Software-Fehlerinjektion

Lena Feinbube, Daniel Richter, Sebastian Gerstenberg,
Patrick Siegler, Angelo Haller, Andreas Polze

Technische Berichte Nr. 109

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 109

Lena Feinbube | Daniel Richter | Sebastian Gerstenberg |
Patrick Siegler | Angelo Haller | Andreas Polze

Software-Fehlerinjektion

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2016
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Druck: docupoint GmbH Magdeburg

ISBN 978-3-86956-386-2

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URN urn:nbn:de:kobv:517-opus4-97435
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-97435

Inhaltsverzeichnis

Glossar

Ausfall (*"Failure"*) Die Abweichung der Systemausgabe von dem spezifizierten Dienst.

Ausnahme (*"Exception"*) Ein Mechanismus in Programmiersprachen, im Fall von Fehlerzuständen Informationen darüber weiterzugeben und den Kontrollfluss zu unterbrechen.

Bug (engl.) Eine Fehlerursache im Quelltext.

Debugging (engl.) Das Suchen und Entfernen von Fehlerursachen im Quelltext ("Bugs").

Fehlerinjektion (*"Fault Injection"*), auch: Fehlerinjektionstesten (*"Fault Injection Testing"*). Das Einfügen von Fehlerursachen und -zuständen in ein System, um dessen Fehlertoleranz zu testen.

Fehlermodell (*"Fault Model"*, *"Failure Cause Model"*) auch: Fehlerursachenmodell, Ausfallursachenmodell. Die Fehlerursachen und -zustände, die als möglich angenommen werden und von einem Programm toleriert werden sollen.

Fehlertoleranz (*"Fault Tolerance"*) Die Fähigkeit eines Computersystems, in der Präsenz von Fehlerursachen weiterhin funktionstüchtig zu bleiben.

Fehlerursache (*"Fault"*) Die Ursache für einen Fehlerzustand. In Software: Eine Abweichung des Programms von einem korrekten Programm.

Fehlerzustand (*"Error"*) Ein erkennbarer Teil des Systemzustands, der einen Ausfall verursachen kann.

Fuzzing (engl.) Das Fehlerinjektionstesten von Funktionsrufen und Schnittstellen mit zufällig generierten Eingabewerten.

Spezifikation Die formale Beschreibung des beabsichtigten Systemverhaltens.

Verlässlichkeit (*"Dependability"*) Die Eigenschaft eines Computersystems, seinen Dienst auf eine Weise zu liefern, auf die man sich verlassen kann.

1 Experimentelle Beurteilung der Verlässlichkeit von Software durch Fehlerinjektion

Die Verlässlichkeit von Software wird aufgrund deren Verbreitung und steigender Komplexität immer wichtiger. Ein Ansatz, die Verlässlichkeit, insbesondere die Fehlertoleranzmechanismen, eines gegebenen Softwaresystems zu ermitteln besteht darin, das laufende Programm künstlich erzeugten Fehlerzuständen und -ursachen auszusetzen. Dieser, als Fehlerinjektion *bezeichnete Ansatz, wird in diesem Kapitel motiviert und kurz vorgestellt.*

Software nimmt in unserem täglichen Leben einen immer höheren Stellenwert ein. Gleichzeitig steigt die Komplexität der Software, die in unternehmenskritischen Anwendungsfällen eingesetzt wird, rasant an. [1, 6, 37] Somit ist es immer bedeutsamer geworden, die Zuverlässigkeit von komplexen Softwaresystemen mit geeigneten Werkzeugen zu messen.

Die Zuverlässigkeit eines Softwaresystems wird nicht nur durch physikalisch unvermeidbare Hardware-Ausfälle bedroht, sondern auch durch Programmierfehler („Bugs“), menschliches Versagen und fehlerhafte Interaktion zwischen verschiedenen Softwarekomponenten. *Fehlertoleranz* beschreibt die Eigenschaft eines Softwaresystems, auch in der Gegenwart solcher Fehlerursachen seinen Dienst erfüllen zu können [38]. Damit ein komplexes Softwaresystem wirklich zuverlässig wird, muss Fehlertoleranz in jeder Software-Abstraktionsschicht berücksichtigt werden. Es gibt hierzu diverse Fehlertoleranz-Maßnahmen, die meist auf einer Form von Redundanz beruhen [21]. *Räumliche Redundanz* beschreibt hierbei die Verwendung mehrerer funktionsgleicher (Hardware-)Komponenten, um den Ausfall einer Untermenge dieser Komponenten verkraften zu können. *Zeitliche Redundanz*, oftmals in Software umgesetzt, beschreibt die mehrfache Ausführung einer Berechnung, um so fehlerhafte Ergebnisse in einer Untermenge der Ausführungen tolerieren zu können. Zudem gibt es, um Datenfehler maskieren zu können, *informationelle Redundanz*, womit fehlererkennende und -korrigierende Codes bezeichnet werden.

Zahlreiche Studien [13, 53, 55] haben gezeigt, dass es utopisch ist, anzunehmen, dass ein komplexes Softwaresystem in allen Fällen korrekt funktioniert. Selbst wenn die Hardware ausfallsicher wäre, müsste mit fortschreitender Komplexität von einer ansteigenden Anzahl Bugs ausgegangen werden. [32, 40]

Die Probleme, die bewältigt werden müssen, um korrekten Quelltext zu schreiben, beginnen schon mit der Frage, was „korrekt“ überhaupt bedeutet. Nur wenige Softwaresysteme wurden nach einer vollständig formalisierten Spezifikation entwickelt, da dies kostspielig und anspruchsvoll ist. Software mit lückenhafter oder

informeller Spezifikation bewegt sich häufig in der Grauzone zwischen „inkorrektem Verhalten“ und „unintuitivem, vom Nutzer selbst verschuldeten Verhalten“. Über einige Ausfallarten herrscht jedoch Konsens: Verhalten wie zum Beispiel Abstürze, Datenkorruption, Sicherheitslücken oder Endlosschleifen sollte in Programmen um jeden Preis vermieden werden. Aufgrund mangelnder formaler Spezifikation und dem damit verbundenen Aufwand ist es oft pragmatischer, nicht die Korrektheit eines Softwaresystems zu beweisen, sondern experimentell und zur Laufzeit zu versuchen, mögliche Fehlerursachen aufzudecken – wie etwa mit Fehlerinjektion.

Software-Fehlerinjektion bezeichnet das künstliche Einfügen von Fehlerursachen oder fehlerhaften Zuständen in ein laufendes Softwaresystem. Sie kann verwendet werden, wo formale Verifikationsmethoden an ihre Grenzen stoßen, da sie keine Vollständigkeit der Spezifikation voraussetzt. Dieser experimentelle Ansatz kann effizient und auf wenig intrusive Art und Weise umgesetzt werden.

Im einfachsten Falle gilt schon die Verwendung einer Schnittstelle mit unerwarteten, zufällig generierten Werten als Fehlerinjektion. Hierfür ist keinerlei Kenntnis des Fehlermodells, der Spezifikation, oder sonstiger Programminterna vonnöten. Gezieltere Fehlerinjektoren arbeiten auf Basis eines detailreicheren und umfangreicheren Fehlermodells und können Spezifikationen zurate ziehen, um effizienter vorzugehen oder eine gewisse Abdeckung zu garantieren. So macht das breite Spektrum existierender Fehlerinjektionsansätze es möglich, kontrolliert zwischen Abdeckung beziehungsweise verifiziertem Verlässlichkeitsgrad und Effizienz abzuwägen.

1.1 Software-Fehlerinjektion

Der Ansatz der Fehlerinjektion zielt darauf ab, Fehlertoleranzmaßnahmen zu testen. Dazu werden künstlich externe Fehlerursachen („faults“) und Fehlerzustände („errors“) in ein laufendes System „injiziert“ und beobachtet, wie sich das System unter dieser zusätzlichen Last verhält. Fehlerinjektions-Werkzeuge können dazu dienen, die Verlässlichkeit verschiedener Systeme oder Systemversionen zu vergleichen und etliche Forschungsfragen zu beantworten, wie beispielsweise:

- Wie hoch ist die Abdeckung von Fehlertoleranzmechanismen?
- Wie verändert sich das Verhalten der Software, wenn Fehlerursachen und Fehlerzustände auftreten? Welchen Einfluss hat dies auf Qualitätsmetriken wie Performanz, Ergebnisgenauigkeit oder Verfügbarkeit?
- Wie gut funktioniert die Fehlererkennung und Fehlerkorrektur zur Laufzeit?
- Gibt es in dem System „Single Points of Failure“ – Komponenten, dessen Ausfall den Ausfall des gesamten Systems nach sich zieht?

Fehlerinjektion kann als ergänzende Methodik zum herkömmlichen Softwaretesten angesehen werden. [3] Während Softwaretests überprüfen sollen, dass sich

die Software unter einer repräsentativen Arbeitslast korrekt verhält, kommt bei Fehlerinjektionstests eine „Fehlerlast“ hinzu. Die Grenze zwischen Testen und Fehlerinjektion ist oft fließend, wie man beispielsweise an den in Kapitel 3 diskutierten Fuzzing-Ansätzen sehen kann.

Wir haben im Rahmen eines Seminars verschiedene Werkzeuge zur Fehlerinjektion ausprobiert und miteinander verglichen. Der vorliegende Bericht stellt eine Zusammenfassung unserer Ergebnisse dar. Es handelt sich dabei um eine Sammlung verschiedener Fehlerinjektionswerkzeuge, die in ganz verschiedenen Anwendungsfällen und auf unterschiedlichen Abstraktionsebenen relevant sind. Jedes Werkzeug wurde praktisch angewendet und hinsichtlich verschiedener Dimensionen eingeordnet, die in den folgenden Abschnitten kurz aufgezeigt werden. Zweck dieses Berichts ist es, auf Basis eigener Erfahrungen, einen zwar unvollständigen und heterogenen, aber dennoch praxisnahen Überblick über Fehlerinjektionswerkzeuge zu bieten.

Grundsätzlich stellen sich bei der Fehlerinjektion stets zwei Fragen, damit die oben genannten Forschungsfragen angemessen beantwortet werden können. Zum einen muss entschieden werden, welches **Fehlermodell** als Grundlage verwendet werden soll, also welche Arten von Fehlerursachen und Fehlerzuständen in den Fehlerinjektionstests berücksichtigt werden sollen. Zum anderen sollten Fehlerinjektionen an relevanten Zielorten innerhalb des Programms stattfinden – **wo und wann** sollen Fehler injiziert werden, damit die Tests möglichst realistisch sind?

1.1.1 Fehlermodelle

Ein Fehlermodell gibt an, welche Arten von Ausfallursachen – externer oder interner Natur – als möglich angenommen werden. Es definiert somit die Fehlerlast, die ein Fehlerinjektor simulieren soll.

Welche Fehlermodelle für das Testen komplexer, moderner Softwaresysteme am besten geeignet sind, ist Gegenstand aktueller Forschung. [51, 57] Es gibt bereits eine Reihe bewährter Modelle und Schemata zur Klassifizierung von Softwarefehlerursachen und Softwareausfällen. [16] Diese beziehen häufig sowohl Fehlerursachen im Quelltext – also Bugs – wie auch Fehlerzustände – also beispielsweise ein zur Laufzeit auftretender Speicherüberlauf – mit ein.

Weiterhin gibt es eine enge Beziehung zwischen dem Fehler(ursachen)modell, dem Systemmodell, dem Zeitmodell und dem Konsistenzmodell bei der Datenspeicherung: Zunächst werden stets zahlreiche Annahmen über das System, in dem ein Programm ausgeführt wird, getroffen. Oft sind diese Annahmen nur implizit formuliert: Welche relevanten Umgebungsvariablen gibt es? An welchen Stellen interagieren Menschen oder andere Komponenten mit dem Programm? Drohen katastrophale Ausfälle aufgrund gemeinsamer Ursache wie beispielsweise Stromausfälle oder Naturkatastrophen – und sollen diese durch Fehlertoleranzmechanismen berücksichtigt werden? Müssen bösartige Angreifer gemäß einem Angriffsmodell angenommen werden?

Das Zeitmodell beeinflusst die Fehlerklassen, die darin Sinn ergeben oder überhaupt erkennbar sind: In asynchronen Systemen kann keine maximale Nachrichten-

übermittlungsdauer angenommen werden, was die Erkennbarkeit und Spezifikation von fehlerhaftem Zeitverhalten einschränkt. In synchronen Systemen hingegen fällt es leichter, die fristgerechte Ausführung von Berechnungen zu überprüfen.

Konsistenzmodelle definieren, welche Zustände bei nebenläufigen Aktualisierungen geteilter Daten erlaubt sind und welche nicht. Da strikte Konsistenz in verteilten Systemen nicht praktisch umsetzbar ist, muss mit zeitweise inkonsistenten Datenzuständen gerechnet werden, die nicht als fehlerhaft gelten können, weil sie im Rahmen eines abgeschwächten Konsistenzmodells zulässig sind.

Cristian [11] kategorisiert Ausfälle von Softwarekomponenten, die sich über Systemgrenzen hinweg als Fehlerursachen weiter propagieren in einer Inklusionshierarchie.

Das *Orthogonal Defect Classification (ODC)*-Schema [10] dient der Kategorisierung vielfältiger Probleme im Softwareentwicklungsprozess und ist daher auf einer etwas höheren Abstraktionsebene einzuordnen. Es ermöglicht die Beschreibung einer Fehlerursache, deren Symptomatik und Korrekturmöglichkeiten in hohem Detailgrad.

Darüber hinaus gibt es eine Vielzahl programmiersprachen-, paradigmen- und anwendungsspezifischer Fehlermodelle. [42, 45, 58]

1.1.2 Implementierungsstrategien

Das experimentelle Injizieren von Fehlerursachen oder Fehlerzuständen in ein laufendes Softwaresystem ist ein vielseitig einsetzbares Werkzeug zur Beurteilung der Fehlertoleranz komplexer Systeme. Führt ein injizierter Fehler zu einem Ausfall des Gesamtsystems, kann dies auf unzureichende Fehlertoleranzmechanismen hinweisen. Wie auch für andere Testansätze gilt hier, dass das Gesamtsystem nur dann als robust und ausfallsicher bezeichnet werden kann, wenn all seine Schichten bzw. Komponenten fehlertolerant sind.

Abhängig davon, auf welcher Abstraktionsschicht Fehlerinjektion durchgeführt werden soll, unterscheidet sich auch deren Implementierungsstrategie. Eine Charakterisierung solcher Implementierungsstrategien wurde in [19] präsentiert und soll hier kurz in einer adaptierten Form wiedergegeben werden:

Triggermechanismen Jedes Fehlerinjektionswerkzeug benötigt einen Auslöser- oder Triggermechanismus, der während der Programmausführung dafür sorgt, dass eine künstlich erzeugte Fehlerursache oder ein fehlerhafter Zustand in den normalen Programmablauf eingefügt wird. Dieser Mechanismus kann auf verschiedene Weisen umgesetzt werden:

- Zeitbasiert (time-based): Die Programmausführung wird zu festen Zeitintervallen zur Fehlerinjektion unterbrochen (Vgl. Kapitel 3);
- speicherortbasiert (location-based): Fehlerhafte Werte werden in vordefinierte Speicherorte geschrieben;

- ausführungsbasiert (execution-driven): Die Injektion erfolgt dynamisch in Abhängigkeit von dem Kontrollfluss.

Selbstverständlich haben diese Triggermechanismen jeweils Vor- und Nachteile. Während zeitbasierte Fehlerinjektion oft einfach und nichtintrusiv umsetzbar ist – etwa durch Zeitgeber, die periodisch dedizierte Unterbrechungsmechanismen auslösen – bevorzugt sie Ausführungspfade, in denen viel Zeit verbracht wird. Speicherortbasierte Fehlerinjektion ist für die Fehlerklasse aller Speicherfehler naheliegend, erschwert jedoch die Kontrolle darüber, wie die Fehlerlast verteilt ist. Ausführungsbasierte Fehlerinjektion schließlich ermöglicht zwar komplexe und realistische Fehlermodelle, doch der Kontrollfluss lässt sich bei „Black Box"-Anwendungen oftmals nur auf intrusive Art und Weise manipulieren.

Injektionszeitpunkt Zunächst unterscheiden wir zwischen *synchronen* Auslösern, wie beispielsweise von der Programmiersprache angebotenen Ausnahmebehandlungsmechanismen, und *asynchronen* Auslösern, wie beispielsweise Hardwareunterbrechungen. Wann genau findet die Fehlerinjektion statt? Hier gibt es eine Reihe von Ansätzen:

- Vor der Laufzeit, beispielsweise durch Quelltextmutation, bei der Fehlerursachen ergänzt werden (Vgl. Kapitel 5);
- während der Laufzeit, durch die Verwendung synchroner oder asynchroner Unterbrechungsmechanismen (Vgl. Kapitel 6);
- zur Ladezeit weiterer Komponenten, wie beispielsweise zum Zeitpunkt des Bindens externer Bibliotheken (Vgl. Kapitel 4).

Welcher Injektionszeitpunkt am sinnvollsten ist, hängt von dem getesteten System sowie dessen Fehlermodell ab.

Injektionsartefakte Des Weiteren stellt sich die Frage, welche Repräsentation eines Programms zu Fehlerinjektionszwecken modifiziert werden soll. Auch hier gibt es diverse Möglichkeiten:

- Maschinensprache/Binärdateien (Vgl. Kapitel 3, Kapitel 4);
- eine Zwischenrepräsentation des Quelltextes (Vgl. Kapitel 5);
- Quelltext (Quelltextmutationsansätze, zum Beispiel mit Hilfe von aspektorientierter Programmierung [33]).

Allgemein ist zu beachten, dass sich das Verhalten eines Softwaresystems unvermeidbar ändert, wenn man es unter einer Fehlerlast beobachtet. Um diese Veränderung minimal zu halten, sind möglichst wenig intrusive Ansätze erstrebenswert.

1.2 Ziel des Berichts

Fehlerinjektionsansätze, die darauf abzielen, komplexe Softwaresysteme zu testen, sind divers und können auf verschiedenste Arten und Weisen umgesetzt werden. In diesem Bericht wird in den folgenden Kapiteln eine Auswahl der im Seminar „Fehlerinjektion in Software“ diskutierten Ansätze näher betrachtet und beschrieben.

2 Fehlerinjektion auf Betriebssystemebene

Fehlertolerante Anwendungen setzen oft ein ausfallsicheres Betriebssystem voraus. Gleichzeitig handelt es sich bei dem Betriebssystem um ein Softwaresystem von zunehmend hoher Komplexität und somit auch Fehleranfälligkeit. Es ist somit von zentraler Bedeutung, Verlässlichkeitseigenschaften auf Betriebssystemebene zu gewährleisten und zu überprüfen.

Das Betriebssystem stellt eine verlässlichkeitskritische Softwareschicht dar, deren Ausfallsicherheit von vielen Seiten bedroht wird – wie etwa durch Hardwareausfälle, Treiber-Bugs, fehlerhafte oder auch bösartige Anwendungen. In dem Mars Rover „Pathfinder“ beispielsweise kam es 1997 aufgrund eines Falles von Prioritätsinversion zu wiederholten Neustarts und somit zu Nichtverfügbarkeit. Die missionskritische Software konnte jedoch vor Ort (auf dem Mars) repariert werden: Dank des in dem Echtzeitbetriebssystem *VxWorks* installierten C-Interpreters konnte nachträglich ein Programm ausgeführt werden, welches die Prioritätsinversion durch das Überschreiben einer globalen Variable entfernte. [27]

Microkernel-Betriebssysteme gelten als zuverlässiger als herkömmliche monolithische Betriebssysteme, da sie erstens aufgrund ihrer geringen Komplexität weniger fehleranfällig sind und zweitens geringere Propagierungsdistanzen fehlerhafter Zustände aufweisen. [56] Allerdings wird die Fehlerbehandlung auch zu großen Teilen in die Anwendungsschicht ausgelagert, was die Problematik lediglich verschiebt.

In äußerst kritischen Fällen werden die hohen Kosten formaler Verifikationsmethoden in Kauf genommen, um die Abwesenheit bestimmter Fehlerklassen im Betriebssystem zu beweisen. [34] Beispielsweise existiert mit *seL4* ein vollständig verifiziertes Microkernel-Betriebssystem. [35] Diese vollständige Verifikation hat circa elf Personenjahre in Anspruch genommen und muss wiederholt werden, sobald dem Betriebssystem neue Funktionalität hinzugefügt werden soll. Sie kommt daher in den meisten Anwendungsfällen nicht in Frage.

Eine alternative Möglichkeit, die Fehlertoleranz von Betriebssystemen an vielen Stellen zu testen, ist die in diesem Bericht diskutierte Fehlerinjektion. Dieses Kapitel diskutiert Fehlermodelle und Angriffspunkte für solche Tests.

Ein Modell, das Ausfälle von Betriebssystemfunktionalität nach ihrem Schweregrad und ihren Auswirkungen bewertet, ist die CRASH-Skala [36]:

Catastrophic – Katastrophal Das Betriebssystem stürzt ab. Dieser Fall soll für sicherheitskritische Anwendungsfälle in jedem Fall verhindert werden.

Restart – Notwendigkeit eines Neustarts Ein Prozess wird unresponsiv, sodass er neu gestartet werden muss.

Abort – Ungeplanter Abbruch eines Prozesses Ein einzelner Prozess stürzt beispielsweise aufgrund einer Schutzverletzung (engl. Segmentation Fault) ab.

Silent – Stummer Ausfall Es kommt zu der Verwendung falscher Werte, ohne dass diese durch einen Fehlercode oder eine Ausnahme erkenntlich wird.

Hindering – Hindernis Die korrekte Diagnose eines Problems wird durch mangelnde oder falsche Fehlermeldungen verhindert.

Die tatsächliche Ordnung des Schweregrads der oben genannten Ausfallklassen hängt von dem Anwendungsfall ab. Insbesondere stumme Ausfälle können je nach Anwendung als unkritisch oder auch sehr problematisch wahrgenommen werden, weil sie zu besonders schwierig vorhersehbarem, unerwartetem Verhalten führen.

Fehlerinjektion kann dazu dienen, Betriebssysteme im Hinblick auf die CRASH-Skala zu vergleichen und, falls notwendig, entsprechende Fehlertoleranzmaßnahmen zu ergreifen.

2.1 Historischer Überblick

Bereits 1969 wurde bei IBM simulierte Hardware-Fehlerinjektion verwendet, um die Datenintegrität von Schaltungen in der Entwurfsphase zu beurteilen. [5]

Wenig später, in den 1970er Jahren, wurden dann die theoretischen Grundlagen für die Fehlerinjektion in Computersystemen geschaffen und die Begrifflichkeit geprägt, vor allem von Avižienis, Laprie et al. [4]. Fehlertoleranz wurde zunächst vorwiegend auf Ebene der Hardware diskutiert und umgesetzt, sodass sich frühere Fehlerinjektoren vorwiegend an Hardware-Fehlermodellen orientierten.

Erst in den 1990er Jahren fassten Fehlerinjektions-Forscher auch die Software und vor allem das Betriebssystem ins Auge. 1991 erregte ein simples Werkzeug namens *crashme* [7] Aufsehen, welches Zufallsdaten als Prozedur ausführt und damit erstaunlich viele Ausfälle bei frühen Betriebssystemen (HP-UX, SUN 4, IBM RT, ...) auslöste. Somit ist *crashme* ein früher Vertreter sogenannter *Fuzzing*-Ansätze und wird in Abschnitt 2.3.1 genauer betrachtet.

FINE (1993) [30] ist eine frühe Fehlerinjektionsumgebung für UNIX-Systeme, in der vor allem die Latenzzeiten bei der Ausbreitung fehlerhafter Zustände (*error propagation*) gemessen wurden. *FERRARI* [29] (1995) ist ein flexibleres softwaregestütztes Rahmenwerk zur Injektion von Fehlerursachen und -quellen, welches ebenfalls unter anderem zum Testen der Betriebssystemschicht verwendet wurde.

In Kapitel 3 wird weiter auf Fehlerinjektionsansätze eingegangen, die auf zufälligem Testen der Betriebssystemschnittstellen basieren.

2.2 Zielorte für Fehlerinjektion

Abbildung 2.1 stellt mögliche Zielorte für Fehlerinjektion auf verschiedenen Abstraktionsschichten des Softwarestacks dar. Während es für jeden Zielort relevante Fehlerinjektions-Forschung gibt, konzentrieren wir uns in diesem auf Software fokussierten Bericht auf die Schnittstellen zwischen Applikationen und dem Betriebssystem: Hier wird beispielsweise das zufällige Testen (englisch *Fuzzing* [12]) von Systemrufen umgesetzt, auf das in den folgenden Abschnitten detaillierter eingegangen wird.

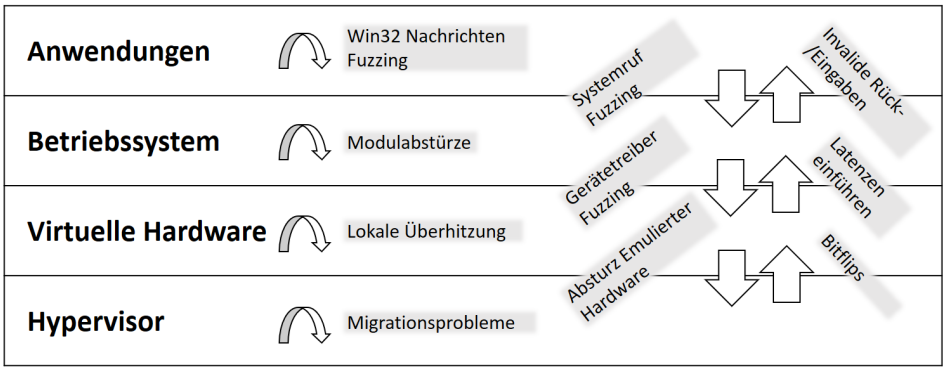

Abbildung 2.1: Mögliche Fehlerinjektionsziele auf verschiedenen Abstraktionsschichten. Dargestellt sind Beispiele für injizierbare Fehler innerhalb jeder Schicht sowie an den Schnittstellen zur nächsthöheren und -niedrigeren Schicht.

2.3 Fehlerinjektion durch Stresstests

Wie schon erwähnt, gibt es diverse Zielorte und damit verbunden auch verschiedenste Ansätze, um Fehlerinjektion auf der Betriebssystemschicht umzusetzen. Wir konzentrieren uns im Folgenden auf Fehlerursachen und -zustände, die von der Anwendungs- in die Betriebssystemschicht „hinein injiziert“ werden, also die Fehlertoleranz und Robustheit des Betriebssystems gegenüber sich falsch verhaltenden Anwendungen testen sollen.

2.3.1 Verlässlichkeitsexperimente mit Crashme

Unter *Fuzzing* versteht man den Zufallstest eines Softwaresystems durch generierte Eingabedaten, die den Eingaberaum ausreizen sollen, um so unwahrscheinliche oder selten ausgelöste Ausführungspfade zu testen.

Ein klassischer Stellvertreter des Fuzzing-Ansatzes zum Testen von Betriebssystemen ist das bereits erwähnte Werkzeug *crashme*. Es ruft zufällig generierte

Daten auf, als handle es sich um eine Prozedur. Ursprünglich für ältere Betriebssysteme entwickelt, wurde *crashme* von uns auch auf modernere Betriebssysteme angewendet:

Windows Server 2008, Windows 7 und Windows 8.1 Diese Betriebssysteme stürzten während der Fehlerinjektion mehrfach ab und wurden neu gestartet. Bei Windows 7 und Windows 8.1 konnten wir in einem Fall auch einen nicht länger responsiven Zustand beobachten.

Windows 10 Hier blockiert bereits das Installationsprogramm von crashme und macht damit Fehlerinjektion überflüssig.

Windows XP und Windows 9 Die Ausführung von crashme wird durch eine Visual-Studio-Abhängigkeit unterbunden. Nach Installation der richtigen Visual-Studio-Version läuft crashme, jedoch ohne Ausgaben oder Fehlermeldungen.

Weiterhin wurde auch in verwandten Arbeiten eine aktuelle Relevanz des scheinbar trivialen (Absturz-)Fehlermodells, das crashme zugrunde liegt, aufgezeigt.

Robustheit des Hypervisors gegen fehlerhaftes Anwendungsverhalten Mithilfe von crashme-Stresstests wurde 2007 überprüft, ob verschiedene *Virtual Machine Monitors* (VVMs) die erwartete Isolation und Sicherheit bieten. Obgleich crashme bereits ein altes und etabliertes Werkzeug war, fiel bei diesen Experimenten eine Reihe an Bugs in *QEMU*, *VMware*-Produkten und *Bochs* auf. Dabei handelt es sich teilweise um „triviale“ Fehlerursachen wie Vorzeichenfehler oder NULL-Zeiger-Dereferenzierungen. [46]

Robustheit von Anwendungen gegenüber einer fehlerhaften (OS-) Umgebung Unter Windows NT wurden Anwendungen mit grafischer Oberfläche auf zwei Weisen mit crashme getestet – durch zufällige Nutzereingaben, wie beispielsweise Mausereignisse, und durch das Versenden zufälliger Win32-Nachrichten. Die Autoren hatten zuvor ähnliche Experimente mit Linux-Systemen durchgeführt und festgestellt, dass viele Anwendungen solche Fehlerursachen nicht erwarten und nicht tolerieren. Ähnlich verhält es sich für Windows NT – es wurde ein erheblicher Verbesserungsbedarf der Fehlertoleranz identifiziert. [17]

3 Fehlerinjektion in Systemaufrufen

Das Zufallstesten (Fuzzing) von Systemaufrufschnittstellen ist ein beliebter Ansatz zur Fehlerinjektion in der Betriebssystemforschung. In diesem Kapitel wird auf das Programm Trinity *eingegangen, welches Fehler in Systemaufrufe des Linux-Betriebssytemkerns injiziert, um unerwartetes Verhalten aufzudecken und zu protokollieren mit dem Ziel, den Linux-Betriebssytemkern robuster zu machen.*

Wie die Sicherheitsforschung letzter Jahre wieder bestätigt hat [14], können bereits kleinste Fehler in Programmen verheerende Auswirkungen auf die Sicherheit und Stabilität eines Systems haben. Während viele der heutigen Betriebssysteme Prozesse voneinander isolieren, stellt sich die Frage, wie robust diese Mechanismen und gleichzeitig das Betriebssystem selbst sind.

Um dies zu testen ist einer der naheliegenden Orte die Schnittstelle zwischen Betriebssystemkern (*Kernel*) und Nutzerraum. Nutzerprozesse und teilweise auch Kernprozesse nutzen zur Kommunikation mit Komponenten des Betriebssystemkerns hierzu die vom Betriebssystem bereitgestellten Systemaufrufe. Beispielsweise handelt es sich in der Version 3.10 des Linux-Kerns hierbei bereits um mehr als 300 Systemaufrufe, von denen viele interne Strukturen des Kerns auslesen und verändern.[1] Sollten an dieser Stelle von Seite des Kerns notwendige Überprüfungen von Eingabewerten vergessen werden, so kann dies verheerende Auswirkungen auf die Stabilität des Gesamtsystems haben.

Um zu verhindern, dass Nutzerprozesse das Gesamtsystem stören können, muss die Schnittstelle zwischen Kernmodus und Nutzermodus getestet werden. Einerseits kann dies durch Modultests geschehen, welche überprüfen, ob sich eine Systemkomponente so verhält, wie es die Spezifikation vorgibt. Häufig werden jedoch die Fehlerfälle vergessen, bei denen Komponenten in einer untypischen, invaliden oder vom Entwickler nicht vorhergesehenen Weise genutzt werden.

Um solche Fälle ebenfalls abzudecken, wurden einige Programme entwickelt, die durch die Eingabe von zufälligen Daten (*Fuzzing*) den Kern einem Stresstest unterziehen sollen. Hierbei werden innerhalb des Eingaberaums Zufallswerte generiert und als Parameter für die Systemaufrufe gewählt.

[1] `https://github.com/torvalds/linux/blob/master/arch/m32r/kernel/syscall_table.S`, 5. Dezember 2016.

3.1 Systemaufruf-Fuzzing am Beispiel von Trinity

Trinity ist ein sogenannter „semi-intelligenter Systemaufruf-Fuzzer“ für Linux. Semi-intelligent bedeutet hier, dass die Aufrufe und generierten Parameter für die Systemaufrufe nicht komplett zufällig sind, sondern gewissen Mustern entsprechen müssen:

- Systemaufrufe bekommen den erwarteten Datentyp übergeben;
- Systemaufrufe mit Einschränkungen des Datenraums bekommen mit hoher Wahrscheinlichkeit Argumente innerhalb dieses Raums übergeben;
- bereits verwendete Argumente werden zwischengespeichert, um die Performanz folgender Ausführungen zu verbessern.

Die Entwicklung von Trinity begann 2006, primär geleitet von Dave Jones. [25] Popularität in der Entwicklung und im Einsatz gewann das Projekt jedoch erst im Jahre 2010. [26] Das Programm ist kostenlos erhältlich und quelloffen (GPLv2). [25] Trinity wird derzeitig aktiv von mehreren Entwicklern genutzt und ebenfalls auf einer Serverfarm von Intel ausgeführt [26], um Fehlerursachen (Bugs) in Kernkomponenten zu finden. Alleine im Jahre 2012 wurden Trinity mehr als 150 Bug-Funde zugeschrieben. [31]

Wie bereits erwähnt arbeitet Trinity nicht vollständig zufällig. Vielmehr verfügt Trinity über Wissen über gewisse Interna der Systemaufrufe. Beispielsweise kennt Trinity das Konzept von Puffern und Puffergrößen, Dateideskriptoren, Speicheradressen und Prozessnummern.

Annotationen Gerade bei der C-Schnittstelle von Linux-Systemaufrufen besteht das Problem, dass der Typ und Argumentenwertebereich nicht unmittelbar aus der Schnittstelle hervorgeht, sondern stark kontextabhängig ist. Beispielsweise erwartet der Systemaufruf *ioctl* als drittes Argument „either an integer value, possibly unsigned (going to the driver) or a pointer to data (either going to the driver, coming back from the driver, or both)“[2] – also einen Ganzzahlwert, möglicherweise vorzeichenlos, oder einen Zeiger auf die Daten (der entweder zu dem Treiber oder von dem Treiber führt oder beides). Es ist beim automatisierten Testen nicht offensichtlich, wie man solche Schnittstellen mit zufällig generierten, aber dennoch semantisch sinnvollen Argumenten aufruft.

Viele Systemaufrufe erwarten Zeiger auf Speicheradressen, die sich in dem tatsächlich verfügbaren Adressraum befinden. Rein zufällig generierte Zeiger würden in den meisten Fällen auf invalide Speicherorte zeigen und immer wieder die gleiche „uninteressante“ Fehlermeldung auslösen. Auch hier ist eine intelligentere Generierung von Zeigern durch Trinity erforderlich.

Auf ähnliche Weise würden viele naive Zufallsargumente bereits an den initialen Parameterüberprüfungen scheitern und spätere Programmpfade, die interessantere

[2] `http://man7.org/linux/man-pages/man2/ioctl.2.html`, 5. Dezember 2016.

Bugs enthalten könnten, nicht abdecken. Die Information, welche Parametertypen und -wertebereiche Systemaufrufe erwarten, wird daher in sogenannten Annotationen festgehalten. Diese können ebenfalls eigene Funktionen enthalten, um spezielleren Anforderungen an gewisse Parameter gerecht zu werden. Listing 3.1 zeigt die Annotationsstruktur für den Systemaufruf `madvise`[3].

Listing 3.1: Annotationsstruktur für Trinity

```
struct syscall syscall_madvise = {
        .name = "madvise",
        .num_args = 3,
        .arg1name = "start",
        .arg1type = ARG_NON_NULL_ADDRESS,
        .arg2name = "len_in",
        .arg2type = ARG_LEN,
        .arg3name = "advice",
        .arg3type = ARG_OP,
        .arg3list = {
            .num = 12,
            .values = { MADV_NORMAL, MADV_RANDOM, MADV_SEQUENTIAL,
                MADV_WILLNEED,
                    MADV_DONTNEED, MADV_REMOVE, MADV_DONTFORK,
                        MADV_DOFORK,
                    MADV_MERGEABLE, MADV_UNMERGEABLE, MADV_HUGEPAGE,
                        MADV_NOHUGEPAGE },
        },
        ...
    };
```

Der Systemruf hat die Signatur `int madvise(void *addr, size_t length, int advice);`. Man sieht beispielsweise, dass es sich bei dem ersten Argument um eine gültige Adresse handelt und welche Enumerationswerte für das dritte Argument möglich sind.

Fuzzing-Algorithmus Bevor Trinity beginnt Systemaufrufe zu „fuzzen“ – also mit zufälligen Daten zu speisen – sammelt der Hauptprozess des Programms eine Anzahl valider Dateideskriptoren und Prozessnummern, um diese später zu nutzen.

Das Testen selbst erfolgt parallel in mehreren Kindprozessen, die mit einem geteilten Speicherbereich (mittels *mmap*) arbeiten. Es gibt zudem einen überprüfenden *watchdog*-Prozess, der die Integrität geteilter Speicherregionen überwacht und korrigiert, falls eine Korruption durch die Fehlerinjektion in einem Kindprozess stattgefunden haben sollte. Zudem ist der *watchdog*-Prozess für das Terminieren unresponsiver Kindprozesse verantwortlich.

Die generierten Eingaben von Trinity, die zum Testen verwendet werden, werden pro Kindprozess in separaten Dateien gespeichert, bevor der Systemaufruf erfolgt.

[3] `http://linux.die.net/man/2/madvise`, 5. Dezember 2016.

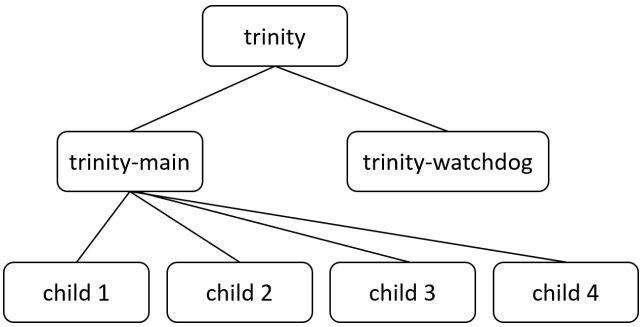

Abbildung 3.1: Prozessstruktur beim Fuzzing mit Trinity

Somit kann im Falle einer Kernel Panic nachvollzogen werden, welche Eingabe dazu führte. Die Resultate des Fuzzings werden zusammen mit der Prozess-PID, einer Testsequenznummer und den verwendeten Argumenten mitgeschrieben:

```
[17913] [2] madvise(start=0x7f59dff7b000,len_in=3505,advice=10) = 0
```

Häufig wird Trinity auf diese Weise betrieben, bis es zu Kernel Panic führt. Anschließend werden die Log-Dateien verwendet, um die möglichen Ursachen für diesen Ausfall nachzuvollziehen und bessere Fehlertoleranzmechanismen umzusetzen. Alternativ dazu kann Trinity auch mit einer Menge an Dateien – sogenannten *victim files* (Opferdateien) – als Argument benutzt werden. Diese werden geöffnet und an Dateisystem-relevante Systemaufrufe übergeben, um Dateisystem-spezifische Ausfälle zu entdecken.

3.2 Weitere Fuzzing-Werkzeuge für Fehlerinjektion in Systemaufrufen

Trinity ist ein repräsentatives Beispiel für die Injektion eines breit gefächerten Software-Fehlermodells in Systemaufrufe hinein. Das Betriebssystem war, wie erwähnt, schon immer besonders im Fokus der Fehlertoleranzforschung da erhöhte Robustheits- und Sicherheitsanforderungen gelten. Es gibt daher eine Menge weiterer Fuzzer für Systemaufrufe mit unterschiedlichem Funktionsumfang und Zielplattformen, die hier kurz Erwähnung finden sollen.

Das Werkzeug Eris [18] baut auf Trinity auf, indem es kombinatorisches Testen ermöglicht. Dieser Ansatz basiert auf einer formalen Modellierung des gesamten Eingaberaums, der dann systematisch getestet werden kann. Eine kombinatorische Abdeckung bedeutet dann, dass jedes t-Tupel denkbarer Eingabeparameterklassen getestet wurde. Die Funktionsweise von Eris wurde prototypisch an einem Systemaufruf validiert, allerdings bleibt eine detaillierte Auswertung der Komplexität unseres Wissens nach bislang aus. Neben Trinity gibt es eine Reihe Fuzzing-Werkzeuge, die dieses Problem angehen:

Der verteilte Fuzzer *syzkaller*[4] verwendet die Instrumentierung von Speicherzugriffen, um die Testabdeckung zu ermitteln und zu optimieren. Es wird daher eine Neukompilation des Kerns erforderlich. Das verteilte Fuzzing selbst findet zur besseren Isolation in virtuellen Maschinen statt, die ihre Testergebnisse sowie Informationen zur Testabdeckung via *Remote Procedure Call* (RPC) an den zentralen Testverwalter schicken. Syzkaller wurde als neuere Alternative zu Trinity entwickelt und hat bereits über 150 Bugs aufdecken können.[5]

iknowthis[6] ist ein alternatives Fuzzing-Werkzeug für Linuxsysteme, das seit 2010 vorwiegend von Tavis Ormandy entwickelt wird. Es unterstützt, ähnlich wie Trinity, die Annotation von Argumenten für eine höhere Abdeckung und zielgerichtete Injektion. iknowthis wird als Fuzzing-Framework bezeichnet – zu Recht, denn es erfordert zusätzlichen Implementierungsaufwand, wie bei der Annotation oder der Verwaltung zum Testen verwendeter Ressourcen.

sysfuzz[7] ist ein Fuzzing-Werkzeug für FreeBSD-Systeme, das in Funktionsumfang und -weise Trinity ähnelt. Es wurde erfolgreich in der Fehlerursachensuche von FreeBSD verwendet. Für Windows-Systeme gibt es das rein zufallsgesteuerte Werkzeug *iofuzz*[8], welches I/O Schnittstellen mit Zufallswerten aufruft.

3.3 Diskussion

Beim Testen von Systemaufrufschnittstellen – nicht nur in Linux-Systemen – besteht grundsätzlich das Problem, dass Entwickler bei handgeschriebenen Testfällen einseitig dazu neigen, das von ihnen erwartete Programmverhalten zu bestätigen. Unerwartete Eingaben werden naturgemäß nicht antizipiert und durch solche Testfälle daher mangelhaft abgedeckt.

Wie bereits erwähnt, wird Trinity erfolgreich zum Auffinden von Fehlerursachen bei der Entwicklung des Linuxkerns eingesetzt und hat schon zahlreiche Bugs[9] aufgedeckt. Einer der ältesten gefundenen Bugs war seit dem Jahre 1996 unentdeckt geblieben.[10]

Wir haben Trinity verwendet und waren in der Lage, in kurzer Zeit – wenige Sekunden – knappe 50.000 Systemaufrufe zu testen, von denen etwa 30.000 zu Ausfällen führten.

Die Weiterentwicklung von Trinity wurde stets vor allem von Dave Jones in dessen Freizeit vorangetrieben. Dieser äußerte vermehrt seinen Unmut darüber, dass die quelloffene Software vielseitig verwendet werde, jedoch selten Quelltextbei-

[4] `https://github.com/google/syzkaller`, 5. Dezember 2016.
[5] `https://github.com/google/syzkaller/wiki/Found-Bugs`, 5. Dezember 2016.
[6] `https://github.com/rgbkrk/iknowthis`, 5. Dezember 2016.
[7] `https://github.com/markjdb/sysfuzz`, 5. Dezember 2016.
[8] `https://github.com/debasishm89/iofuzz`, 5. Dezember 2016.
[9] `http://codemonkey.org.uk/projects/trinity/bugs-found.php`, 5. Dezember 2016.
[10] `http://codemonkey.org.uk/2012/09/25/trinity-finds-ancientr-bug/`, 5. Dezember 2016.

träge zurückgegeben würden. Es habe nennenswerte Beiträge lediglich von sechs Entwicklern gegeben, obwohl mit der Software vielerorts Geld verdient würde.

Nachdem bekannt wurde, dass unter anderem die umstrittene IT-Überwachungsfirma *Hacking Team* Trinity angepasst und auf Android-Geräten verwendet hatte, stellte Jones seine Entwicklungstätigkeit an dem Werkzeug im Juli 2015 mit der erzürnten Aussage „I'm done enabling assholes" ein. [11]

[11] `http://codemonkey.org.uk/2015/07/12/future-trinity/`, 5. Dezember 2016.

4 Fehlerinjektion an Bibliotheksschnittstellen

Die Verlässlichkeit von Anwendungen wird auch durch das Fehlverhalten von externen Bibliotheken bedroht, die oftmals von anderen Parteien entwickelt wurden und deren Interna unbekannt sind. Es ist daher sinnvoll, die Toleranz gegenüber Fehlerursachen in externen Bibliotheken experimentell zu testen. In diesem Kapitel stellen wir exemplarisch einige derartige Ansätze vor.

In der modernen Softwareentwicklung wird zunehmend modularisiert und auf extern entwickelten Quelltext zurückgegriffen. Dies erhöht die Produktivität und Standardisierung. Allerdings geht mit der Verwendung externer Bibliotheken auch ein erhöhtes Verlässlichkeitsrisiko einher: Häufig sind solche Bibliotheken nicht quelloffen, lückenhaft dokumentiert, oder sie weisen unerwartetes bis hin zu fehlerhaftes Verhalten auf. Oft haben verschiedene Versionen einer Bibliothek, die den gleichen Zweck erfüllen, unterschiedliche Verlässlichkeitseigenschaften – beispielsweise in der Form unterschiedlicher verteilter Konsistenzmodelle. Diese Eigenschaften sind Anwendungsentwicklern oft nicht bewusst und unter Umständen auch mangelhaft dokumentiert.

Fehlerinjektion an den Schnittstellen zu externen Bibliotheken lohnt sich, weil damit die Toleranz der Anwendung gegenüber inkonsistentem, fehlerhaftem oder unkonventionellem Verhalten der Bibliothek getestet werden kann, ohne dass die Interna der Bibliothek bekannt sein müssen. In diesem Kapitel diskutieren wir einige Ansätze, die solche Fehlerinjektionswerkzeuge verfolgen.

4.1 Fehlermodelle für Schnittstellen

Das als „Schalenmodell“ bekannte Fehlermodell nach Cristian [11] ist zwar ursprünglich für verteilte Systeme konzipiert. Es erlaubt jedoch auch die Charakterisierung von Fehlverhalten während der Interaktionen zwischen verschiedenen, voneinander abgekapselten Softwarekomponenten, wie einem Programm und den darin verwendeten externen Bibliotheken. Denkbar wären auf einer hohen Abstraktionsebene also folgende Fehlerklassen, die jeweils ineinander enthalten sind:

Absturz (Crash) Die Bibliothek stürzt bei dem Aufruf einer Funktion ab.

Fehlende Rückgabe (Omission) – Das Ergebnis eines Funktionsrufes wird nicht rechtzeitig (bezüglich einer Deadline) zurückgegeben.

Falsches Zeitverhalten (Timing) Das Ergebnis eines Funktionsrufes wird außerhalb des vorgesehenen Zeitfensters, oder nie, zurückgegeben.

Berechnungsfehler (Computation) Die Bibliothek führt eine fehlerhafte Berechnung durch.

Byzantinischer Fehler Ein beliebiges Fehlverhalten ist zu beobachten, beispielsweise auch inkonsistente Rückgabe verschiedener Ergebnisse, Korrumpierung der Ausführungsumgebung etc.

Die Schwierigkeit, Fehlerursachen zu tolerieren, steigt mit der Größe der Fehlerklasse an. Während Abstürze und Deadline-Verletzungen seitens einer Anwendung noch gut erkennbar sind, bedarf es im Fall von Berechnungsfehlern bereits detaillierten Wissens über den Algorithmus.

Perry et al. [49] haben Probleme an Softwareschnittstellen kategorisiert und bestehende Systeme im Hinblick darauf analysiert. Es gibt demnach 16 detaillierte Fehlerklassen, die anhand der Phase, in welcher das Problem auftritt – beispielsweise „Fehler zur Konstruktionszeit der Schnittstelle“ versus „Zeitprobleme“, oder anhand der getroffenen Annahmen – beispielsweise „Verletzung von Dateneinschränkungen“ – definiert sind.

In Unterabschnitt 4.2.2 wird beschrieben, wie ein weiteres denkbares Fehlermodell für Schnittstellen in den Fehlerinjektor Hovac integriert wurde.

4.2 Fehlerinjektionsansätze

Um an Schnittstellen Fehlerinjektion zu betreiben, benötigt man einen Mechanismus, der die Kommunikation abfängt und manipuliert. Grundsätzlich gibt es viele Möglichkeiten, das Abfangen und Manipulieren von Schnittstellenkommunikation umzusetzen: Man kann den Quelltext modifizieren, beispielsweise mit Hilfe von aspektorientierter Programmierung [33]. Instrumentierung ist automatisiert auch auf einer Zwischenrepräsentation bei der Kompilierung möglich, wie beispielsweise in dem LLVM-Framework [39]. Schlussendlich gibt es Ansätze, die lediglich auf den Binärdateien und zur Laufzeit arbeiten und im Kontext von Funktionsrufen in Bibliotheken hinein als *Hooking* bezeichnet werden. Wir konzentrieren uns hier auf solche Ansätze – da dafür kein Zugriff auf den Quelltext erforderlich ist, sind sie für größere Anwendungen am realistischsten.

4.2.1 Library Fault Injector

Das Ziel bei der Entwicklung von Library Fault Injector (LFI) war es, die Verwendung von Fehlerinjektion in Vielzwecksoftwaresystemen zu vereinfachen, die nicht extrem sicherheitskritisch sind und daher kein großes Zeit- und Entwicklungsbudget auf erschöpfende Fehlerinjektionstests verwenden können. Das Fehlermodell besteht auch hier aus Fehlerursachen, die ihren Ursprung in externen Bibliotheken

haben. Typische Windows- und Linux-Systeme weisen eine Vielzahl solcher Bibliotheken auf, sodass zunächst eine relevante Untermenge solcher Bibliotheksrufe für die Fehlerinjektionstests identifiziert werden muss.

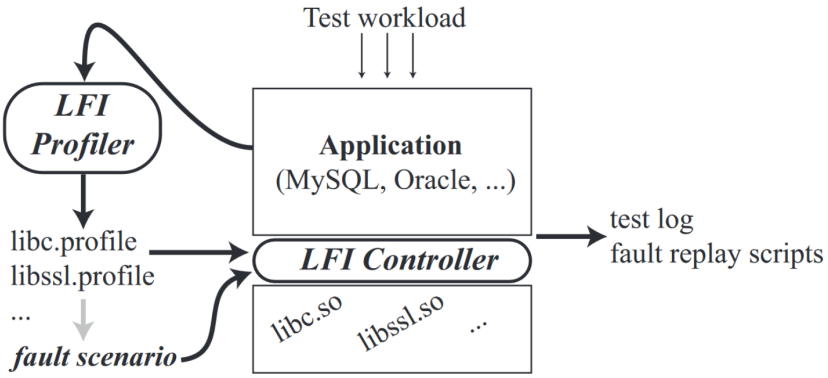

Abbildung 4.1: Funktionsweise des LFI-Fehlerinjektors (aus [41])

Abbildung 4.1 zeigt die Funktionsweise des LFI von Marinescu et al. [41]: Zunächst erstellt das Werkzeug mittels Binäranalyse der Anwendung ein Profil der externen Funktionsaufrufe und der Rückgabewertbehandlung. Hierzu werden die externen Bibliotheken disassembliert. Der Kontrollflussgraph wird dabei auf mögliche Rückgabewerte hin analysiert, von denen viele als Konstanten definiert sind. Weiterhin werden Seiteneffekte aufgefunden, die zur Kommunikation von Fehlerzuständen verwendet werden, wie beispielsweise Variablen im *Thread-local Storage* (TLS, ausführungsfadenlokaler Speicher). Das Resultat der Analyse ist ein „Fehlerursachen-Profil“ – ein XML-Dokument, in dem für die exportierten Funktionsrufe einer Bibliothek die möglichen Rückgabewerte und Seiteneffekte dokumentiert sind.

Basierend auf dem Fehlerursachen-Profil werden von LFI anschließend Fehlerinjektionskampagnen durchgeführt. Hierbei sind die Trigger-Intervalle ebenfalls über ein XML-Dokument konfigurierbar. Standardmäßig werden erschöpfende und zufällige Fehlerinjektionsszenarien angeboten. Zur Injektion wird eine Stub-Bibliothek generiert, in welcher die Trigger-Bedingung überprüft wird. Falls diese wahr ist, wird einer der bei der Analyse identifizierten Rückgabewerte im Fehlerfall zurückgegeben; andernfalls wird die Originalfunktion gerufen. Die Stub-Bibliothek wird mit Hilfe der *LD_PRELOAD*-Umgebungsvariable, die unter Linux zur Verfügung steht, vor der Originalbibliothek geladen.

Evaluierung Die Autoren von [41] haben mit LFI Fallstudien an *Pidgin*, einem Sofortnachrichtenprogramm, und *MySQL*, einem populären Datenbankmanagementsystem, durchgeführt. Sie konnten in kurzer Zeit einen Bug in Pidgin finden und Lücken in der Testsammlung von MySQL aufdecken. Die Laufzeit der initialen Profilerstellung hängt wie erwartet von der Anzahl der Instruktionen in der Binär-

datei ab. Für eine größere Bibliothek wie *libxml*2 beträgt sie bis zu 20 Sekunden – um alle Bibliotheken in einem typischen Linux-System zu analysieren, bedarf es also mehrerer Stunden. Allerdings ist dies ein einmaliger Aufwand. Die tatsächliche Fehlerinjektion mittels *LD_PRELOAD* ist leichtgewichtig und performant.

Insgesamt ist LFI ein leichtgewichtiges, einfach benutzbares Fehlerinjektionswerkzeug mit dem schnell die Fehlertoleranz gegenüber unerwarteten Rückgabewerten verbessert werden kann. Es ist allerdings auf das Fehlermodell unerwarteter Rückgabewerte eingeschränkt.

4.2.2 Hovac

Hovac ist ebenfalls ein Fehlerinjektor, der an den Schnittstellen zwischen einer Anwendung und ihren externen Bibliotheken arbeitet.

Ähnlich zu LFI lassen sich bei Hovac die Orte für Fehlerinjektion über XML-Dateien konfigurieren. Da es sich bei den injizierten Fehlerursachen jedoch nicht nur um fehlerhafte Rückgabewerte, sondern auch um Manipulationen des Speichers und der Umgebung sowie des Zeitverhaltens handelt, werden die Fehlerinjektionsorte nicht wie bei LFI automatisch aus Binäranalyse ermittelt, sondern müssen vom Anwender vorgegeben werden.

Der Fokus bei der Konzipierung von Hovac lag auf einem weitreichenden, realistischen Fehlermodell. Die *Common Weakness Enumeration*-Datenbank (CWE), die auf Softwareproblemen basiert, die in der Industriepraxis beobachtet wurden, bietet eine gute Grundlage für solch ein Fehlermodell. Basierend auf der Datenbank wurden für Hovac sechs Fehlerklassen, die an Bibliotheksschnittstellen implementierbar sind, abgeleitet:

Berechnungsfehler Die Rückgabewerte einer Funktion weichen von der Erwartung ab.

Umgebungsfehler Der Zustand der Ausführungsumgebung oder des Betriebssystems weicht von der Erwartung ab.

Verzögerung Ein Funktionsruf dauert länger als erwartet.

Kritischer Wettlauf (engl. Race Condition) Die Zugriffsreihenfolge auf geteilte Speicherbereiche beeinflusst das Endergebnis.

Speicher Der für die Anwendung sichtbare Speicher befindet sich in einem unerwarteten Zustand.

Kontrollfluss Es gibt Abweichungen von dem erwarteten Ausführungspfad, beispielsweise durch Ausnahmebehandlung.

Eine Hovac-Konfigurationsdatei, in der definiert ist, dass in einer Funktion sowohl fehlerhaftes Zeitverhalten als auch Berechnungsfehler injiziert werden sollen, kann wie folgt aussehen:

Listing 4.1: Konfigurationsdatei zur Fehlerinjektion mit Hovac

```
<faulttarget>
  <signature>
  unsigned int
  calculateMandelbrotIterations
    (std::complex c, unsigned int maxiter);
  </signature>
  <callingconvention>__cdecl</callingconvention>
  <timing>true</timing>
  <computation>true</computation>
</faulttarget>
```

Hovac verwendet – ähnlich zu LFI – die Technik des *DLL-API-Hookings*, um vor oder nach dem Ruf dynamisch geladener Bibliotheken zusätzlichen Fehlerinjektionscode auszuführen. Die Implementierung erfolgte hier anders als bei LFI nicht für Linux, sondern für Windows. Dort lässt sich die Umleitung von Funktionsrufen in eine eigene Bibliothek nicht so leicht mit einer Umgebungsvariable wie *LD_PRELOAD* implementieren. Stattdessen wurde das von Microsoft Research entwickelte Werkzeug *Detours* [22] verwendet. Basierend auf den Schnittstellen, an denen Fehlerursachen injiziert werden sollen, wird eine Zwischenbibliothek generiert. Mit Hilfe von Detours werden Funktionsrufe dynamisch in diese Zwischenbibliothek umgelenkt, die intern wiederum die Originalbibliothek aufruft.

Evaluierung Für Hovac wurde ein teils automatisierbares, teils manuelles Vorgehen zur reproduzierbaren Verlässlichkeitsevaluation vorgeschlagen. Es sieht drei Schritte vor:

1. **Orte für Fehlerinjektion finden**: Mit Hilfe herkömmlichen Profilings werden die am häufigsten ausgeführten Teile des Programms identifiziert, beispielsweise mit Werkzeugen wie *NTrace* [48]. Falls verfügbar, können Dokumentation und Quelltextanalyse zu Rate gezogen werden, um die Injektionsorte nach den Kriterien 1. Ausführungshäufigkeit, 2. Anfälligkeit für Fehlerzustände und 3. Kritikalität im Hinblick auf die Gesamtanwendung zu ordnen.

2. **Die relevanten Fehlerklassen identifizieren**: Die Arten von Fehlerursachen, die für jeden Funktionsaufruf injiziert werden sollen, werden wie in Listing 4.1 spezifiziert. Die relevanten Fehlerklassen lassen sich teilweise automatisiert ableiten – beispielsweise ergibt es wenig Sinn, bei als konstant deklarierten Argumenten Berechnungsfehler zu injizieren, oder die Injektion von Verzögerungen ist bei länger andauernden Berechnungen besonders interessant.

3. **Injektionsfrequenz und -schweregrad festlegen**: Hier konfiguriert der Nutzer je nach verfügbarem Zeitbudget und basierend auf Domänenwissen die Häufigkeit (Wird bei jedem oder lediglich bei jedem n-ten Funktionsruf injiziert?) und vordefinierte Schweregradstufen, die beispielsweise die Dauer einer injizierten Verzögerung, die Menge manipulierten Speichers oder das Ausmaß injizierter Berechnungsfehler beeinflussen.

4. **Ausführung unter Fehlerlast**: Basierend auf der Konfiguration wird die Anwendung unter Fehlerlast ausgeführt, das heißt, Funktionsrufe werden zunächst in eine generierte, Fehler injizierende Bibliothek umgeleitet. Die Ergebnisse werden mitgeschrieben und können über verschiedene Läufe hinweg verglichen werden.

4.3 Diskussion

In diesem Kapitel wurden zwei Ansätze zur Injektion von Softwarefehlerursachen an den Schnittstellen zu externen Bibliotheken vorgestellt. Beide bieten gegenüber anderen Verifikationsmethoden und Fehlerinjektoren zwei entscheidende Vorteile:

Werkzeuge, die an den Schnittstellen von Binärdateien zur Ausführungszeit arbeiten, sind mit geringem Aufwand für eine Vielzahl an Anwendungen verwendbar. Es ist möglich, sie ohne manuellen Analyseaufwand und ohne Zugriff auf den Quelltext – der bei Bibliotheken möglicherweise nicht vorliegt – einzusetzen. Formale Fehlermodelle, Spezifikationen oder eine detaillierte Dokumentation müssen nicht vorliegen – diese Informationen werden wie im Falle von LFI aus den Binärdateien extrahiert, oder sind wie im Falle von Hovac konfigurierbar und können vorher durch Vermessen oder Domänenkenntnis ermittelt werden.

Zudem ist der Testaufwand mit Bibliotheks-Fehlerinjektoren geringer oder zumindest kontrollierbarer als bei Werkzeugen zur erschöpfenden Verifikation. Man kann den Fehlerraum inkrementell bis zu einer ausreichenden Abdeckung hin testen und dabei zugleich eine Vielzahl an unterschiedlichen Fehlerursachen injizieren. Hier ist das Fehlermodell von Hovac ein Ausgangspunkt, von dem aus noch weitere anwendungsspezifische Fehlertypen denkbar sind. Sowohl LFI als auch Hovac bieten eine Schnittstelle zur einfachen Implementierung weiterer Fehlerursachen an.

5 Fehlerinjektion in verwaltete Sprachen mit Laufzeitumgebungen

Verwaltete Sprachen wie C# oder Java, bei denen Anwendungen in Laufzeitumgebungen ausgeführt werden, erlauben durch Introspektion und Instrumentierung weitreichende Modifikationen der Programmausführung. Diese Modifikationswerkzeuge erlauben ein breites Spektrum an Fehlerinjektionstests.

Im Unterschied zu den bislang untersuchten Methoden bieten Sprachen, die virtuelle Maschinen als Ausführungsumgebung nutzen, vielfältige und ausdrucksstarke Möglichkeiten, den Code zur Ausführungszeit zu untersuchen und sein Verhalten zu verändern. Zumeist kommt zwischen Quelltext und durch den Prozessor ausführbarem Binärcode eine Zwischenrepräsentation des Programms zum Einsatz, die durch einen sogenannten *Just-in-time Compiler* (JIT-Compiler) erst während der Programmausführung bei Bedarf zu direkt ausführbarem Binärcode kompiliert wird.

Bei Microsofts *.NET* heißt diese Zwischenrepräsentation *Common Intermediate Language (CIL)* [44] und kann aus Sprachen wie zum Beispiel C#, Visual Basic .NET oder F# kompiliert werden. Für Java wird die Zwischenrepräsentation *ByteCode* genannt.

Diese Architektur macht zwei wesentliche Eingriffe sehr einfach: Einerseits können durch *Introspektion* („Reflection“) sowohl verschiedenste Informationen über die Struktur und das Verhalten von Typen ermittelt werden als auch neue Typen erzeugt werden. Andererseits erlaubt die Nutzung einer virtuellen Maschine *Instrumentalisierung*, also eine Veränderung des Ausführungsverhaltens ohne Änderung des Quelltextes oder des Zwischencodes, der als Binärdatei vorliegt.

5.1 Fehlermodelle

Bei der Untersuchung eines Anwendungsprogramms ergeben sich folgende Anwendungsmöglichkeiten für Fehlerinjektion:

Testen der Ausnahmebehandlung Ein Anwendungsbeispiel ist das Testen der Reaktion eines Systems auf Ausnahmen. Es ist unter Umständen schwierig, spezifische Ausnahmen zu erzeugen, wie zum Beispiel die Zeitüberschreitung eines Sockets. Um die Behandlung einer solchen Ausnahme durch Tests abzudecken, müsste innerhalb des Testfalls eine Gegenstelle zur Socket-Kommunikation erstellt werden, welche der Spezifikation des realen Systems

entspricht und gleichzeitig auf Befehl eine Nachricht über die Zeitüberschreitung generieren kann. Da dies häufig einen größeren Aufwand verursacht, stellt Fehlerinjektion eine einfache Möglichkeit dar, um in einem Test gezielt eine Ausnahme zu generieren und deren Behandlung dann durch Tests abzudecken.

Testen des Umgangs mit Berechnungs- und Wertefehlern Ein weiteres Szenario für die Anwendung von Fehlerinjektion in Laufzeitumgebungen ist das Verändern von Rückgabewerten. Ziel ist es, die Robustheit gegenüber fehlerhaften oder unerwarteten Antworten zu testen. Ähnlich wie im zuerst beschriebenen Fall könnte der Kommunikationspartner auf eine Anfrage mit ungültigen Werten antworten, die eventuell zu einem Überlauf oder anderem fehlerhaften Verhalten führen.

Bewertung einer Testsammlung Existiert für ein Programm eine Testsammlung, kann diese bewertet werden, indem man untersucht, ob künstlich ins Programm eingefügte Fehler zu Test-Fehlschlägen führen. Umso mehr injizierte Fehler Tests fehlschlagen lassen, desto besser ist die Testsammlung zur Entdeckung von Bugs geeignet.

5.2 Implementierungsansätze

Wir unterscheiden im Wesentlichen zwischen zwei Ansätzen:

Reflexion und Abhängigkeitsinjektion (engl. „Dependency Injection“) kommt zur Injektion von veränderten Objekten zum Einsatz. Dies geschieht, oft im Rahmen von Modultests, mithilfe von sogenannten *Stubs* (ersetzen ein Objekt, sodass man mit einem anderen Objekt ohne Probleme interagieren kann; steht unter Kontrolle des Testers bzw. des Fehlerinjektors), *Mocks* (Objekt, dass in der Regel auf Grundlage von Interaktionsmustern entscheidet, ob ein Test erfolgreich war) oder *Shims* (eine Möglichkeit, die Implementierung einzelner Methoden zu überschreiben, die nicht unter Kontrolle des Fehlerinjektors stehen). [47]

Laufzeit-Instrumentalisierung kann verwendet werden, um einzelne Aufrufe abzufangen. Im Gegensatz zum Erstellen von Stub- und Mock-Objekten müssen Testfälle nicht explizit auf das zu testende Verhalten angepasst werden. So bleibt das reale Verhalten größtenteils erhalten und der Aufwand zur Anpassung wird minimiert.

Neben Modultests kann Fehlerinjektion auch zur Fehlerursachensuche (*Debugging*) verwendet werden. Beispielsweise kann im Produktivbetrieb eines Programms unter bestimmten Vorbedingungen in einer spezifischen Situation ein Ausfall auftreten, der im Normalfall – ohne die Vorbedingungen – nicht zu beobachten wäre. Um die Ursache des Ausfalls zu ermitteln, müsste jetzt ein Testfall konstruiert

werden, in dem die Vorbedingungen erfüllt sind, was sehr aufwändig sein kann. Da Vorbedingungen für den Ausfall fremdbedingt sein können, wie beispielsweise unerwartetes Zeitverhalten, müssten auch die Ausführungsumgebung und externe Komponenten vorbereitet sein, um unter genau diesen Umständen das gesuchte Verhalten zu zeigen. Da diese jedoch recht aufwändig zu implementieren sind, muss oft anhand von Aufzeichnungen der Fehler postmortem nachvollzogen werden. Oftmals wurden die relevanten Informationen nicht vom Logging erfasst, sodass im schlechtesten Fall eine neue Version mit aktualisierter Logging-Funktionalität erstellt werden muss.

5.2.1 Fehlerinjektion durch Stubs & Mocks

Der einfachere und intuitivere Ansatz ist bereits aus dem herkömmlichen Modultesten in diesen Sprachen bekannt: Die Abhängigkeit, die ersetzt werden soll, implementiert eine öffentliche Schnittstelle, die innerhalb des Programms genutzt wird. Während des Testens wird zur Laufzeit ein Stub- oder Mock-Objekt erstellt, das diese Schnittstelle rudimentär implementiert und anstelle der Abhängigkeit übergeben wird. Je nach verwendetem Isolations-Framework – welches eine Programmierschnittstelle zum einfachen Erstellen von Stub- und Mock-Objekten bietet – kann nun das Verhalten der Stub- und Mock-Objekte dynamisch festgelegt werden. Zumeist ist es auch möglich, Interaktionen aufzuzeichnen und im Nachhinein zu überprüfen. Das Isolations-Framework bedient sich dabei der Introspektions-API der Sprache, um zur Laufzeit eine Klasse zu erstellen, die die Schnittstelle vererbt. Je nach zuvor festgelegtem Verhalten führen die generierten Stub- und Mock-Objekte dann den benutzerdefinierten Code aus, geben vordefinierte Werte zurück oder generieren eine Ausnahme.

Dieses Vorgehen begrenzt die injizierbaren Methoden natürlich erheblich: Zwar muss nicht notwendigerweise eine Schnittstelle genutzt werden; es muss sich jedoch um eine öffentliche, vererbbare Klasse handeln, die virtuelle Methoden besitzt. Bei nicht-virtuellen Methoden kann zwar ein Ersatz generiert werden, das Programm wird jedoch die Original-Implementierung verwenden, da das Mock-Objekt als Instanz der Basisklasse übergeben und verwendet wird. Damit sind die meisten Klassen des .NET-Frameworks nicht gut injizierbar, da sie keine passenden Schnittstellen implementieren und viele Methoden nicht virtuell sind.

Damit dieser Ansatz funktioniert, muss die zu testende Klasse so angelegt sein, dass ihr die Abhängigkeiten nicht direkt bekannt sind, sondern durch *Abhängigkeitsinjektion* übergeben werden. Dies kann im einfachen Fall als Konstruktor-Argument oder durch Verwendung von Abhängigkeitsinjektions-Frameworks wie beispielsweise Ninject[1] geschehen. Für bereits existierende Programme, die nicht entsprechend angelegt sind, muss bei diesem Ansatz also zunächst eine Umstrukturierung durchgeführt werden, um Fehlerinjektion durch Mocking nutzen zu können. Ne-

[1] `http://www.ninject.org/`, 5. Dezember 2016.

ben höherem Aufwand bedeutet dies vor allem, dass Zugriff auf den Quelltext erforderlich ist.

In Listing 5.1 ist eine Umsetzung des Ansatzes in C#/.NET illustriert – beispielhaft wird ein Web-Client getestet, der einen Text von einem Webservice abfragt. Der Client erzeugt einen `System.Net.HttpWebRequest` und liest das Ergebnis in eine Zeichenkette, die er zurückgibt. In der beispielhaften Spezifikation ist vorgesehen, dass beim Auftreten von Netzwerkfehlern der Text „Error:“ zurückgegeben wird. Zur Umsetzung wurde die Bibliothek *Rhino Mocks*[2] verwendet, die auf *DynamicProxy*[3] des Castle Project[4] zurückgreift, um die Stub-Objekte zu erzeugen.

Listing 5.1: Injektion einer Ausnahme mittels eines Mock-Objekts in C# im .NET Framework. Es muss die Hilfsklasse `WebAccess` eingeführt werden, damit eine andere als die .NET-Framework-Implementierung des `HttpWebRequest` genutzt wird. Da `HttpWebRequest.GetResponse()` jedoch virtuell ist, gelingt in diesem Fall das Erstellen eines Stubs. Der Test führt den Webservice mit einem Attrappen-`WebAccess`-Objekt aus und kann dann prüfen, ob die zurückgegebene Zeichenkette der Spezifikation entspricht.

```
[TestMethod]
public void ServiceFailsProperly_Stubs()
{
  // Erzeugung zweier Stubs
  var webAccess = MockRepository.GenerateStub<WebAccess>();
  var webRequest = MockRepository.GenerateStub<HttpWebRequest>();

  // Überschreiben zweier Methoden der Stub-Objekte
  webAccess.Stub(a => a.CreateRequest(Arg<string>.Is.Anything)).Return(
      webRequest);
  webRequest.Stub(r => r.GetResponse()).Throw(new System.Net.WebException
      ("Special Error Details"));

  string result = MyWebService.Run(webAccess);

  Assert.IsTrue(result.StartsWith("Error:"));
}
```

5.2.2 Fehlerinjektion durch Instrumentalisierung der Laufzeitumgebung

Ist es notwendig, nicht-virtuelle Methoden oder Eigenschaften zu verändern, ist der Quelltext nicht verfügbar oder darf er nicht geändert werden, muss ein mächtigerer Ansatz wie Instrumentalisierung der Laufzeitumgebung gewählt werden.

[2] `http://www.hibernatingrhinos.com/oss/rhino-mocks`, 5. Dezember 2016.
[3] `http://www.codeproject.com/Articles/9055/Castle-s-DynamicProxy-for-NET`, 5. Dezember 2016.
[4] `http://www.castleproject.org/`, 5. Dezember 2016.

5.2.2.1 .NET Framework

Die nativen Komponenten der .NET-Runtime, die *Common Language Runtime* (CLR), beinhalten die *Profiling API*[5], die zur Fehlerinjektion genutzt werden kann. Beim Start einer .NET-Applikation wird zunächst die CLR geladen, die die Umgebungsvariable `COR_PROFILER` auf einen Identifikator einer COM-Klasse untersucht. Der Profiler wird geladen, instanziiert und wird danach bei verschiedenen Ereignissen aufgerufen. Gleichzeitig hat der Profiler die Möglichkeit, den geladenen CIL-Code zu inspizieren und zu modifizieren. Auf diese Weise kann ein Profiler jede CIL-Methode neu schreiben, bevor die Kompilierung in Maschinencode geschieht (`JitCompliationStarted`), sodass sie den Aufruf ins Fehlerinjektions-Framework weiterleitet.

TestApi Der Ansatz, den Profiler für Fehlerinjektion zu verwenden, wurde im *TestApi*-Projekt[6] umgesetzt. Er erlaubt es, selbst statische Methoden innerhalb der .NET Framework-Klassen, wie `DateTime.Now`, mit beliebigem Verhalten zu ersetzen. Der große Nachteil von TestApi ist, dass Setup und tatsächlicher Test in verschiedenen Prozessen stattfinden: Der Hauptprozess definiert zunächst sogenannte *Fault Rules*, die aus einer zu injizierenden Methode in Form ihres qualifizierten Namens als Zeichenkette sowie Eintrittsbedingungen und der zu injizierenden Fehlerursache bestehen – beispielsweise eine Ausnahme, ein veränderter Rückgabewert, oder benutzerdefiniertes Verhalten. Die Regeln werden in temporären Dateien persistiert und anschließend der injizierte Prozess gestartet, dem der TestApi-Profiler angehängt wird.

Listing 5.2: Injektion einer Ausnahme mittels Shimming mit TestApi

```
[TestMethod]
public void ServiceFailsProperly_TestApi()
{
  // FaultRule und FaultSession erstellen
  FaultRule rule = new FaultRule(
    "System.Net.HttpWebRequest.GetResponse()",
    BuiltInConditions.TriggerOnEveryCall,
    BuiltInFaults.ThrowExceptionFault(new System.Net.WebException("
        Special Error Details")));

  FaultSession session = new FaultSession(rule);

  ProcessStartInfo psi = session.GetProcessStartInfo("WebService.exe");
  psi.RedirectStandardOutput = true;
  Process p = Process.Start(psi);

  // Ausgabe ermitteln
  var output = new StringBuilder();
```

[5] `https://msdn.microsoft.com/en-us/library/bb384493.aspx`, 5. Dezember 2016.
[6] `https://testapi.codeplex.com/`, 5. Dezember 2016.

```
  while (!p.HasExited)
    output.Append(p.StandardOutput.ReadToEnd());
  output.Append(p.StandardOutput.ReadToEnd());

  string result = output.ToString();

  Assert.IsTrue(result.StartsWith("Error:"));
}
```

Die genannten Schritte – dargestellt in Listing 5.2 – erschweren den Einsatz von TestApi in umfangreichen Modultests deutlich, denn einerseits bringt es viel Aufwand mit sich – für jeden Testfall muss ein eigener Prozess gestartet werden –, andererseits beschränkt sich das beobachtbare Verhalten nun auf die Prozessgrenze: Da Test- und Kindprozess eigenständige Betriebssystem-Prozesse sind, muss der Autor des Testfalls gleichzeitig die Kommunikation zwischen den Prozessen berücksichtigen und Standard-Ein- und -Ausgabe, temporäre Dateien oder geteilten Speicher nutzen, um Testbeobachtungen oder -ergebnisse zur Auswertung zurück zum Hauptprozess zu senden.

Neben den Komponenten zur Fehlerinjektion beinhaltet TestApi weitere Bibliotheken zur Testautomatisierung, zum Testen der grafischen Oberfläche und zur Erkennung von Speicherlecks.

Pex & Moles Das bei Microsoft Research entstandene Projekt *Pex*[7] beinhaltet *Moles*, das ähnlich funktioniert, und darüber hinaus Shimming innerhalb desselben Prozesses erlaubt. Moles wurde weiterentwickelt und als *Microsoft Fakes*[24] fester Bestandteil der Visual Studio-Entwicklungsumgebung. Fakes arbeitet enger an Visual Studio gebunden und erstellt bereits während des Bau-Prozesses eigene Assemblies mit Klassen, die die Definition des Verhaltens als Delegate erlauben. Dabei wird zwischen im vorherigen Abschnitt erläuterten *Stubs*, die jedoch schon zur Kompilierzeit entstehen, und *Shims* unterschieden, die dem TestApi-Ansatz entsprechen. Auch das Fakes-Framework nutzt einen eigenen Profiler, durch die Integration in Visual Studio wird der Testprozess aber direkt mit angehängtem Profiler gestartet. Entsprechend wird das überschriebene Verhalten nicht aus Dateien ausgelesen, sondern erst während des Testlaufs im selben Prozess definiert. Dieses Vorgehen eliminiert die beiden Schwächen von TestApi und ermöglicht eine Verwendung, die sich kaum von der bereits diskutierten Verwendung von Stub- und Mock-Objekten unterscheidet, dafür aber von Visual Studio abhängig macht. Ein Anwendungsbeispiel ist in Listing 5.3 zu sehen.

Listing 5.3: Injektion einer Ausnahme mittels Shimming mit Microsoft Fakes

```
[TestMethod]
public void ServiceFailsProperly_MSFakes()
{
  using (ShimsContext.Create())
```

[7] http://research.microsoft.com/en-us/projects/pex/, 5. Dezember 2016.

```
  {
    var request = new System.Net.Fakes.ShimHttpWebRequest();
    System.Net.Fakes.ShimWebRequest.CreateHttpString = (url) => request.
        Instance;
    request.GetResponse = () => { throw new System.Net.WebException("
        Special Error Details"); };

    string result = MyWebService.Run();

    Assert.IsTrue(result.StartsWith("Error:"));
  }
}
```

Auch bei Pex ist die Fehlerinjektion nur ein Teil eines Projektes, das beispielsweise auch durch Quelltextanalyse Testfälle mit hoher Abdeckung generieren kann.

5.2.2.2 Java

Der *Java Instrumentation Agent* erlaubt es, mit Hilfe der *Java Plattform Debugger Architecture* Klassen zur Laufzeit neu zu laden und bisherige Instanzen der Klasse zu ersetzen, um so neues Verhalten bzw. Fehler gezielt zu injizieren. [23, 28] Der sogenannte Agent wie beispielsweise Byteman dient dabei als Werkzeug zur Veränderung des Quelltextes, welches dann Javassist verwendet um den neuen Quelltext zu kompilieren. Da *Java Classloader* grundsätzlich nicht das Neuladen bereits geladener Klassen erlauben, muss die Debugger Architecture verwendet werden, um zur Laufzeit Bytecode zu ersetzen. So ist es möglich, Quelltext und somit Fehlerursachen bzw. Bugs in ein bereits laufendes Programm mit vollständig geladenen Klassen zu injizieren.

Die Fehlerinjektion findet an fest definierten Punkten mit einem vom Nutzer fest definieren Verhalten statt. Der Nutzer muss deshalb über ein tiefgreifendes Wissen über das zu injizierende System verfügen, um die Injektionsmöglichkeiten gezielt einzusetzen. Damit grenzt sich die Fehlerinjektion in Java von Blackbox-Ansätzen wie dem Zufallstesten (englisch *Fuzzing*) ab. Dementsprechend ist das typische Anwendungsgebiet nicht das Finden von Bugs in bestehendem Quelltext, sondern der Einsatz für Modultests um die Robustheit eines Programms gegenüber Fehlerursachen zu testen. [52]

Die Vorgehensweisen der beiden Fehlerinjektoren *JACA* der University of Campinas und *Byteman*, das von Jboss entwickelt wird, sind einander ähnlich – beide verwenden *Javassist*[8] [9] zum Ändern und Neukompilieren von bereits kompilierten Klassen. Anhand der aus den Bytecode extrahierten Klassen- und Methodensignaturen bieten die Werkzeuge die Möglichkeit, an bestimmten Stellen innerhalb der Klassen die ursprüngliche Funktionalität zu ändern. Die Injektionsmöglichkeiten umfassen das Ändern von Parametern vor Beginn einer Methode und von Rückgabewerten am Ende einer Methode. Zusätzlich hat man mit Byteman die Möglichkeit, nahezu beliebigen Quelltext in Funktionen hinein zu injizieren.

[8] `http://jboss-javassist.github.io/javassist/`, 5. Dezember 2016.

JACA JACA ist ein Fehlerinjektionswerkzeug, dessen Fokus weniger auf dem dynamischen Anlegen und Injizieren von Fehlern liegt als auf der regelbasierten Wiederholung von Fehlersituationen und dem Vergleich mit einem fehlerlosen Ablauf. [43] Es ist ein grafisches Werkzeug, welches auf Java 1.4 basiert. Dennoch ist es auch möglich, Fehlerursachen in Programme höherer Versionen zu injizieren.

Pro JACA-Projekt können beliebig viele Fehlerklassen angelegt werden, die dann bei der Ausführung injiziert werden. Um dies zu überwachen, kann für jede Klasse ein Monitor erstellt werden, der die Aufrufe seiner Methoden sowie die injizierten Fehlerursachen mitschreibt. Fehlerursachen und Monitore können dann in beliebiger Permutation zu Kampagnen zusammengestellt werden, welche dann ausgeführt werden können.

Bei der Ausführung zeichnet JACA sowohl die normale Ausgabe des Programms als auch die der Monitore auf und legt sie in Dateien im Projektpfad ab. Theoretisch möglich ist auch ein sogenannter „Golden Run“, bei dem keine Fehler injiziert werden, um diesen dann mit den fehlerhaften Durchläufen zu vergleichen.

Die injizierbaren Fehlerzustände sind in drei Kategorien unterteilt:

Rückgabewerte JACA verändert am Ende einer Methode den tatsächlich zurückgegebenen Wert, der ursprüngliche Wert wird verworfen.

Parameter Am Anfang der Methode wird ein Parameter überschrieben und dann der neue Wert innerhalb der Methode verwendet.

Attribut Beim Lesen des Feldes wird ein anderer Wert zurückgegeben. [9]

Für jede Fehlerkategorie kann der neue Wert und das Wiederholungsmuster festgelegt werden. Bei der Angabe der neuen Werte ist man auf eine Regelsprache eingeschränkt, die nur die primitiven Datentypen unterstützt. Dementsprechend ist es auch nur möglich, Fehlerursachen in Werte primitiver Datentypen zu injizieren. Auch das Injizieren von eigenem Quelltext ist nicht möglich. Die Fehlerursache kann entweder immer, einmalig oder zufällig mit einer einstellbaren Wahrscheinlichkeit pro Aufruf injiziert werden. Zusätzlich lässt sich noch ein Startpunkt nach der Anzahl der Aufrufe definieren.

Um die Funktionsweise der Werkzeuge zu vergleichen, sei als Szenario ein Programm gegeben, dass die Klasse `Math` mit der Funktion `multiply(int x, int y)` enthält. Es soll nun jeweils eine Fehlerursache injiziert werden, damit das Ergebnis von `multiply` nicht korrekt ist. Eine beispielhafte JACA-Konfiguration ist in Tabelle 5.1 dargestellt.

JACA liefert nach erfolgreichem Ausführen der Fehlerinjektion neben der Ausgabe des Programms die Ausgaben der Monitore, die über jede Fehlerursache informieren. Zusätzliche Konfiguration der Fehlerursachen ist über eine XML-Datei theoretisch möglich, jedoch ist die Dokumentation hier lückenhaft, sodass unklar

[9] Die Implementierung dieser Fehlerkategorie wurde von uns mehrfach getestet, hat jedoch aus uns unbekannten Gründen nicht funktioniert – weder wurde die Fehlerursache injiziert, noch hat der dafür angelegte Monitor den Aufruf registriert.

Tabelle 5.1: Konfiguration einer Fehlerursache in JACA. Es wird der erste Parameter der Funktion `multiply(int, int)` gleich `0` gesetzt. Die Fehlerursache wird dauerhaft ab dem fünften Aufruf der Funktion injiziert.

Regel-Element	**Wert**
Type	`Parameter`
Class	`class de.hpi.fault_injection.Math`
Method	`public int de.hpi.fault_injection.Math.multiply(int,int)`
Parameter	`0`
Rule	`=, int 0`
Repetition Pattern	`Permanent`
Start	`5`

bleibt, wie es möglich ist, diese Konfiguration auch zu verwenden. Es wird deshalb in diesen Bericht nicht darauf eingegangen.

Byteman Byteman ist ein von JBoss entwickeltes Werkzeug zur Quelltextinjektion.[10] Das gemeinschaftlich getragene quelloffene Projekt wird seit 2009 ständig weiterentwickelt.[11] Neben der Fehlerinjektion ist auch das Erweitern mit Programmverfolgungs-Quelltext eines der Einsatzgebiete von Byteman. Ebenso wie JACA verwendet Byteman Javassist, das ebenso von JBoss entwickelt wird; die Entwickler arbeiten derzeit an beiden Projekten parallel.

Im Gegensatz zu JACA ist Byteman ausschließlich kommandozeilenbasiert. Über die Kommandozeile wird ein Java-Programm zusammen mit Byteman als Instrumentierungsagent ausgeführt. Um Quelltext zu injizieren, werden Byteman in einer domänenspezifischen Regelsprache übergeben. Neben der Injektion von nahezu beliebigem Java-Quelltext steht eine Reihe von eingebauten Funktionen bereit, mit denen das Fehlverhalten genau gesteuert werden kann. Im Folgenden werden die Regelsprache und einige Funktionen genauer beschrieben, bevor auf das Anwendungsbeispiel von JACA eingegangen wird.

Eine Regel besteht aus mehreren Parametern: Dem Namen der Regel, dem Namen der Klasse, der Signatur der zu injizierenden Methode, dem Eintrittspunkt der Injektion, einer Menge von Variablen, Bedingungen und beliebigem Java Quelltext zur Ausführung an der beschriebenen Stelle.

Für die Definition von Eintrittspunkten stehen die folgenden Möglichkeiten bereit:

AT ENTRY Bei Eintritt in die Funktion.

AT EXIT Beim Verlassen der Funktion.

AT INVOKE Beim Aufrufen einer Funktion innerhalb der oben aufgeführten Funktion.

[10] `http://byteman.jboss.org/`, 5. Dezember 2016.
[11] `https://github.com/bytemanproject/byteman`, 5. Dezember 2016.

AT READ / WRITE Beim Lesen oder Schreiben einer Variable.

Zusätzlich verfügt Byteman über eine Reihe von eingebauten Funktionen, die eine feinere Steuerung der Injektion erlauben. Es handelt sich um Objekte, die global definiert sind und so über Regeln hinweg zur Koordination genutzt werden können. Die folgenden eingebauten Funktionalitäten stehen global bereit:

CountDowns Numerische Werte, die bis 0 herunter gezählt werden. So kann man erreichen, dass eine Fehlerursache erst ab dem n-ten Funktionsaufruf injiziert wird.

Flags Wahrheitswerte.

Counter Numerische Werte, die anders als CountDowns inkrementiert und dekrementiert werden können.

Timer Variablen, die über Regeln hinweg Zeit messen.

Rendezvous Eine Barriere, an der ein Ausführungsfaden angehalten wird. Sobald n Ausführungsfäden an einem Rendezvous-Punkt angekommen ist, werden alle gemeinsam weiter ausgeführt.

Waiter Halten einen Ausführungsfaden an, bis ein anderer Ausführungsfaden den Waiter benachrichtigt.

Mit CountDowns und Countern kann die Funktionalität von JACA abgebildet werden, eine vordefinierte Anzahl von Fehlerzuständen ab einem bestimmten Punkt zu injizieren. Zusätzlich verfügt Byteman aber auch über Funktionen zur Synchronisierung von Ausführungsfäden. Dies ist unter anderem hilfreich, um das einzelne Eintreten in einen kritischen Abschnitt zu testen. Man kann sicherstellen, dass ein Rendezvous-Punkt innerhalb des kritischen Abschnitts nicht von mehreren Ausführungsfäden gleichzeitig erreicht werden darf. Timer finden bei der Ablaufverfolgung und beim Vermessen Verwendung.

Soll das für JACA eingeführte Beispielprogramm mit Hilfe von Byteman injiziert werden, muss dafür eine Regel geschrieben werden. Eine Beispielregel ist in Listing 5.4 zu sehen.

Listing 5.4: Beispielregel für JACA, mit der bei dem fünften Aufruf der multiply-Methode das erste Argument auf 0 gesetzt wird: Die `DO`-Anweisung der Regel wird ausgeführt wenn der CountDown erstellt (mit 5 initialisiert) und nach fünfmaligem Aufruf der Regel auf 0 herunter gezählt wurde. Danach wird das erste Argument der `multiply(int, int)`-Methode bei Beginn der Methode gleich 0 gesetzt.

```
RULE injection test
CLASS de.hpi.fault_injection.Math
METHOD multiply(int, int)
AT ENTRY
IF !createCountDown("counter", 5) && countDown("counter")
```

```
DO $1 = 0;
ENDRULE
```

Ausgeführt wird das Programm durch Aufruf von Java mit Byteman als Agent und den Regeln als Argument:

```
java -javaagent:<Pfad zu Byteman>=script:<Regeldatei> <Main> <Args>
```

Um den Aufruf zu verkürzen, stellt Byteman mehrere Skripte bereit; für das Starten der *Java Virtual Machine* (JVM) beispielsweise:

```
byteman.bat -l <Regeldateien> <Main> <Args>
```

Das Skript registriert außerdem den Byteman-Classloader als initialen Classloader, um dadurch die Injektion von Java Standardklassen zu erlauben. Weitere Regeln können durch das Skript *bmsubmit* einem laufenden Agenten übergeben werden, durch das Argument -l werden die aktuell verwendeten Regeln aufgelistet. Für die Kommunikation mit dem Agenten wird dabei eine TCP Verbindung genutzt. Hierfür muss der Standard-Port 9091 frei sein, alternativ muss sowohl beim Agenten als auch für das *bmsubmit* der Port spezifiziert werden. *bminstall* ermöglicht das nachträgliche Installieren eines Agenten in einem laufenden Programm.

5.3 Diskussion

Aufgrund des Aufbaus der Sprache ermöglicht .NET die dynamische Injektion von Fehlerursachen – beispielsweise in Form von Ausnahmen – auf verschiedenen Wegen, die sich vor allem durch ihre Ausdrucksstärke unterscheiden. Bei Software, deren Quelltext verändert werden kann, ist es einfach, das Verhalten von externen Modulen durch Mock-Objekte abzubilden, um den Testumfang zu reduzieren und die Fehlerbehandlung und den Umgang mit Ausnahmefällen zu testen. Ausführliche Tests mit Isolation durch Mocking gehören inzwischen zu bewährten Praktiken, für die bereits zahlreiche Frameworks entstanden sind, und decken meist – wenn auch nicht als Fehlerinjektion bezeichnet – auch den Umgang mit Fehlerzuständen ab. In komplizierteren Fällen, in denen Verhalten des .NET-Frameworks oder nichtvirtuelle Methoden geändert werden müssen, oder das Programm nicht verändert werden kann, besteht über die nativen Schnittstellen der CLR die Möglichkeit, gezielt spezielles Verhalten einzufügen.

Die Byteman und JACA erlauben es, auch ohne Zugriff auf den Quellcode, Fehlerursachen anhand der Klassen und Methodensignaturen direkt in Bytecode zu injizieren. Byteman ist für vielseitige Anwendungsfälle ausgelegt. Es kann beliebiger Java Quelltext injiziert werden, wodurch sowohl Fehlerursachen als auch neue Aspekte zu einem System hinzugefügt werden können. JACA bietet die Möglichkeit vordefiniertes Fehlverhalten zu injizieren, wodurch eine beschränkte Auswahl fehlerhafter Zustände injiziert werden kann.

Da das Injizieren sowohl in eigenem als auch fremdem Quelltext (Bibliotheken, Standardbibliothek) möglich ist, eignen sich die beschriebenen Fehlerinjektionswerkzeuge insbesondere, um die Reaktion auf fehlerhafte Werte oder Ausnahmen zu testen. Ein robustes Programm sollte mit beliebigen Rückgabewerten umgehen können. Da es jedoch unter Umständen schwierig oder unmöglich ist, eine Bibliothek zur Ausgabe eines bestimmten Wertes zu zwingen, um das Verhalten darauf testen zu können, ist Fehlerinjektion ein nützliches Hilfsmittel, um diese Fälle abzudecken.

6 Fehlerinjektion in Grafikbeschleuniger

Mit der zunehmenden Verwendung von Grafikkarten für nicht-grafische Berechnungsaufgaben steigen die Anforderungen an deren Fehlertoleranz. Fehlerinjektion für GPUs ist aufgrund des hohen Parallelitätsgrades und der schwierigen Beobachtbarkeit eine Herausforderung. Hier wird der Fehlerinjektor GPU-Qin diskutiert, der zum Testen der Fehlertoleranz von CUDA-Anwendungen dient.

Grafikkarten wurden ursprünglich mit dem Ziel entworfen, die Bildsynthese zu beschleunigen. Diese Berechnungslast stellt typischerweise keine hohen Anforderungen an Verlässlichkeit – ist ein einzelner Pixel in einem Bild fehlerhaft, so fällt es dem Endnutzer höchstwahrscheinlich nicht auf. Entsprechend wurde bei der Konzeption der Hardwarearchitektur vor allem auf Performanz, kaum jedoch auf Fehlertoleranz- oder Fehlererkennungsmechanismen Wert gelegt.

Die massiv parallele Grafikhardware wird nun allerdings vermehrt auch als Beschleuniger für andere Berechnungsaufgaben verwendet – ein Trend, den man als *General-purpose GPU*-Rechnen (GPGPU, Vielzweck-Grafikprozessor) bezeichnet. Im Zuge dieser Entwicklung ist es wichtig geworden, die Verlässlichkeit von Programmen, die auf der Grafikkarte laufen, zu evaluieren. Programme, die auf einer *Graphics Processing Unit* (GPU, Grafikprozessor) ausgeführt werden, müssen oft eine größere Anzahl an Fehlerursachen aus der Hardware tolerieren können, als herkömmliche CPU-basierte Programme.

6.1 Fehlermodelle und Fehlertoleranzmechanismen für Grafikkarten

In Hardware-Fehlermodellen unterscheidet man zwischen *permanenten* und *transienten* Fehlerursachen (engl. *faults*).[1]

Permanente Fehlerursachen sind oft durch Fehler in der Konzeption der Hardware, oder durch Verarbeitungsschäden einzelner Teile begründet. Hierbei ist der Zustand der Hardware permanent, also unabhängig vom Beobachtungszeitpunkt, korrumpiert. Ein prominentes Beispiel ist der Intel Pentium FDIV-Bug [50], bei dem ein Hardwarefehler zu ungenauen Ergebnissen bei der Division von Gleitkommazahlen führte. Permanente Fehlerursachen sind in der Regel gut erkennbar und

[1] Wir sprechen hier von Fehler*ursachen*, da die Zustände, die im Folgenden beschrieben werden, als Störungen der Softwareschicht zu betrachten sind, die wiederum zu Fehlerzuständen im laufenden Programm führen können. Allerdings wäre auch der Begriff „Fehlerzustand“ (engl. *error*) angemessen.

führen dann zu einer schnellen Wartung oder einem Austausch der betroffenen Hardware.

Anders verhält es sich bei transienten Fehlerursachen, die oft nichtdeterministisch auftreten, da sie externe Auslöser wie kosmische Strahlung, Überhitzung bestimmter Komponenten oder elektrostatische Effekte haben. Solche Fehlerursachen werden aufgrund steigender Strukturdichten immer wahrscheinlicher und müssen durch anspruchsvollere Fehlerkorrekturmechanismen behandelt werden.

Für Grafikhardware ist es weiterhin relevant, bei der Formulierung des Fehlermodells zwischen der Anzahl betroffener Bits und möglicher Zustände zu differenzieren:

Single-Bit-Fehler betreffen nur ein einziges Bit.

Multiple-Bit-Fehler betreffen mehrere Bits, die zum gleichen Zeitpunkt korrumpiert werden. In Fehlermodellen wird oft von n-Bit-Fehlern gesprochen, womit dann gleichzeitige Korruption von bis zu n Bits gemeint sind.

Stuck-At-Fehler beschreiben Szenarien, in denen die betroffenen Bits entweder nur noch 0 („Stuck-At-0“) oder nur noch 1 („Stuck-At-1“) enthalten können – auch wenn ein anderer Wert geschrieben wird.

Fehlertoleranz in Hardware wird meist in Form fehlerkorrigierender Codes (*Error Correcting Code*) implementiert. Da solche Codes mit höheren Kosten und Rechenaufwand verbunden sind, werden sie insbesondere auf Grafikhardware sparsam verwendet. NVIDIA Grafikkarten unterstützen seit der Fermi-Architektur ECC für Register, DRAM, sowie die Zwischenspeicherhierarchie. Hierbei wird *SECDED* („Single-bit Error Correction, Double-bit Error Detection“; Ein-Bit-Fehlerkorrektur, Zwei-Bit-Fehlererkennung) implementiert. [20]

6.2 Fehlerinjektion auf Grafikkarten am Beispiel von GPU-Qin

Um die Ausfallsicherheit auch in der Präsenz von Hardwarefehlerursachen auf der Grafikkarte zu testen, wurde *GPU-Qin* konzipiert. [15]

Eine Problematik bei Fehlerinjektion auf Grafikkarten ist der gewaltige theoretisch mögliche Zustandsraum, der durch die vielen parallelen Ausführungspfade aufgespannt wird. Obwohl es extrem aufwändig wäre, sämtliche mögliche Verschachtelungen der parallelen Ausführungsfäden mit Fehlerinjektionstests abzudecken, möchte man dennoch eine sowohl zeiteffiziente, als auch für tatsächliche Probleme repräsentative Abdeckung erreichen.

GPU-Qin basiert auf der Annahme, dass wie oben beschrieben, die Zwischenspeicherhierarchie, der Arbeitsspeicher und die Register durch ECC geschützt sind und somit zumindest Single-Bit-Fehler tolerieren. Die Injektion beschränkt sich daher auf das Fehlermodell *transienter Hardwarefehlerursachen in funktionalen Einheiten des Grafikkartenprozessors* wie der *Arithmetic Logical Unit* (ALU, arithmetisch-logische

Berechnungseinheit) oder *Load Store Unit* (LSU, Lade- und Speicher-Einheit). Als de-facto-Standard wurde auch hier ein Single-Bit-Fehlermodell vorausgesetzt.

GPU-Qin ist der erste uns bekannte Fehlerinjektor für Grafikkarten und adressiert die folgenden Anforderungen:

Repräsentativität Die injizierten Fehlerursachen sollten repräsentativ für tatsächlich auftretende Hardwarefehlerursachen sein und gleichmäßig über alle ausgeführten Instruktionen verteilt auftreten.

Effizienz Die Fehlerinjektion sollte so effizient sein, dass sie statistisch signifikante Aussagen über die Resilienz der Anwendung innerhalb absehbarer Zeit liefert.

Minimale Interferenz Die Fehlerinjektionsexperimente sollten so wenig wie möglich mit der Anwendung interferieren, das bedeutet, die Verlässlichkeitscharakteristika, der Quelltext selbst, oder die Daten sollten nicht durch die Fehlerinjektion modifiziert werden.

6.2.1 Implementierung

GPU-Qin besteht aus einer Profiler- und einer Injektorkomponente. Die Implementierung, die auf dem Simulator GPGPUSim[2] sowie der CUDA-Werkzeugkette basiert, ist in [15] genauer beschrieben und soll im Folgenden kurz zusammengefasst werden.

Bei der Anwendung von GPU-Qin gibt es drei Phasen:

1. Zunächst werden die Ausführungsfäden anhand ähnlichen Verhaltens gruppiert. Als Metrik hierfür dient die Anzahl der Instruktionen.

2. Dann wird für je einen stellvertretenden Ausführungsfaden aus jeder Gruppe ein Vermessen-Lauf durchgeführt, während dessen eine Ablaufverfolgung des auf der Grafikkarte laufenden Programmteils durchgeführt wird. Dies geschieht unter Zuhilfenahme der *cuda-gdb* Werkzeuge.[3] Weiterhin geschieht in der Vermessen-Phase eine Abbildung von Quelltextzeilen auf Instruktionen in der CUDA-Assembly.

3. Liegen eine Gruppierung der Ausführungsfäden anhand ihres Verhaltens sowie die Ablaufprofile je eines Stellvertreters vor, kann die eigentliche Fehlerinjektion beginnen. Hierbei wir aus jeder Gruppe ein Ausführungsfaden für Fehlerinjektionsexperimente verwendet, die wiederum nach der Größe ihrer Gruppe gewichtet werden.

Das Vorgehen bei der tatsächlichen Fehlerinjektion ist in Abbildung 6.1 zu sehen. In den ersten beiden Phasen wurden basierend auf der Instruktionsfolge und der Gruppierung von Ausführungsgruppen bedingte Haltepunkte in der Anwendung

[2] `http://www.gpgpu-sim.org/`, 5. Dezember 2016.
[3] `http://docs.nvidia.com/cuda/cuda-gdb/`, 5. Dezember 2016.

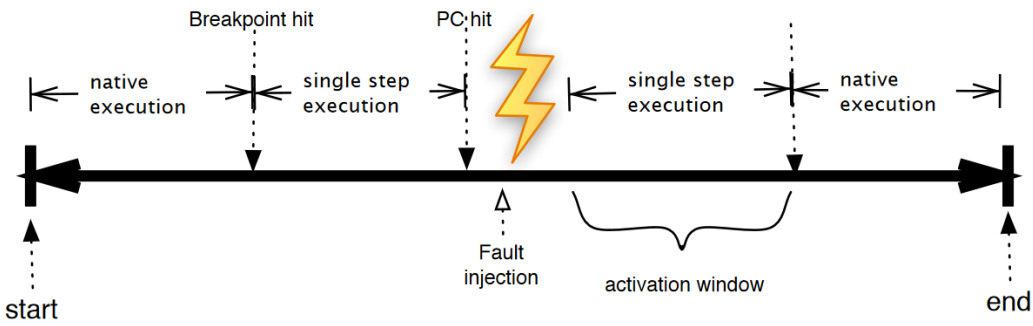

Abbildung 6.1: Vorgehen bei der Fehlerinjektion mit GPU-Qin, welches auf cuda-gdb basiert. Aus [15].

definiert. Wird dieser Haltepunkt zur Ausführungszeit auf der Grafikkarte erreicht, werden abhängig von dem aktuellen Instruktionstyp Fehlerursachen injiziert:

Arithmetische Instruktionen Das Zielregister wird manipuliert, um so Fehlverhalten der ALU oder Gleitkomma-Berechnungseinheit zu simulieren.

Speicherinstruktionen Die Ziel- und Adressregister von Speicher- oder Ladeinstruktionen werden verändert. Da der Speicher selbst als ECC-geschützt angenommen wird, repräsentiert dies Fehlverhalten der LSU.

Kontrollflussinstruktionen Die Quellregister einer Kontrollflussverzweigung werden verändert – womit verschiedenstes allgemeines Fehlverhalten ausgelöst werden kann.

Wie häufig bei der Fehlerinjektion muss auch im Anwendungsfall von GPU-Qin eine Abwägung zwischen der Abdeckung des Fehlerraums – der in diesem Falle alle möglichen Single-Bit-Fehler in allen möglichen Instruktionen umfasst, also groß ist – und der Effizienz gemacht werden. Für GPU-Qin werden drei Heuristiken zu diesem Zwecke vorgeschlagen. Erstens wird, wie erwähnt, nur in einen repräsentativen Ausführungsfaden pro Gruppe mit gleichen Charakteristiken injiziert. Zweitens wird die Anzahl der mit Fehlerinjektionstests versehenen Schleifendurchläufe stets auf 64 eingeschränkt. Drittens gilt eine Fehlerursache als „nicht aktiviert“, also keinen Ausfall verursachend, wenn sie über 1600 Instruktionen hinweg zu keiner Datenmodifikation geführt haben.

Das Resultat der Fehlerinjektionen wird also über ein Intervall von 1600 Instruktionen hinweg schrittweise beobachtet. Falls das Programm in diesem Zeitraum abstürzt, hängt, oder eine falsche Ausgabe liefert – ein Fall von *Silent Data Corruption* (SDC, stille Datenkorruption), wird der Ausfall dokumentiert; andernfalls gilt die injizierte Fehlerursache als toleriert.

6.2.2 Experimente mit GPU-Qin

Die Installation und Verwendung von GPU-Qin gestaltete sich aufgrund der Abhängigkeit vom Python-Modul *pexpect* unter Windows als schwierig.

Die Autoren von [15] haben ihr Werkzeug anhand von verschiedenen Benchmark-Programmen (aus den Parboil- und Rodinia-Sammlungen [8, 54] entnommen) auf NVIDIA Tesla C Grafikkarten getestet. Dabei wurde zum Einen festgestellt, dass die vorgestellte Methodik mit cuda-gdb um eine Größenordnung performanter ist als eine entsprechende Ausführung mit Fehlerinjektion in einem GPU-Simulator. Auch die Heuristiken zur Effizienzsteigerung wurden beurteilt und für angemessen befunden, weil sie unter Variationen zu ähnlichen Ausfallraten führten.

Bei den Experimenten, die zu einem Absturz des Programms führten, waren stark von der Anwendung abhängige Ursachen zu beobachten. Das beobachtete Ausfallverhalten ließ sich jeweils auf die Charakteristika des entsprechenden „Berkeley-Dwarf“ genannten Typen paralleler Berechnungsprobleme [2] zurückführen.

6.3 Diskussion

GPU-Qin ist er erste und einzige uns bekannte publizierte und quelloffene Fehlerinjektor für Single-Bit-Fehlermodelle in Grafikkarten.

Das datenparallele Ausführungsmodell auf Grafikkarten sieht vor, dass – auf unterschiedlichen Werten – von den verschiedenen Ausführungsfäden jeweils sehr ähnliche Berechnungen ausgeführt werden. Von der Programmierung mithilfe vieler Verzweigungen wird aus Performanzgründen ohnehin abgeraten. Insofern erscheint die Gruppierung anhand des Ausführungsprofils für viele Grafikkartenberechnungslasten nicht besonders aussagekräftig.

Die Gruppierung von Ausführungsfäden, wie sie von den Autoren vorgeschlagen wurde, ist bereits Teil des Programmiermodells für GPUs.

Wie immer bei Single-Bit-Fehlermodellen sollte auch ein kritischer Blick auf dessen Repräsentativität geworfen werden. ECC ist, wie bereits erwähnt, nämlich ein etabliertes und effektives Mittel gegen solche Fehler und wird verwendet, wann immer es sich aus Kostenerwägungen heraus lohnt. Auch wenn es in der Forschungsliteratur noch immer das vorherrschend diskutierte Fehlermodell ist (möglicherweise aufgrund seiner einfachen Erkenn- und Tolerierbarkeit), führen multiple gleichzeitige Bitfehler zu komplexeren, von Grafikhardware derzeit nicht tolerierbaren Fehlerzuständen, die ebenfalls untersucht werden sollten.

Insbesondere in Anbetracht des in Hardware vorhandenen SECDED – welches auch die Erkennung von zwei-Bit-Fehlern ermöglicht – wäre es spannend zu untersuchen, inwiefern diese Fehlermeldungen berücksichtigt werden.

7 Zusammenfassung

In dem vorliegenden Bericht wurde ein Spektrum an bestehenden Fehlerinjektionsansätzen für Software diskutiert und evaluiert.

Die Betriebssystemforschung hat Fehlerinjektionstechniken in Software maßgeblich vorangetrieben. Da an Betriebssysteme erhöhte Verlässlichkeitsanforderungen bestehen, existiert ein breites Spektrum an Ansätzen, die Robustheit von Betriebssystemen zu testen. Fehlerinjektion basiert hierbei auf Fehlermodellen, die sowohl Software- als auch Hardwarefehlerursachen beinhalten. Bestehende Fehlerinjektoren sind ein etabliertes Werkzeug und werden aktiv bei der Entwicklung von Betriebssystemen verwendet. In Kapitel 2 wurde die Geschichte dieser Werkzeuge umrissen und Fallstudien mit dem Programm *crashme* vorgestellt. In Kapitel 3 wurde der Fokus auf Werkzeuge für das Zufallstesten (Fuzzing) für Systemrufe gelegt. *Trinity*, aber auch andere Fuzzer haben in der Linux-Welt bereits viele Bugs aufgefunden.

Die Schnittstellen zwischen Softwarekomponenten bieten gute Fehlerinjektionspunkte. Angesichts immer komplexerer Interaktionen zwischen heterogenen Softwaremodulen ist es wichtig, gerade die Robustheit an den Schnittstellen ausgiebig zu testen. In Kapitel 4 wurden existierende Fehlermodelle für diesen Anwendungsfall diskutiert. Fehlerinjektoren, die an der Schnittstelle zwischen Softwarekomponenten aktiv werden, sind beispielsweise die von uns untersuchten Werkzeuge *LFI* und *Hovac*.

Objektorientierte Programmiersprachen, die in einer Laufzeitumgebung ausgeführt werden, ermöglichen vielfältige Fehlerinjektionsexperimente. Fehlerinjektion für Betriebssysteme oder systemnahe, in kompilierten Sprachen wie C und C++ geschriebene Software zu entwickeln, ist aufwändig, da zur Ausführungszeit in Binärdateien injiziert werden muss, über die nicht viele Informationen vorliegen. Anders verhält es sich bei interpretierten Sprachen oder solchen, die in einer eigenen Laufzeitumgebung bzw. „virtuellen Maschine“ ausgeführt werden – dort liegen auch zur Laufzeit viele Informationen und Manipulationsmöglichkeiten vor, die Fehlerinjektion vereinfachen. Wir haben daher beispielhaft zwei solche Ansätze betrachtet: In Kapitel 5 wurde auf das Fehlerinjektionstesten mit Unterstützung von verschiedenen Werkzeugen innerhalb des .NET-Frameworks eingegangen; ebenso wird beschrieben, wie man mithilfe der Werkzeuge *JACA* und *Byteman* Fehlerinjektion in Java Programmen betreiben kann.

Alternative Hardwarearchitekturen erfordern neue Ansätze zur Fehlerinjektion. Für schwierige Berechnungslasten, beispielsweise im Hochleistungsrechnen, werden auch unkonventionelle Hardwarearchitekturen, beispielsweise Grafikkarten, die als Beschleuniger verwendet werden, relevant im Hinblick auf Verlässlichkeit. In Kapitel 6 wurde diskutiert, wie man die Fehlertoleranz von GPU-Anwendungen testen kann. Hierbei stellt der massive Parallelismus eine zentrale Herausforderung dar. Das Werkzeug *GPU-Qin* ist der erste Fehlerinjektor für Grafikkarten und wurde von uns evaluiert.

Danksagung

In erster Linie gilt unser Dank den Teilnehmern der Lehrveranstaltung „Fehlerinjektion in Software“, die im Rahmen des Masterstudiums am HPI durchgeführt wurde. Sie haben im Rahmen des Seminars die wissenschaftliche Literaturrecherche sowie praktische Evaluierung der vorgestellten Fehlerinjektionsansätze durchgeführt: Sebastian Gerstenberg, Anne Gropler, Angelo Haller, Lukas Pirl, Patrick Siegler und Sijing You.

Quellenangaben

[1] H. Alemzadeh, R. K. Iyer, Z. Kalbarczyk und J. Raman. „Analysis of safety-critical computer failures in medical devices“. In: *Security & Privacy, IEEE* 11.4 (2013), Seiten 14–26.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams u. a. *The landscape of parallel computing research: A view from berkeley*. Technischer Bericht. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[3] A. Avižienis, J.-C. Laprie und B. Randell. „Dependability and its threats: a taxonomy“. In: *Building the Information Society*. Springer, 2004, Seiten 91–120.

[4] A. Avižienis, J.-C. Laprie, B. Randell u. a. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science Newcastle upon Tyne, UK, 2001.

[5] M. Ball und F. Hardie. „Effects and detection of intermittent failures in digital systems“. In: *Proceedings of the November 18–20, 1969, fall joint computer conference*. ACM. 1969, Seiten 329–335.

[6] M. Broy. „Challenges in automotive software engineering“. In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, Seiten 33–42.

[7] G. J. Carrette. „Crashme“. In: *Web Page https://crashme.codeplex.com/, 22.11.2016* (1996).

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee und K. Skadron. „Rodinia: A benchmark suite for heterogeneous computing“. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE. 2009, Seiten 44–54.

[9] S. Chiba. „Load-time structural reflection in Java“. In: *ECOOP 2000—Object-Oriented Programming*. Springer, 2000, Seiten 313–336.

[10] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray und M.-Y. Wong. „Orthogonal defect classification-a concept for in-process measurements“. In: *Software Engineering, IEEE Transactions on* 18.11 (1992), Seiten 943–956.

[11] F. Cristian. „Understanding fault-tolerant distributed systems“. In: *Communications of the ACM* 34.2 (1991), Seiten 56–78.

[12] J. DeMott. „The evolving art of fuzzing“. In: *DEF CON* 14 (2006).

[13] M. Eisenstadt. „My hairiest bug war stories“. In: *Communications of the ACM* 40.4 (1997), Seiten 30–37.

[14] C. Evans und T. Ormandy. *The poisoned NUL byte, 2014 edition*. 25. Aug. 2014. URL: http://googleprojectzero.blogspot.de/2014/08/the-poisoned-nul-byte-2014-edition.html (besucht am 2015-05-27).

[15] B. Fang, K. Pattabiraman, M. Ripeanu und S. Gurumurthi. „Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications“. In: *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE. 2014, Seiten 221–230.

[16] L. Feinbube, P. Tröger und A. Polze. „The landscape of software failure cause models“. In: *CoRR* abs/1603.04335 (2016).

[17] J. E. Forrester und B. P. Miller. „An empirical study of the robustness of Windows NT applications using random testing“. In: *Proceedings of the 4th USENIX Windows System Symposium*. Seattle. 2000, Seiten 59–68.

[18] B. Garn und D. E. Simos. „Eris: A tool for combinatorial testing of the Linux system call interface“. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*. IEEE. 2014, Seiten 58–67.

[19] C. Giuffrida, A. Kuijsten und A. S. Tanenbaum. „EDFI: A dependable fault injection tool for dependability benchmarking experiments“. In: *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*. IEEE. 2013, Seiten 31–40.

[20] P. N. Glaskowsky. „NVIDIA's Fermi: the first complete GPU computing architecture“. In: *White paper* 18 (2009).

[21] R. Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013.

[22] G. Hunt und D. Brubacher. „DETOURS: Binary interception of win32 functions“. In: *3rd usenix windows nt symposium*. 1999.

[23] *Instrumentation (Java Platform SE 7)*. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html (besucht am 2016-10-06).

[24] *Isolating Code Under Test with Microsoft Fakes*. URL: https://msdn.microsoft.com/en-us/library/hh549175.aspx (besucht am 2016-10-06).

[25] D. Jones. *Trinity 1.5*. 2. März 2015.

[26] D. Jones. *Trinity: a Linux kernel fuzz tester.* Linux Conf Australia. 2013.

[27] M. Jones. „What really happened on mars rover pathfinder“. In: *The Risks Digest* 19.49 (1997), Seiten 1–2.

[28] *JPDA Overview*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jpda.html (besucht am 2016-10-06).

[29] G. A. Kanawati, N. A. Kanawati und J. A. Abraham. „FERRARI: A flexible software-based fault and error injection system“. In: *Computers, IEEE Transactions on* 44.2 (1995), Seiten 248–260.

[30] W.-L. Kao, R. K. Iyer und D. Tang. „FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults“. In: *Software Engineering, IEEE Transactions on* 19.11 (1993), Seiten 1105–1118.

[31] M. Kerrisk. *LCA: The Trinity fuzz tester*. 6. Feb. 2013. URL: `https://lwn.net/Articles/536173/` (besucht am 2015-05-22).

[32] T. M. Khoshgoftaar und J. C. Munson. „Predicting software development errors using software complexity metrics“. In: *Selected Areas in Communications, IEEE Journal on* 8.2 (1990), Seiten 253–261.

[33] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier und J. Irwin. „Aspect-oriented programming“. In: *European conference on object-oriented programming*. Springer. 1997, Seiten 220–242.

[34] G. Klein. „Operating system verification – an overview“. In: *Sadhana* 34.1 (2009), Seiten 27–69.

[35] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish u. a. „seL4: Formal verification of an OS kernel“. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, Seiten 207–220.

[36] P. Koopman, J. Sung, C. Dingman, D. Siewiorek und T. Marz. „Comparing operating systems using robustness benchmarks“. In: *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*. IEEE. 1997, Seiten 72–79.

[37] A. J. Kornecki und J. Zalewski. „Aviation Software: Safety and Security“. In: *Wiley Encyclopedia of Electrical and Electronics Engineering* (2015).

[38] J.-C. Laprie. „Dependable computing and fault-tolerance“. In: *Digest of Papers FTCS-15* (1985), Seiten 2–11.

[39] C. Lattner und V. Adve. „LLVM: A compilation framework for lifelong program analysis & transformation“. In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, Seiten 75–86.

[40] M. Lipow. „Number of faults per line of code“. In: *Software Engineering, IEEE Transactions on* 4 (1982), Seiten 437–439.

[41] P. D. Marinescu und G. Candea. „LFI: A practical and general library-level fault injector“. In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE. 2009, Seiten 379–388.

[42] E. Martin und T. Xie. „A fault model and mutation testing of access control policies“. In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, Seiten 667–676.

[43] E. Martins, C. M. Rubira und N. G. Leme. „Jaca: A reflective fault injection tool based on patterns“. In: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE. 2002, Seiten 483–487.

[44] G. C. Necula, S. McPeak, S. P. Rahul und W. Weimer. „CIL: Intermediate language and tools for analysis and transformation of C programs“. In: *International Conference on Compiler Construction*. Springer. 2002, Seiten 213–228.

[45] J. Offutt, R. Alexander, Y. Wu, Q. Xiao und C. Hutchinson. „A fault model for subtype inheritance and polymorphism“. In: *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*. IEEE. 2001, Seiten 84–93.

[46] T. Ormandy. *An empirical study into the security exposure to hosts of hostile virtualized environments*. 2007.

[47] R. Osherove. *The Art of Unit Testing*. 2nd edition. Shelter Island, NY: Manning, 5. Dez. 2013. 266 Seiten.

[48] J. Passing, A. Schmidt, M. von Löwis und A. Polze. „NTrace: Function boundary tracing for Windows on IA-32“. In: *2009 16th Working Conference on Reverse Engineering*. IEEE. 2009, Seiten 43–52.

[49] D. E. Perry und W. M. Evangelist. „An empirical study of software interface faults – an update“. In: *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*. Band 2. 1987, Seiten 113–126.

[50] V. Pratt. „Anatomy of the Pentium bug“. In: *Colloquium on Trees in Algebra and Programming*. Springer. 1995, Seiten 97–107.

[51] A. Pretschner, D. Holling, R. Eschbach und M. Gemmar. „A generic fault model for quality assurance“. In: *Model-Driven Engineering Languages and Systems*. Springer, 2013, Seiten 87–103.

[52] R. Said. *Cool Tools: Fault Injection into Unit Tests with JBoss Byteman - Easier Testing of Error Handling*. The Holy Java. 25. Feb. 2012. URL: `https://theholyjava.wordpress.com/2012/02/25/cool-tools-fault-injection-into-unit-tests-with-jboss-byteman-easier-testing-of-error-handling/` (besucht am 2016-10-06).

[53] B. C. Smith. „The limits of correctness“. In: *ACM SIGCAS Computers and Society* 14.1 (1985), Seiten 18–26.

[54] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu und W.-m. W. Hwu. „Parboil: A revised benchmark suite for scientific and commercial throughput computing“. In: *Center for Reliable and High-Performance Computing* 127 (2012).

[55] M. Sullivan und R. Chillarege. „Software defects and their impact on system availability: A study of field failures in operating systems“. In: *FTCS*. 1991, Seiten 2–9.

[56] A. S. Tanenbaum, J. N. Herder und H. Bos. „Can we make operating systems reliable and secure?“ In: *IEEE Computer* 39.5 (2006), Seiten 44–51.

[57] P. Tröger, L. Feinbube und M. Werner. „What activates a bug? A refinement of the Laprie terminology model“. In: *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE. 2015.

[58] S. Zhang und J. Zhao. „On identifying bug patterns in aspect-oriented programs“. In: *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*. Band 1. IEEE. 2007, Seiten 431–438.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
108	978-3-86956-377-0	**Improving Hosted Continuous Integration Services**	Christopher Weyand, Jonas Chromik, Lennard Wolf, Steffen Kötte, Konstantin Haase, Tim Felgentreff, Jens Lincke, Robert Hirschfeld
107	978-3-86956-373-2	**Extending a dynamic programming language and runtime environment with access control**	Philipp Tessenow, Tim Felgentreff, Gilad Bracha, Robert Hirschfeld
106	978-3-86956-372-5	**On the Operationalization of Graph Queries with Generalized Discrimination Networks**	Thomas Beyhl, Dominique Blouin, Holger Giese, Leen Lambers
105	978-3-86956-360-2	**Proceedings of the Third HPI Cloud Symposium "Operating the Cloud" 2015**	Estee van der Walt, Jan Lindemann, Max Plauth, David Bartok (Hrsg.)
104	978-3-86956-355-8	**Tracing Algorithmic Primitives in RSqueak/VM**	Lars Wassermann, Tim Felgentreff, Tobias Pape, Carl Friedrich Bolz, Robert Hirschfeld
103	978-3-86956-348-0	**Babelsberg/RML : executable semantics and language testing with RML**	Tim Felgentreff, Robert Hirschfeld, Todd Millstein, Alan Borning
102	978-3-86956-347-3	**Proceedings of the Master Seminar on Event Processing Systems for Business Process Management Systems**	Anne Baumgraß, Andreas Meyer, Mathias Weske (Hrsg.)
101	978-3-86956-346-6	**Exploratory Authoring of Interactive Content in a Live Environment**	Philipp Otto, Jaqueline Pollak, Daniel Werner, Felix Wolff, Bastian Steinert, Lauritz Thamsen, Macel Taeumel, Jens Lincke, Robert Krahn, Daniel H. H. Ingalls, Robert Hirschfeld
100	978-3-86956-345-9	**Proceedings of the 9th Ph.D. retreat of the HPI Research School on service-oriented systems engineering**	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich (Hrsg.)

ISBN 978-3-86956-386-2
ISSN 1613-5652