionen [SEQ] einzuhalten.

Zur Hilfe bei der Erstellung der statische Analyse kommt SISSy zum Einsatz. Um die Ergebnisse zu speichern verwendet SISSy von der Open Source Datenbank PostgreSQL. Diese Datenbank wird auch genutzt, um die Resultate der Analyse zu speichern.

Für die Durchführung der Tests während des Code-Review wird JUnit eingesetzt.

Ausarbeitung, Präsentationen und weitere Artikel werden mit LaTeX erstellt. Dazu werden die Vorlagen von SEQ für Diplomarbeiten und Präsentationen genutzt. Die Ausarbeitung erfolgt auf Englisch. Laut Prüfungsordnung ist notwendig, dass die dazugehörige Zusammenfassung auf Deutsch geliefert wird.

3.3 Artefakte

Während der sechsmonatigen Zeit der Durchführung sollen folgende Artefakte entstehen:

- Literaturüberblick über bereits vorhandene verwandte Forschungsthemen, Tools, Verfahren und Konzepte,
- Software, die die erforschte Konzepte implementiert,
- JUnit Test für das zu erstellende Software,
- Dokumentation für die Installation und Nutzung des Tools,
- Entwickler-Dokumentation für die möglichen Weiterentwicklung des Tools
- Ausarbeitung des Forschungsthemas.

Desweiteren sind wissenschaftlichen Arbeiten und Veröffentlichungen erwünscht.

3.4 Zeitplan

Gemäß der Prüfungsordnung der Universität Karlsruhe (TH) für den Diplomstudiengang Informatik beträgt die Bearbeitungszeit einer Diplomarbeit an der Uni Karlsruhe von der Anmeldung bis zum Abschluss sechs Monate. Ein grober Zeitplan mit Meilensteinen ist in der folgenden Abbildung dargestellt. Dieses Diagramm stellt nur Rahmen der Diplomarbeit dar, im Laufe der Zeit soll der Ablauf verfeinert werden.

Während der Bearbeitungszeit finden regelmäßige Treffen zwischen der betreuten Studentin und Betreuer statt, mindestens einmal in der Woche. Desweiteren besucht der Prüfling die regelmäßigen Treffen der Diplomanden am SDQ und FZI.

Die wichtige Planungsmerkmale sind:

3.4.1 Ausarbeitungsphase

Die Ausarbeitungsphase dauert sechs Monate. In dieser Zeit werden alle Textdokumente (die unter Artefakte aufgelistet werden) geschrieben. Die Meilensteine dieser Phase sind:

- Gliederung und Literatur reichen - einen Monat nach dem Beginn der Diplomarbeit soll dieser Artefakt abgegeben werden.
- Abgabe der Rohfassung - Betreuer bekommt die Rohfassung. Änderungsvorschläge werden in der danach in der Ausarbeitung aufgenommen.

3.4.2 Formalisierung-Phase

In dieser Zeit wird die formale Darstellung der Vulnerabilitäten entwickelt. 25 Arbeitstagen sind dafür gedacht, weitere 5 als Puffer.

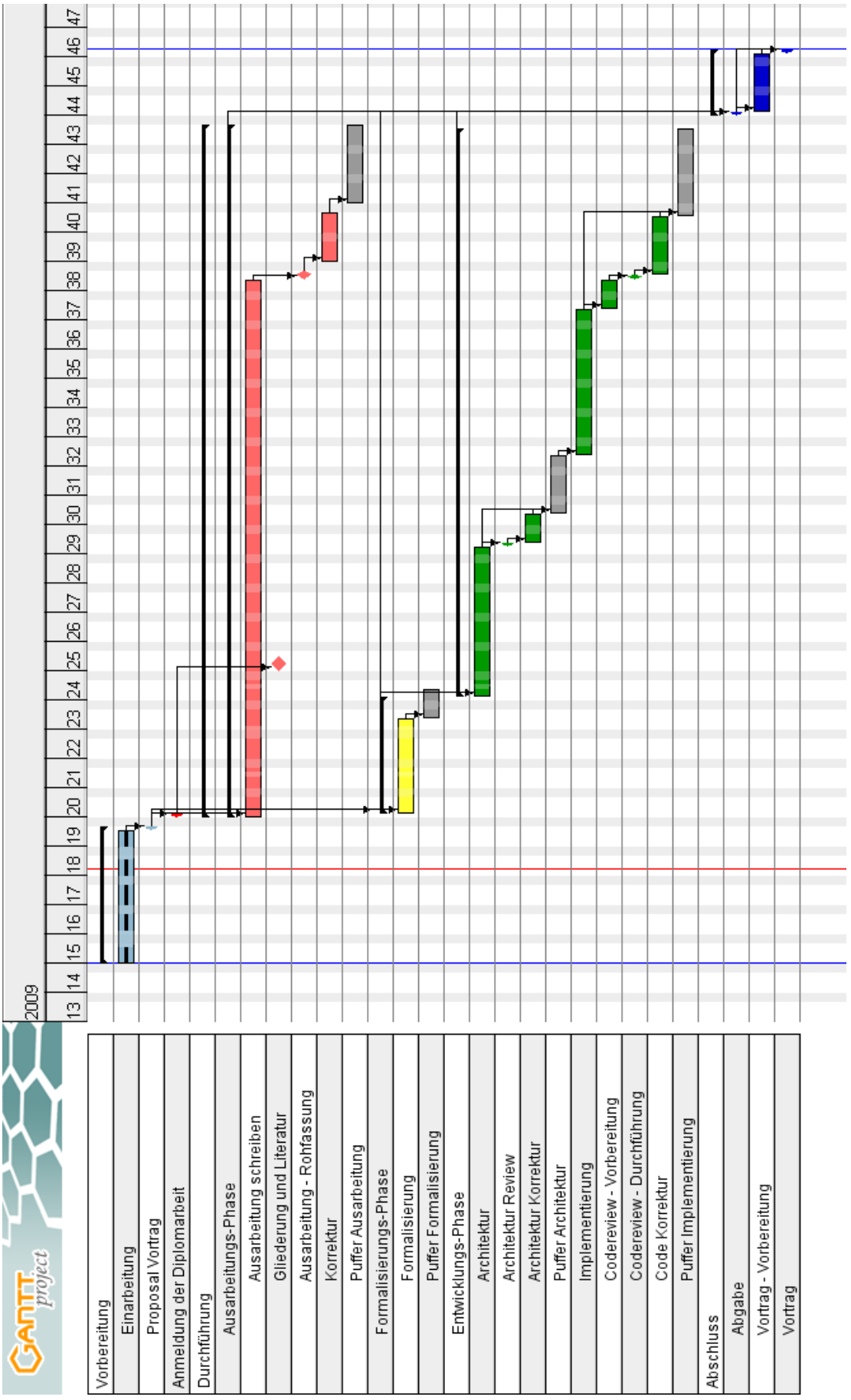

Abbildung 3.1: Zeitliche Planung der Diplomarbeit

3.4.3 Entwicklungsphase

Die Entwicklungsphase umfasst die Architektur des Tools als auch seine Implementierung, Testen und Korrektur. Die Architekturentwurf (25 Arbeitstage) wird von dem Betreuer abgenommen, bei Bedarf werden Änderungen durchgeführt. Steht der Entwurf fest, wird mit der Implementierung angefangen. In dieser Zeit wird der Code des Tool zusammen mit dem JUnit erstellt. Am Ende der Implementierungszeitraum (25 Arbeitstage), nach einer Woche Review-Vorbereitung (bei der die JUnit-Tests und Anleitung für Test vervollständigt werden, haben die Tester (ausgewählten Studenten und Mitarbeiter des Instituts) eine Woche Zeit, den Code zu Testen. Die Befunde des Testen und weitere Kritikpunkte werden im Review-Treffen dem Studenten vorgestellt und im Protokoll des Treffs festgehalten. Der nächsten Schritt beschäftigt sich mit der Korrektur der Codes.

3.4.4 Puffer

Für jede wichtige Phase wird ein Puffer von mindestens 30% einkalkuliert. Diese Puffer sorgen für eine punktliche Durchführung der Diplomarbeit, trotzt mancher Risikos aufgelistet in Kapitel „Konzeption“.

3.4.5 Abgabe

Dem Betreuer wird alle Artefakte vollständig abgegeben. Nach der Abgabe kann keine Korrekturen mehr unternommen werden. Nach zwei Wochen folgt die Präsentation der Diplomarbeit vor der Mitarbeiter und Studenten des Instituts.

Literatur

[BlGa05] Joshua Bloch und Neal Gafter. *Java Puzzlers - Traps, Pittfallls and Corner Cases*. Addison-Wesley. 1. Auflage, 2005.

[Chec] Checkstyle. Checkstyle Documentation on Sourceforge.net - http://checkstyle.sourceforge.net/index.html.

[ChWe07] Brian Chess und Jacob West. *Secure Programming with Static Analysis*. Addison Wesley. 2007.

[Crew97] Roger F. Crew. ASTLOG: A Language for Examining Abstrakt Sysntax Trees. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, Oktober 1997. sal.

[FLLN+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe und Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, New York, NY, USA, 2002. ACM, S. 234–245.

[FZI] FZI. QBench Poject - http://www.qbench.de/QBench/CMS/index.html.

[GoOA05] Simon F. Goldsmith, Robert O'Callahan und Alex Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, 2005. ACM, S. 385–402.

[HoPu04] David Hovemeyer und William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12), 2004, S. 92–106.

[JLin] JLint. Jlint - http://artho.com/jlint/.

[MaLL05] Michael Martin, Benjamin Livshits und Monica Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 2005.

[Mave] Maven. Apache Maven Project - http://maven.apache.org/.

[McFe98] Gary Mcgraw und Edward Felten. Twelve rules for developing more secure Java code. *JavaWorld.com*, January 1998.

[PaFr07] Pierre Parrend und Stéphane Frénot. Java components vulnerabilities - an experimental classification targeted at the OSGi platform. Research Report RR-6231, INRIA, 06 2007.

[PaFr08] Pierre Parrend und Stéphane Frénot. Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms. In *Conference on Component-based Software Engineering (CBSE'2008)*, Band 5282/2008 der *LNCS*, Karlsruhe, Germany, October 2008. Springer Berlin / Heidelberg.

[Parr09] Pierre Parrend. Enhancing Automated Detection of Vulnerabilities in Java Components. In *Forth International Conference on Availability, Reliability and Security (AReS 2009)*, Fukuoka, Japan, March 2009.

[PMD] PMD. PMD - http://pmd.sourceforge.net.

[RuAF04] Nick Rutar, Christian B. Almazan und Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. *Software Reliability Engineering, International Symposium on*, Band 0, 2004, S. 245–256.

[SEQ] SEQ. SDQ-Code-Konventionen - http://sdqweb.ipd.uka.de/wiki/Checkstyle#SDQ-Code-Konventionen.

[Sun 07] Inc Sun Microsystems. *Secure coding guidelines for the Java programming language, version 2.0.* Sun Microsystems Inc., 2007.