TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4

Verteilte Systeme und Betriebssysteme

Simon Schuster

Ein Kontrollflussüberwachungsdienst für KESO Anwendungen

Bachelorarbeit im Fach Informatik

1. Juni 2015

Please cite as:
Simon Schuster, “Ein Kontrollflussüberwachungsdienst für KESO Anwendungen,” Bachelor’s Thesis (Studienarbeit), University of Erlangen, Dept. of Computer Science, June 2015.

Friedrich-Alexander-Universität Erlangen-Nürnberg

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme

Martensstr. 1 · 91058 Erlangen · Germany

www4.cs.fau.de

Ein Kontrollflussüberwachungsdienst für KESO Anwendungen

Bachelorarbeit im Fach Informatik

vorgelegt von

Simon Schuster

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dr.-Ing. Peter Ulbrich,** **Dipl.-Inf. Isabella Stilkerich,** **Dipl.-Inf. Christoph Erhardt**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**1. Januar 2015**
Abgabe der Arbeit:	**1. Juni 2015**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Simon Schuster)
Erlangen, 1. Juni 2015

Abstract

The increasing usage of embedded systems for safety-critical tasks in adverse environments requires the usage of fault detection and fault tolerance mechanisms to tackle the issue of transient faults. The KESO project is an implementation of the Java VM for statically-configured, deeply embedded systems. In this context however, the primary target, microcontrollers, often lack the desired hardware based fault tolerance measures. A subset of these transient faults, the control flow errors, cause derivations in a programs control flow. In the past, various control flow checking approaches have been proposed to improve upon this situation, including Plain Inter-Block Error Detection [1], [2], Enhanced Control-flow Checking using Assertions (ECCA) [3], Control Flow Checking by Software Signatures (CFCSS) [4], Yet Another Control flow Checking Approach (YACCA) [5], [6] and an dominatorbased approach [7]. In the course of this work, those methods were implemented in the KESO project and their fitness to the specific necessities of this environment is discussed.

During this analysis, the commonly used comparison metric fault coverage is refuted and a new metric based on the absolute count of undetected errors resulting of a full scan of the fault space is established. Using this measure, the widely accepted assumptions on the effectiveness of softwarebased control flow checking techniques were reevaluated, yielding a more diverse picture. The measurements performed using the Fault Injection Leveraged (FAIL*) tool suite for the Intel Architecture 32 class of processors indicate three different implications. First of all the methods discussed work, but the improvements are not as effective as previous experimental evaluations using the fault coverage metric claim, in some cases they are even negligible, especially when compared to the high runtime overhead incurred ranging between 130% to 1900% depending on the method used. Second, the behaviour of a said application or application variant under injection is heavily dependent upon the assembly generated and the layout in memory, so the aptness of a specific method to a specific application cannot be determined statically. Third, control flow and data hardening may not be discussed as orthogonal goal, as both influence the fault space dimensions. Especially this disparity between a slightly reduced numbers of undetected control flow errors compared to a rapidly rising number of newly introduced cases of silent data corruption question the general idea of software implemented control flow checking techniques, at least for the IA32 discussed here.

Kurzfassung

Der zunehmende Einsatz eingebetteter Systeme für sicherheitskritische Aufgaben unter teilweise widrigen äußeren Umständen erfordert in zunehmendem Maße den Einsatz von Fehlererkennungs- und -toleranzmechanismen um die Problematik transienter Fehler abzumildern. Das KESO Projekt stellt eine Umsetzung der Java VM für statisch konfigurierte, tief eingebettete Systeme bereit. Die hier verbreiteten Mikroprozessoren lassen jedoch häufig hardwarebasierte Fehlertoleranzmechanismen missen. Eine Untermenge der transienten Fehler, die Kontrollflussfehler, erzeugen eine Abweichung vom eigentlich im Programm vorhandenen Kontrollfluss. In der Vergangenheit wurden mehrere Ansätze zur Bekämpfung dieser Problematik vorgeschlagen, unter anderem Plain Inter-Block Error Detection [1], [2], Enhanced Control-flow Checking using Assertions (ECCA) [3], Control Flow Checking by Software Signatures (CFCSS) [4], Yet Another Control flow Checking Approach (YACCA) [5], [6] und ein dominatorbasierter Ansatz [7]. Im Laufe dieser Arbeit wurden diese Verfahren in KESO umgesetzt und ihre Eignung in diesem Kontext diskutiert.

Während dieser Analyse wird dabei die verbreitete Vergleichsmetrik der Fehlerabdeckung widerlegt und stattdessen ein neues, auf den absoluten unentdeckten Fehlerzahlen einer vollen Abtastung des Fehlerraumes basierendes Vergleichkriterium etabliert. Eine vor diesem Hintergrund durchgeführte Neubewertung der Effektivität softwarebasierter Kontrollflusshärtung zeigt gemischte Ergebnisse. Die hierzu mittels FAIL* für die IA32-Prozessorfamilie durchgeführten Injektionsexperimente legen dabei vor allem drei Schlüsse nahe. Einerseits zeigt sich die allgemeine Wirksamkeit der implementierten Verfahren, auch wenn die Verbesserung deutlich schwächer ausfällt, als die in der Literatur durch Fehlerabdeckung ausgewerteten Vergleichsexperimente erwarten lassen, teilweise sind sie sogar vernachlässigbar gering, besonders im Vergleich zu den damit einhergehenden hohen Laufzeitkosten zwischen 130% bis 1900% je nach Verfahren. Andererseits wird das Verhalten einer bestimmten Anwendung oder Anwendungsvariante unter einer Fehlerinjektion maßgeblich durch das erzeugte Maschinenprogramm und die zugehörige Anordnung im Speicher beeinflusst, eine statische Einstufung der Verfahrenseignung ist nicht möglich. Drittens dürfen Kontroll- und Datenhärtung nicht als orthogonale Probleme diskutiert werden, beide nehmen Einfluss auf die Dimensionen des gemeinsamen Fehlerraumes. Besonders dieses Missverhältnis zwischen mäßig sinkenden Fehlerzahlen unentdeckter Kontrollflussfehler und rapide wachsender unentdeckter Datenfehler stellen die softwarebasierte Kontrollflussüberwachung generell in Frage, zumindest auf der in dieser Arbeit diskutierten IA32.

Inhaltsverzeichnis

Abstract **iii**

Kurzfassung **v**

1 Einleitung **1**
1.1 Motivation 1
1.2 Die KESO-Umgebung 2
1.3 Aufbau dieser Arbeit 3

2 Grundlagen **5**
2.1 Transiente Fehler 5
2.2 Kontrollflussfehler 6
2.2.1 Kontrollflussfehler 6
2.2.2 Basisblöcke 6
2.2.3 Kontrollflussgraph 7
2.2.4 Kategorisierung 8
2.2.5 Verwendetes Fehlermodell 9
2.2.6 Kontrollflussüberwachung 10
2.3 Resümee 10

3 Problemanalyse und Umsetzung **11**
3.1 Auswahl der Verfahren 11
3.2 Allgemeine Rahmenbedingungen 12
3.3 Umsetzung 14
3.3.1 Allgemeines 14
3.3.2 Plain Inter-Block Error Detection 16
3.3.2.1 Verfahren 16
3.3.2.2 Einstufung 17
3.3.2.3 Anpassungen auf KESO 18
3.3.3 Enhanced Control flow Checking using Assertions – ECCA 19
3.3.3.1 Verfahren 19

3.3.3.2 Einstufung 21
3.3.3.3 Anpassungen auf KESO 21
3.3.4 Control Flow Checking by Software Signatures – CFCSS 23
3.3.4.1 Verfahren 23
3.3.4.2 Einstufung 26
3.3.4.3 Anpassungen auf KESO 27
3.3.5 Yet Another Controlflow Checking using Assertions – YACCA . . . 28
3.3.5.1 Verfahren 28
3.3.5.2 Einstufung 34
3.3.5.3 Anpassungen auf KESO 35
3.3.6 Dominatorbasierter Ansatz 36
3.3.6.1 Grundlagen: Dominanz und Dominanzgrenzen 36
3.3.6.2 Verfahren 37
3.3.6.3 Einstufung 40
3.3.6.4 Umsetzung in KESO 40
3.4 Resümee 43

4 Evaluation und Diskussion 45
4.1 Testaufbau 46
4.1.1 Rahmenbedingungen 48
4.1.2 Laufzeit und Größe 49
4.2 Fehlerinjektion 52
4.2.1 Die FAIL* - Fehlerinjektionsumgebung 52
4.2.2 Fehlermodell und Auswertung 53
4.2.3 Metriken: Fehlerrate und -abdeckung 55
4.3 Messergebnisse und Auswertung 58
4.3.1 Kontrollflussfehler 59
4.3.1.1 Matrixmultiplikation 59
4.3.1.2 Sortierung 64
4.3.1.3 Zustandsmaschine 68
4.3.1.4 Zusammenfassung 72
4.3.2 Allgemeine Einbitfehler 75
4.4 Diskussion 76
4.5 Resümee 78

5 Fazit 79

Literaturverzeichnis 87

Kapitel 1

Einleitung

1.1 Motivation

Durch den zunehmenden Einsatz eingebetteter Systeme in immer mehr Aufgabenfeldern, von Haushaltselektronik über Unterhaltungselektronik hin zu Regel- und Steuerungstechnik ergeben sich immer höhere Ansprüche an die Zuverlässigkeit derartiger Systeme. Gleichzeitig lassen der hohe Kostendruck und die geringe Rechenleistung dieser Systeme klassische Verfahren der Zuverlässigkeitssicherung unattraktiv erscheinen [8].

Gleichsam sind durch die im Sinne einer Energiebedarfsoptimierung an integrierten Schaltungen durchgeführten Modifikationen wie beispielsweise die immer weiter sinkenden Strukturbreiten der Prozessoren und Logikschaltungen eine potentiell höhere Anfälligkeit der Schaltung gegenüber widrigen äußeren Einflüssen und somit eine weitere Zunahme der daraus resultierenden *transienten Fehler* zu erwarten [9].

Zur Datenhärtung sind fehlertolerante Kodierungen sowohl in Hardware in Form von fehlerbehebendem Speicher (engl. *ECC* Memory) als auch in Software etabliert.

Solche Ansätze finden sich auch in KESO, einer statisch konfigurierbaren Javalaufzeitumgebung für *tief eingebettete Systeme* (engl. *deeply embedded systems*), bspw. in Form von Replikationsmechanismen oder der Mitführung von Paritätsinformation für bestimmte besonders kritische Datenstrukturen wie Objektreferenzen. Physikalische Fehlerinjektionsexperimente, sowohl durch Strahlungseinwirkung als auch durch bewusste Eingriffe in die Versorgungsspannung an einem MC6809E 8-bit Mikrocontroller legen jedoch nahe, dass bis zu 90% der auftretenden Fehler im Prozessor vor allem eine nach außen beobachtbare Änderung des Kontrollflusses bewirken [10]. Eine Härtung der Programme auch gegen diese Fehlergattung scheint wünschenswert.

Da die KESO-Umgebung im Sinne der Konfigurierbarkeit auf eine Vielzahl unterschiedlicher Prozessorarchitekturen portabel ist, welche mehrheitlich über keinerlei hardwarebasierte Fehlertoleranzmechanismen verfügen, ist hier eine softwarebasierte Lösung geboten. In der Literatur finden sich eine Vielzahl solcher Verfahren zur Kontrollflusshärtung. Im

Laufe dieser Arbeit werden im speziellen die Mechanismen Plain Inter-Block Error Detection [1], [2], Enhanced Control-flow Checking using Assertions (ECCA) [3], Control Flow Checking by Software Signatures (CFCSS) [4], Yet Another Control flow Checking Approach (YACCA) [5], [6] und ein dominatorbasiertes Verfahren [7] vor allem vor dem Hintergrund ihrer Eignung zur automatisierten Härtung von Javaprogrammen betrachtet.

In der Vergangenheit durchgeführte Experimente lassen hier einen Einfluss der gewählten Programmiersprache auf die Eignung von Kontrollflussüberwachung im Allgemeinen und bestimmter Verfahren im Besonderen erkennen [10]. Dementsprechend stellt sich die Frage nach der Effektivität und Effizienz der jeweiligen Verfahren im spezifischen Umfeld des KESO-Projekts.

1.2 Die KESO-Umgebung

Bei der KESO-Laufzeitumgebung [12] handelt es sich um eine Implementierung der *virtuellen Java Maschine* (engl. *Java Virtual Machine*, JVM) für Anwendungen in tief eingebetteten Systemen, welche auf OSEK oder AUTOSAR-kompatiblen Echtzeitbetriebssystemen aufsetzt. Üblicherweise operieren JVMs als Interpreter oder *Laufzeitübersetzer* (engl. *Just-in-Time Compiler*) auf einer besonderen Zwischensprache, dem sog. Bytecode. Abweichend von dieser Praxis wird in KESO diese Zwischensprache bereits zur Übersetzungszeit durch die zugehörige Übersetzungskomponente JINO in OSEK-kompatible C oder C++ Anwendungen transformiert. Diese vorgezogene Übersetzung ermöglicht eine umfassende, statische

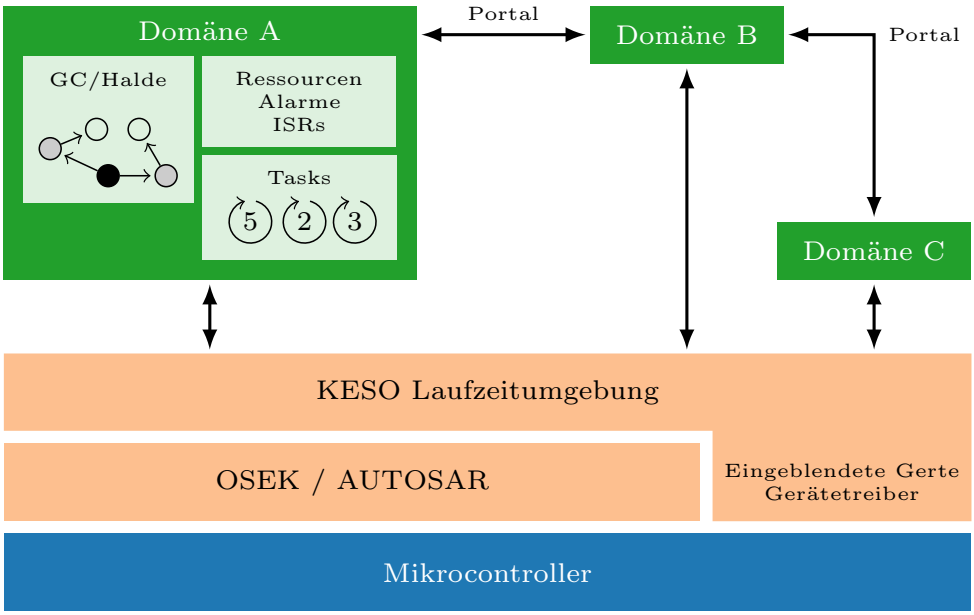


Abbildung 1.1 – Schematischer Überblick des KESO-Systems (Quelle: [11])

Optimierung des Programmes. Aufgrund dieses Umstandes wird in KESO auf die Unterstützung bestimmter, vor diesem Hintergrund nur schwer realisierbarer Funktionalitäten wie dynamischem Nachladen, Laufzeittypinformationen (engl. *Reflections*) oder auch eine Ausnahmebehandlung verzichtet, es wird also lediglich eine große Untermenge der Programmiersprache implementiert. Gleichzeitig wird durch die Übersetzung in diese im eingebetteten Bereich weit verbreiteten Zielsprachen und Betriebssystemschnittstellen ein hoher Grad der Portabilität erreicht.

Durch die Typsicherheit der Programmiersprache wird dabei softwarebasierter Speicherschutz der Anwendungen gewährleistet. In Abbildung 1.1 findet sich eine Darstellung der zugrundeliegenden Architektur. OSEK kennt hier ein dem traditionellen Prozessmodell verwandtes Konzept der *Tasks* [13, S. 16] zur Abgrenzung unterschiedlicher Aktivitäten innerhalb des Systems, welches in KESO übernommen wurde. Jeder Task ist dabei in KESO einer Domäne zugeordnet, die einen Schutzraum im Sinne des Speicherschutzes realisiert. Alle Anwendungen einer Domäne können auf gemeinsame Objekte und Daten zugreifen. Für die Kommunikation über Domänengrenzen hinweg sind explizite Kommunikationskanäle, die sog. Portale vorgesehen.

Weiterhin ermöglicht die KESO-Umgebung die Umsetzung von Gerätetreibern bzw. das Ansprechen von Geräten in der Sprache Java selbst. Da jedoch die Programmiersprache Java als solche hierfür keine Schnittstelle bereitstellt, bietet KESO an diesen Stellen über sogenannte *Einwebungen* (engl. *Weavelets*), vergleichbar zum Java Native Interface (JNI), die Möglichkeit, selektiv nativen C-Quelltext einzubringen.

1.3 Aufbau dieser Arbeit

Im Sinne einer Studie gliedert sich diese Arbeit in drei Abschnitte. In Kapitel 2 erfolgt kurz eine Einführung der Grundlagen. Auf der Betrachtung *transienter Fehler* aufbauend wird das dieser Arbeit zugrundeliegende Fehlermodell erarbeitet und daraus die allgemeinen Eigenschaften und Anforderungen an eine Kontrollflussüberwachung abgeleitet. Im Anschluss erfolgt in Kapitel 3 eine Betrachtung der Besonderheiten der KESO-Laufzeitumgebung und der sich daraus ableitenden Rahmenbedingungen für die ebenfalls in diesem Kapitel beschriebenen, in dieser Arbeit umgesetzten Verfahren. Dabei wird für jedes der Verfahren zunächst die in der Literatur beschriebene Reinform dargestellt und eine kurze Einordung der theoretischen Detektionsfähigkeiten gegeben, bevor jeweils abschließend die eigentliche Umsetzung und die hierfür nötigen Einschränkungen bzw. Modifikationen beschrieben werden. Zum Schluss werden die so entstandenen, konkreten Ausprägungen der Verfahren in Kapitel 4 einer umfassenden Analyse, sowohl bezüglich Speicher- und Laufzeitkosten als auch bezüglich des Nutzes im Sinne einer Verbesserung des Verhaltens unter den Einflüssen transienter Kontrollflussfehler unterzogen. Hierzu werden zunächst in Abschnitt 4.2.3 die Unzulänglichkeiten bisher gebräuchlicher Metriken zum Vergleich der Detektionsfähigkeiten basierend auf der *Fehlerabdeckung* (engl. *fault coverage*) aufgezeigt. Aus dieser Betrachtung wir eine

zuverlässigere Vergleichsmetrik für die Ergebnisse aus Fehlerinjektionsexperimenten abgeleitet, diese wird im Anschluss auf die Resultate einer mittels Fault Injection Leveraged (FAIL*) durchgeführten Versuchsreihe angewandt. Die sich daraus ableitenden Konsequenzen werden dann in Abschnitt 4.4 diskutiert, bevor darauf aufbauend in Kapitel 5 dann abschließend die eingangs gestellte Frage nach der Effektivität und Effizienz der jeweiligen Verfahren im spezifischen Umfeld des KESO-Projekts beantwortet wird.

Kapitel 2

Grundlagen

In informationstechnischen Systemen existiert eine Vielzahl unterschiedlicher Fehlertypen, die sich alle je nach Ursache, Ausprägung und Effekten unterscheiden. Zur Entwicklung von Fehlerdetektionsmechanismen ist nun zunächst eine klare Definition und Klassifikation der durch den jeweiligen Mechanismus zu betrachtenden Fehlertypen nötig, da nur so eine sinnvolle Optimierung und Evaluation der Verfahren möglich wird. Hierzu wird in Abschnitt 2.1 zunächst eine Abgrenzung der in dieser Arbeit thematisierten transienten Fehler gegeben, bevor dann in Abschnitt 2.2 daraus eine Definition der Kontrollflussfehler abgeleitet wird und die Grundlagen einer Kontrollflussüberwachung eingeführt werden.

2.1 Transiente Fehler

Durch äußere Einflüsse bedingte Fehler in informationstechnischen Geräten werden für diese Arbeit gemäß ihrem Effekt in zwei wichtige Gruppen aufgeteilt, die *permanenten* und *transienten* Fehler. Während erstere die Leistungsfähigkeit des Systems dauerhaft beeinträchtigen, handelt es sich bei letzteren um vorübergehend auftretende, flüchtige Erscheinungen, ausgelöst unter anderem durch die mangelnde Präzision in der Fertigung, Induktionserscheinungen aufgrund elektromagnetischer Strahlung, Schwankungen in der Stromversorgung oder auch ionisierender Strahlung, z.B. kosmischer Strahlung oder verunreinigte Baugruppenelemente (engl. *packaging*) [14].

Dabei überwiegt die Häufigkeit transienter Fehler die der permanenten deutlich, Messungen 1982 ergaben einen Faktor von 15 bis 50 [14], aufgrund der heute deutlich geringeren Strukturbreiten, steigenden Taktfrequenzen und des höheren Integrationsgrades verbunden mit immer weiter sinkenden Versorgungsspannungen dürfte dieser Faktor mittlerweile weitaus höher liegen. Während nach einem permanenten Fehler eine korrekte Funktion der elektrischen Komponenten nicht mehr garantiert werden kann ist diese jedoch bei transienten Fehlern noch gegeben, bei Detektion ist ein Zurücksetzen in einen sicheren Zustand und eine von dort beginnende Neuausführung möglich.

2.2 Kontrollflussfehler

Je nach den beobachtbaren Auswirkungen lassen sich diese Fehler in Daten- und Kontrollflussfehler untergliedern.

2.2.1 Kontrollflussfehler

Die Ausführung einer Anwendung bedingt, gleiche Eingaben vorausgesetzt, eine eindeutige Instruktionsfolge. Durch Abweichungen davon lassen sich Kontrollflussfehler definieren:

Definition 1. Ein *Kontrollflussfehler* (engl. *Control Flow Error*, CFE) tritt immer dann auf, wenn der Prozessor aufgrund eines Fehlers eine falsche, das heißt von der fehlerfreien Ausführung abweichende, Sequenz von Befehlen ausführt [15].

Werden hingegen durch den Defekt die von der Anwendung verrechneten Daten beeinflusst, so handelt es sich bei dem Fehler um einen *Datenfehler*. Dabei sind Datenfehlern und Kontrollflussfehlern jedoch keineswegs Komplemente. Vielmehr besitzen die Fehlerklassen eine Schnittmenge, immer dann, wenn das durch den Fehler modifizierte Datum Einfluss auf den weiteren Kontrollfluss nimmt, indem es in einer Bedingung ausgewertet oder als Adresse oder Index für einen Kontrollflussübergang herangezogen wird.

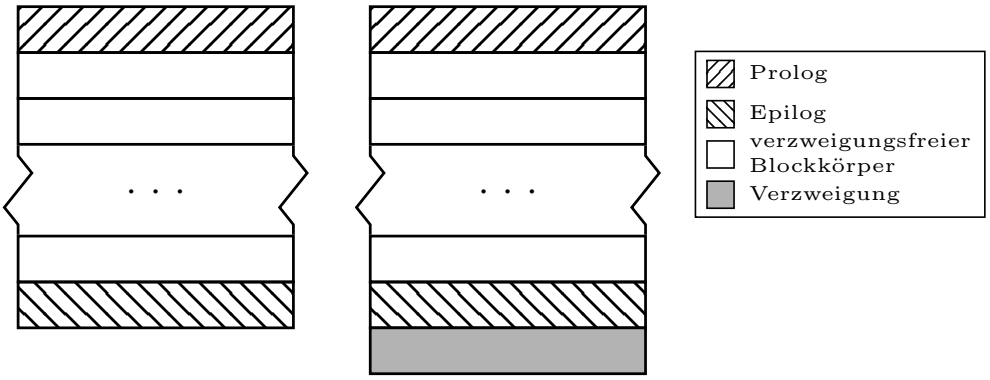
2.2.2 Basisblöcke

Diese Definition des Kontrollflusses als vom Prozessor ausgeführten Instruktionsstrom ist jedoch für die effektive, algorithmische Handhabung aufgrund der zu feingranularen Betrachtung auf Instruktionsebene ungeeignet. Üblich ist es, Mengen immer sequentiell durchlaufender Instruktionen zu *Basisblöcken* zu gruppieren:

Definition 2. „*Basisblöcke* sind maximale Sequenzen aufeinanderfolgender [...] Instruktionen mit der Eigenschaft, dass der Kontrollfluss den Basisblock nur durch die erste Instruktion des Blocks betreten kann [... und] den Block ohne Halten oder Verzweigen verlässt, mit Ausnahme der letzten Instruktion“ [16, S.525].

Diese Blöcke bilden gewissermaßen die grundlegende Granularität der Kontrollflussdarstellung, da nur jeweils am Ende eines Basisblocks eine Kontrollflussentscheidung getroffen wird.

Oft ist es nötig, bestimmte Instruktionen am Anfang oder am Ende eines Basisblocks zu platzieren. Da jedoch besonders der Begriff des Basisblockendes im Falle eines mit einer Verzweigung abschließenden Basisblocks nur unpräzise definiert ist, wird in dieser Arbeit der Begriff des Prologs und Epilogs verwendet, in Abbildung 2.1 findet sich eine Darstellung der verwendeten Positionen. Während die Position des Prologs mit dem Beginn des Basisblocks übereinstimmt, bezeichnet der Epilog für einen verzweigungsfreien Basisblock die hinter



(a) verzweigungsfreier Basisblock **(b)** verzweigender Basisblock

Abbildung 2.1 – Platzierung von Prolog und Epilog bei verzweigenden und verzweigungsfreien Basisblöcken

der letzten Instruktion liegende Region, ansonsten die auf die vorletzte Instruktion folgende Stelle im Basisblock.

2.2.3 Kontrollflussgraph

Die Basisblöcke bilden die atomaren Bestandteile des *Kontrollflussgraphen*. Dieser gerichtete Graph besteht aus der Knotenmenge B, welche der Menge der Basisblöcke entspricht, und der Kantenmenge $K \subseteq \{B \times B\}$. Der Graph beinhaltet eine Kante von Block BB_{vor} zum Block BB_{nach} genau dann, wenn BB_{vor} mit einem bedingten oder unbedingten Sprung nach BB_{nach} endet oder BB_{nach} direkt an BB_{vor} anschließt und dieser nicht auf einen unbedingten Sprung endet [16, S.529].

Dabei heißt BB_{nach} *Nachfolger* von BB_{vor} und BB_{vor} ist *Vorgänger* von BB_{nach}. In Abbildung 2.2 findet sich ein Beispiel für den Kontrollflussgraphen eines Javaprogrammes.

Diese Definition des Kontrollflussgraphen entstammt mehrheitlich dem Umfeld des Übersetzerbaus und dient dort primär der Analyse innerhalb von Prozeduren. Dabei erstreckt sich der Kontrollflussgraph zwischen dem ersten Basisblock, dem *Eingang* der Prozedur, bis zu den *Ausgängen*, also den auf einer `return`-Instruktion endenden Basisblöcken und dem letzten Basisblock der Prozedur, falls die Funktion keinen Rückgabewert besitzt.

Dieser prozedurlokale Bezug ist jedoch für die Kontrollflussüberwachung nicht ausreichend, da für die meisten hier betrachteten Verfahren kein Aufrufkonzept für Methoden als solches existiert. Aufgrund der statischen Konfigurierbarkeit der KESO-Anwendungen liegen zum Übersetzungszeitpunkt alle Informationen zu den verwendeten Prozeduren und ihrer Implementierungen vor, es existiert keine separate Kompilierbarkeit. Dies ermöglicht das bilden eines *globalen Kontrollflussgraphen* (engl. *super control-flow graph*). Dabei werden Aufrufe in den Kontrollflussgraphen aufgenommen, indem Basisblockgrenzen auch an Aufrufstellen eingefügt werden. Der aufrufende Basisblock wird dabei zum Vorgänger des Eingangs der

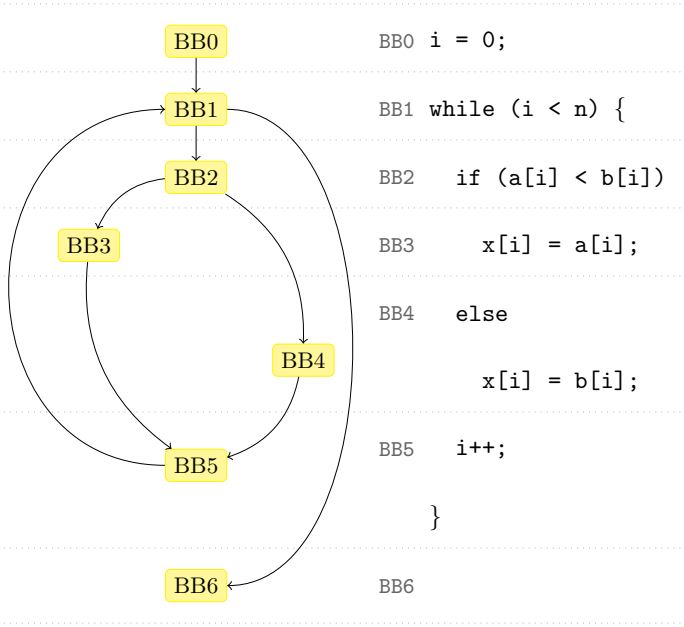


Abbildung 2.2 – Beispiel eines Kontrollflussgraphen, in Anlehnung an Goloubeva et al. [5]

aufgerufenen Prozedur, der auf den Aufruf folgende Basisblock wird zum Nachfolger der Ausgänge der Prozedur, die Methodenkontrollflussgraphen werden beim Aufruf also in den gesamten Kontrollflussgraphen „eingebettet“ [16, S.907].

Dadurch ist es möglich den gesamten Kontrollfluss der Anwendung in einem einzigen Graphen abzubilden. Gleichzeitig wird hierdurch für die weitere Analyse die konkrete Ausprägung des Aufrufs abstrahiert. Soweit nicht anders angegeben verwendet diese Arbeit den globalen Kontrollflussgraphen als Kontrollflussgraphen, auch die Definition der Begriffe Vorgänger und Nachfolger bezieht sich auf diesen globalen Graphen.

2.2.4 Kategorisierung

Korrektheitsüberprüfungen stellen in der Regel die Übereinstimmung des beobachteten Kontrollflusses mit dem im eigentlichen Programm vorhandenen Ablauf sicher. Gemäß des Übergangs im Kontrollflussgraphen lässt sich jeder Abweichung hierbei ein Typ zuweisen [6, S. 67], die Unterscheidung wird hier anhand von Ursprungs- und Zielort geführt. Die Fehler der Kategorien eins, zwei und drei entspringen alle im Ende eines Basisblocks. Sowohl bei *Typ eins* als auch bei *Typ zwei* führt dabei der Übergang zum Beginn eines anderen Basisblocks, die beiden Fehlerklassen differenzieren sich im Bezug auf ihrer *Legalität*. Ein Übergang zwischen zwei Knoten im Kontrollflussgraphen ist legal, falls er mit einer Kante des Graphen übereinstimmt. Während sich bei den Fehlern der *ersten Kategorie* keine Entsprechung im Kontrollflussgraphen findet, so stellt der durch einen Fehler *zweiter Art*

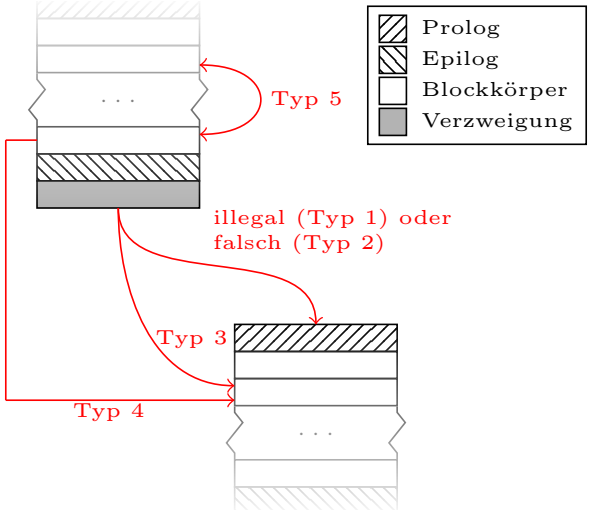


Abbildung 2.3 – Fehlerkategorisierung gemäß Goloubeva et al. [6]

ausgelöste, veränderte Programmfluss einen legalen, aber falschen Übergang dar. Ein Beispiel für eine solche Abweichung ist eine bedingte Sprunganweisung, welche aufgrund eines auftretenden Fehlers eine unzulässige Sprungentscheidung trifft. Zwar existiert hier ein entsprechender Übergang im Kontrollflussgraphen, jedoch wird der ursprüngliche Kontrollfluss fälschlicherweise verlassen. Fehler der *Kategorie drei* schließlich bezeichnen Abweichungen vom Ende eines Basisblocks zur Mitte eines anderen.

Weiterhin kann ein Kontrollflussfehler in der Mitte eines Basisblocks auftreten, z.B. durch eine unzulässige Änderung des Instruktionszeigers oder das Entstehen einer Sprunginstruktion aus einer regulären Instruktion aufgrund eines transienten Fehlers in der Befehlskodierung. Führt die Abweichung wieder in den ursprünglichen Basisblock zurück, so spricht man von einem Fehler des *Typs fünf*, ansonsten von Abweichungen der *Kategorie vier*. In Abbildung 2.3 findet sich eine graphische Darstellung aller hier aufgeführter Fehlerkategorien.

2.2.5 Verwendetes Fehlermodell

Für die Entwurfsentscheidung und auch Auswertung eines Härtungsverfahrens gegen transiente Fehler ist dabei wichtig festzulegen, gegen welche Fehlerarten geschützt werden soll, nur so ist eine effektive Umsetzung möglich.

Das Fehlermodell dieser Arbeit basiert hier auf folgenden Annahmen: Das Textsegment und somit der Programmtext befindet sich in einem *nur lesbaren Speicherbereich* (engl. *Read Only Memory (ROM)*), welcher gesondert gegen Fehler gehärtet ist und somit als sicher angenommen wird. Diese Härtung schützt jedoch lediglich gegen permanente Änderungen, beim Abruf der Werte können während der Übertragung über Busse transiente Fehler in den abgerufenen Werten auftreten.

Allgemein werden stochastisch unabhängige, gleichverteilte Einbitfehler, sowohl auf den im Arbeitsspeicher vorhandenen Daten als auch in den Registern angenommen, mit den oben beschriebenen Einschränkungen für das Textsegment.

2.2.6 Kontrollflussüberwachung

Vor diesem Hintergrund ist die Aufgabe einer Kontrollflussüberwachung, die Ausführung der richtigen Instruktionsfolge durch den Prozessor sicherzustellen um im Falle von Abweichungen einen Fehler zu signalisieren.

Wie bereits in Abschnitt 1.2 erwähnt, handelt es sich bei der KESO-JVM um eine Laufzeitumgebung für statisch konfigurierbare Java-Anwendungen auf eingebetteten Systemen. Diese Anwendungen werden dementsprechend auf einer Reihe verschiedener OSEK- bzw. AutoSAR-basierter Betriebssysteme zur Ausführung gebracht. Durch die Übersetzung zur Hochsprache C wird dabei ein gewisses Maß an Portabilität, auch über unterschiedliche Rechnerarchitekturen hinweg, erreicht. Gleichzeitig schränkt dies die Möglichkeiten des Jino-Übersetzers und der durch ihn erzeugten Programme auf die durch den jeweiligen Betriebssystemstandard vorgegebene Programmierschnittstelle und die Ausdrucksmächtigkeit der erzeugten Zwischensprache C ein. Unter der Prämisse, diese Portabilität zu erhalten, kann Fehlererkennung maximal auf der Ebene der Programmierschnittstelle dieser Hochsprache erfolgen. Aus diesem Grund beschränken sich die in dieser Arbeit betrachteten Verfahren alle auf eine Korrektheitsüberprüfung des Kontrollflusses auf Ebene der Basisblöcke, d.h. es wird die Übereinstimmung des Kontrollflusses mit Pfaden im Kontrollflussgraphen überwacht.

Da die KESO-JVM bereits über diverse Verfahren zur Fehlertoleranz auf Daten, durch Absicherung von Objektreferenzen, Parität, fehlertolerante Kodierungsmechanismen und sogar komplette Replikation verfügt, liegt das Augenmerk dieser Arbeit im Wesentlichen auf dem Schutz vor statischen Kontrollflussfehlern. Dementsprechend werden all diejenigen Fehler, welche durch Datenveränderung einen Kontrollflussfehler erzeugen, also legale aber falsche Übergänge im Graphen auslösen, der Klasse der Datenfehler zugeordnet, da sich hier die Methoden der Erkennung gleichen. Das Hauptziel der Härtung ist also zunächst die Erkennung fehlerbedingt ausgelöster, illegaler Übergänge, also der statischen Kontrollflussfehler.

2.3 Resümee

Transiente Fehler stellen eine durch äußere Einflüsse bedingte fälschliche Änderung von Werten dar. Einige dieser fälschlichen Modifikationen führen zu einer Abweichung des Programmflusses von der im Kontrollflussgraphen vorgegebenen Folge der Basisblöcke. Die in dieser Arbeit betrachteten Verfahren sollen die Zahl dieser statischen Kontrollflussfehler reduzieren.

Kapitel 3

Problemanalyse und Umsetzung

Für diese Arbeit wurden zur Detektion statischer Kontrollflussfehler aus der Literatur mehrere Verfahren ausgewählt, in Abschnitt 3.1 werden die dieser Auswahlentscheidung zugrundeliegenden Hintergründe kurz erläutert. Da zwar fast alle betrachteten Verfahren in der zugeordneten Literatur für die Zielsprache C entworfen wurden und KESO zu dieser Hochsprache übersetzt wird, ist eine grundlegende Anwendbarkeit dieser Methoden sichergestellt. Allerdings besitzt der Java-Bytecode, und somit auch die von KESO daraus erzeugte Darstellung der Zwischensprache, einige Besonderheiten, welche sich nicht direkt in der durch die jeweiligen Veröffentlichungen genutzten Untermenge der Programmiersprache C abbilden lassen. Die Einflüsse dieser Abweichungen auf den Kontrollflussgraphen werden in Abschnitt 3.2 kurz erläutert, anschließend erfolgt dann in Abschnitt 3.3 die detaillierte Beschreibung der jeweiligen Verfahren.

3.1 Auswahl der Verfahren

Wie bereits erwähnt, existieren diverse Verfahren der Kontrollflussüberwachung. Die Portabilitätsanforderung für KESO schließt zunächst all diejenigen aus, welche spezielle Unterstützung im Prozessor oder der Hardware allgemein erfordern. Aus der Menge der verbleibenden, softwarebasierten Verfahren wurden anschließend fünf ausgewählt. Um eine möglichst genaue Detektion, ggf. auf Kosten der Laufzeit, zu gewährleisten wurde dabei auf eine möglichst feine Überwachungsgranularität auf Ebene der Basisblöcke geachtet. Die Auswahl erfolgte hier mit einem Augenmerk auf ansteigende Detektionsmöglichkeiten zwischen den jeweiligen Verfahren. Während das Plain Inter-Block Error Detection [1], [2] Verfahren lediglich auf der Eingrenzung von Basisblöcken basiert, ziehen Enhanced Control-flow Checking using Assertions (ECCA) [3] und Control Flow Checking by Software Signatures (CFCSS) [4] die im Übersetzer vorhandenen, statischen Kontrollflussinformationen mit in die Überprüfung ein. Dabei verwendet ECCA jedoch ein zuverlässigeres wenn auch teureres Verfahren der Vorgängerprüfung, ein Vergleich der beiden Verfahren erscheint lohnend. Über diese grund-

legende Überwachung der Ausführungsreihenfolge der Basisblöcke hinaus unterscheidet Yet Another Control flow Checking Approach (YACCA) [5], [6] bei diesen Test zwischen Basisblockeingängen und Ausgängen und erreicht somit nochmals eine feinere Granularität.

Als Gegenstück wurde darüber hinaus ein dominatorbasiertes Verfahren [7] realisiert, welches einerseits auf der gröberen Granularitätsstufe der Dominanzregionen (siehe Abschnitt 3.3.6) agiert, aber dafür andererseits durch eine effiziente Umsetzung nur geringe Laufzeitkosten fordert. Dadurch wird später in der Auswertung auch eine Bewertung der unterschiedlichen Granularitätsstufen möglich.

3.2 Allgemeine Rahmenbedingungen

Wie bereits in Abschnitt 1.2 dargelegt handelt es sich bei der zu KESO gehörigen Übersetzerkomponente JINO um einen Übersetzer für JVM-Bytecode nach C. Da in diesem Falle bereits die Quellsprache als einfach zu verarbeitende Zwischensprache konzipiert wurde, ist eine weitere, dedizierte Zwischendarstellung im Übersetzer unnötig, stattdessen entlehnt sich die übersetzerinterne Befehlsrepräsentation im Wesentlichen der Struktur des Java-Bytecodes und unterliegt dementsprechend denselben Vorteilen und Beschränkungen.

Für die Umsetzung der eigentlich für C-Untermengen konzipierten Kontrollflusshärtungsverfahren sind hier vor allem diejenigen Eigenschaften dieser Zwischensprache interessant, welche eine deutliche von C abweichende Semantik besitzen.

Der augenfälligste derartige Unterschied ist die Existenz von Objekten und diesen zugeordneter Methoden. JINO transformiert bei der Übersetzung die Methoden einer Klasse zu Funktionen, welche das aufgerufene Objekt als Eingabeparameter erwarten. Dadurch bleibt die Funktionsweise der Verfahren zunächst unbeeinflusst. Eine Sonderstellung nehmen jedoch virtuelle Aufrufe ein. Im Rahmen der Vererbung ist es hier möglich, Methoden in Unterklassen zu überschreiben, je nach Typ des Objekts ist dabei im Sinne der dynamischen, späten Bindung die Implementierung der in der Vererbungshierarchie am nahestehensten Klasse zu wählen. In einigen dieser Fälle ist es dem Übersetzer nicht möglich, statisch die korrekte Implementierung zu bestimmen, dann ist auch in KESO eine Auswahl der jeweiligen Funktion zur Laufzeit nötig. Da so jedoch wie in Abbildung 3.1 gezeigt alle möglichen Implementierungen als gültige Nachfolger im Kontrollflussgraphen eingetragen werden, sind an diesen Stellen Basisblöcke mit mehr als zwei Nachfolgern möglich. Da jedoch die hier umgesetzten Verfahren üblicherweise lediglich eine Überprüfung der Vorgänger und nicht der Nachfolger vornehmen, stellt dieser Unterschied im Allgemeinen kein Problem dar.

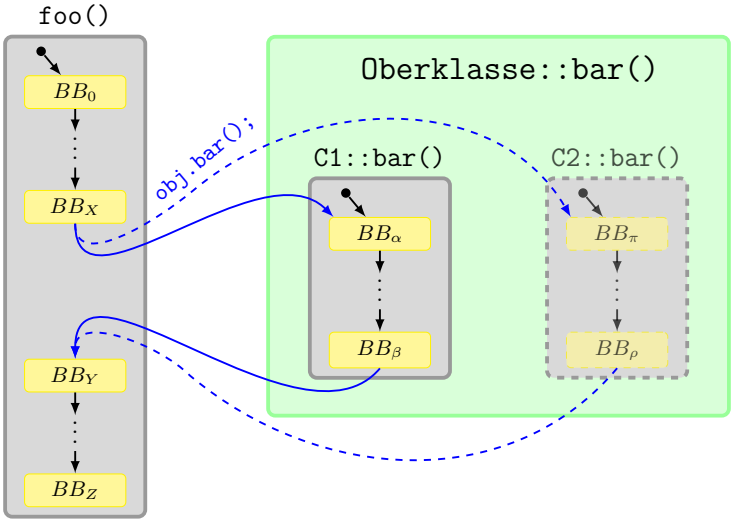


Abbildung 3.1 – Umsetzung von virtuellen Aufrufen, in Anlehnung an [7]

Auch das Konzept der *Ausnahmen* (engl. *Exceptions*) und der *Ausnahmebehandlung* in Java ist im Sinne einer statischen Kontrollflussanalyse problematisch, da hier am eigentlichen Kontrollfluss vorbei mögliche Ausnahmebehandlungen angesprungen werden können. Während diese Situation für die regulären Laufzeitausnahmen noch handhabbar ist, da deren mögliches Auftreten durch die auslösende Funktion im vornherein deklariert werden muss, trifft dies bei der Untermenge der *nicht geprüften Ausnahmen* (engl. *unchecked Exceptions*) jedoch nicht zu. Eine sinnvolle Kontrollflussanalyse ist an dieser Stelle somit nicht möglich, da derartige Ausnahmen mit jeder ausgeführten Instruktion auftreten können. Da jedoch Ausnahmen in KESO abweichend von der eigentlichen Sprachdefinition in Java eine *"Halte-Mit-Fehler"-Semantik* (engl. *fail-stop*) besitzen und ein Fangen dieser Ausnahmen nicht möglich ist, kann für diese Arbeit diese Problematik umgangen werden. Allerdings existiert mit der in KESO vorhandenen Unterbrechungsbehandlung ein in den Auswirkungen verwandtes Konstrukt: Eine Anwendung kann mitten in der Ausführung unterbrochen werden und der Kontrollfluss geht vorübergehend an eine andere Programmstelle über. Aus diesem Grund wurde für als Unterbrechungsbehandlung gekennzeichnete Funktionen die Kontrollflussüberwachung in dieser Arbeit deaktiviert.

Auch die in KESO durch die in Abschnitt 1.2 angesprochenen, mittels *Weavelets* angebundenen, nativen Bestandteile sind aus Sicht der Kontrollflussüberwachung problematisch, da die hier aufgerufenen Implementierungen auf der Ebene des Übersetzers nicht sichtbar und somit auch nicht analysierbar sind. In allen bisher in KESO für dieses Konstrukt vorhandenen Anwendungsfällen lösen diese Aufrufe jedoch keine Veränderung des Kontrollflusses innerhalb eines Tasks aus und können dementsprechend für die Kontrollflussüberwachung einfach als komplexe, aber verzweigungsfreie Instruktionen übergangen werden.

3.3 Umsetzung

Im Folgenden wird nun nach einigen kurzen allgemeinen Bemerkungen zur Art der Umsetzung in Abschnitt 3.3.1 eine vollständige Beschreibung der jeweiligen Verfahren gegeben. Diese Beschreibungen erläutern dabei zunächst das allgemeine Verfahren und seine theoretischen Detektionsmöglichkeiten, bevor die bei der Umsetzung des jeweiligen Verfahrens für KESO getroffenen Anpassungen angegeben werden.

3.3.1 Allgemeines

Da alle hier beschriebenen Verfahren sowohl die Buchführung über den bisherigen Kontrollfluss als auch die Überprüfung durch Instrumentierung im eigentlichen Instruktionsstrom der Anwendung selbst platzieren, werden die für die einzelnen Verfahren notwendigen Befehle in den Prologen oder Epilogen der Basisblöcke einfach direkt an den entsprechenden Stellen des für die Anwendung generierten C-Quelltextes emittiert.

Wie in Abschnitt 1.2 erwähnt, kennt die KESO-Umgebung das Konzept der OSEK-Tasks für die pseudoparallele oder parallele Ausführung unterschiedlicher Aktivitäten oder gar Anwendungen im Sinne eines Prozesskonzeptes. Um diese parallelen Kontrollflüsse umzusetzen erfolgt eine Kontrollflussüberwachung jeweils nur auf der Ebene dieser Tasks, jedoch nicht zwischen diesen. Die im Folgenden betrachteten Verfahren gehören alle zur Gruppe der signaturbasierten Verfahren. Hier wird in einer oder mehreren dedizierten Variablen eine Signatur mitgeführt, welche Informationen über den momentanen Zustand des Kontrollflusses liefert. Um die pseudoparallele Ausführung mehrerer Tasks zu ermöglichen, wird diese Variable, wie auch alle sonstigen verfahrensspezifischen Daten im zum Task gehörigen Deskriptor abgelegt. Da der für KESO-Anwendungen erzeugte C-Quelltext in der Regel mit im eingebetteten Bereich üblichen Optimierungseinstellungen übersetzt wird, wurden diese Signaturfelder als `volatile` markiert, um zu aggressive, datenflussgesteuerte Optimierungen an dieser Variable durch den C-Übersetzer zu verhindern.

Weiterhin weist die Konzeption der Verfahren Unterschiede in der Behandlung von dynamischen Kontrollflussfehlen auf. Während einige der Verfahren lediglich den statischen Kontrollfluss, also die Übereinstimmung der Basisblockfolge mit dem Kontrollfluss überwachen, sehen andere Verfahren explizit eine Wiederholung von Vergleichsoperationen am jeweiligen Zielortes eines bedingten Sprunges vor, um auch dynamische Kontrollflussfehler besser detektieren zu können. Da es sich hierbei jedoch um eine zur Auswahl des eigentlichen Verfahrens orthogonale Maßnahme handelt, wurde für alle betrachteten Verfahren auf die Umsetzung dieser Vergleichsduplizierung verzichtet. Für das Plain Inter-Block Error Detection Verfahren werden im zugrundeliegenden Papier weiterhin explizite Verfahren der Datenhärtung beschrieben, auch hier wurde die Umsetzung unterlassen.

Die Kontrollflussüberwachung als solche ist in KESO als Konfigurationsoption mittels des `cfm` Parameters aktivierbar, die Auswahl der jeweiligen Härtungsvariante erfolgt ebenfalls mittels des zugehörigen Konfigurationsparameters `cfm_<verfahrensname>`.

3.3.2 Plain Inter-Block Error Detection

3.3.2.1 Verfahren

Bei dem von Rebaudengo, Cheynet et al. in [1] und [2] beschriebenen Verfahren handelt es sich um ein sehr grundlegendes, auf einfachen Transformationen auf dem in einer Hochsprache verfassten Quelltext beruhendes Verfahren. Die Zielsprache der ursprünglichen Umsetzung ist eine Untermenge der Programmiersprache C, die Übertragung der beschriebenen Umformungen auf andere imperative Hochsprachen ist jedoch weitestgehend unproblematisch. Die Autoren verzichteten auf eine explizite Benennung dieser Technik, die hier verwendete Bezeichnung *Plain Inter-Block Error Detection* entstammt [6].

Die beschriebenen Transformationen zielen sowohl auf die Erkennung von Daten- als auch von Kontrollflussfehlern ab.

Die Erkennung von Datenfehlern basiert hier im Wesentlichen auf Duplikation aller Variablen. Für jede verwendete Variable wird eine Kopie eingeführt, welche zeitgleich zum Original aktualisiert wird. Bei jedem Lesezugriff wird eine Äquivalenzprüfung durchgeführt, weichen die Werte ab, so wird ein Fehler gemeldet. Diese Verdopplung erstreckt sich auch auf Funktionsparameter, Rückgabewerte und Vergleichsoperationen in Bedingungen.

Wie bereits in Kapitel 2 angemerkt, verfügt KESO mittels Replikation über einen vergleichbaren, jedoch ungleich mächtigeren, Mechanismus, weshalb diese Arbeit auf die Umsetzung dieser Transformationen verzichtet und sich auf die zur Erkennung statischer Kontrollflussfehler entwickelten Umformungen beschränkt. Diese Beruhen wie auch ECCA, CFCSS und YACCA auf Blocksignaturen.

Dabei wird jedem Basisblock eine positive ganze Zahl k_i, die Signatur, zugewiesen. In einer globalen Variable, dem *Ausführungsindikator* (engl. *execution check flag*, ecf) wird der Wert des sich in Ausführung befindlichen Basisblocks mitgeführt. Dazu wird dem Ausführungsindikator im Prolog eines Basisblock BB_i die zugehörige Signatur k_i zugewiesen. Die Überprüfung dieses Wertes erfolgt dann im Epilog, indem auf Übereinstimmung von Ausführungsindikator und Basisblocksignatur k_i getestet wird. Da der Basisblock eine Folge verzweigungsfreier Instruktionen darstellt muss der hier gelesene Wert mit dem im Prolog gesetzten übereinstimmen.

Neben dieser Eingrenzung der Basisblöcke wird durch *Methodensignaturen* ein gesonderter Mechanismus zur Absicherung von Funktionsaufrufen und der Rückkehr aus solchen geschaffen. Neben den Basisblocksignaturen wird jeder Funktion ein eigener Signaturwert k_j, die *Methodensignatur* zugeordnet. Vor jeder `return`-Anweisung, welche sich naturgemäß als Verzweigung am Ende eines Basisblocks befindet, wird dem Ausführungsindikator im Anschluss an die oben beschriebene Prüfung dieser Signaturwert k_j zugewiesen und dieser vom Aufrufer direkt nach der Rückkehr überprüft.

Hierdurch soll insbesondere die Erkennung der folgenden Fehlerklassen sichergestellt werden:

- Sprünge in den Programmtext der Funktion
- Sprünge auf die unmittelbar einem Aufruf folgende Instruktion
- das Ziel eines Aufrufs beeinflussende Fehler
- die Rücksprungadresse einer Funktion beeinflussende Fehler

In Algorithmus 3.1 findet sich eine Beschreibung des Aufbaus eines so gehärteten Basisblocks.

Require: k_i ist Signatur des aktuellen Basisblocks BB_i
k_{method} ist die Signatur der aktuellen Methode
falls der Block mit einem Aufruf endet, ist k_{called} die Methodensignatur der aufgerufenen Methode

1: $ecf \leftarrow k_i$
2: ...{Regulärer Körper des Basisblocks ohne abschließenden Sprung}
3: **if** $ecf \neq k_i$ **then**
4: signalError()
5: **end if**
6: **if** Basisblock ist Endblock einer Methode *method* **then**
7: $ecf \leftarrow k_{method}$
8: **end if**
9: {Optional: Verzweigungsinstruktion des Basisblocks}
10: **if** Basisblock endete mit Aufruf an *called* **then**
11: **if** $ecf \neq k_{called}$ **then**
12: signalError()
13: **end if**
14: **end if**

Algorithmus 3.1 – Code des Basisblocks BB_i in Plain Inter-Block Error Detection

3.3.2.2 Einstufung

Da das Verfahren über keine Überprüfungen der Kontrollflussinformation im Sinn von Vorgänger- oder Nachfolgermengen verfügt, sind durch das hier geschilderte Verfahren keine Fehler, welche Übergänge vom Ende eines Basisblocks zum Beginn eines anderen auslösen, also Fehler des Typs eins gemäß der in Abschnitt 2.2.4 dargelegten Kategorisierung erkennbar [6]. In der durch die ursprünglichen Autoren beschriebenen Ausführung beinhaltet das Verfahren wie in Abschnitt 3.1 beschrieben eine Vergleichsverdopplung, auf deren Umsetzung aus Gründen der Vergleichbarkeit hier verzichtet wurde. Durch diese wären fehlerhafte Übergänge des Typus zwei erkennbar.

Durch die Einrahmung der Basisblöcke werden alle Abweichungen erkannt, welche von einem Basisblock ausgehend in die Mitte eines anderen übergehen. Dies umfasst alle Fehler der Kategorie drei und einen Großteil der Fehler der Kategorie vier.

Da bei Kontrollflussfehlern mit Ursprung und Ziel innerhalb desselben Basisblocks die Integrität der Signatur erhalten bleibt, sind Fehler des Typs fünf nicht durch das Verfahren detektierbar.

3.3.2.3 Anpassungen auf KESO

Neben dem erwähnten Verzicht auf Verdopplung der Vergleichsoperationen nach bedingten Sprüngen wurde das Verfahren an zwei weiteren Stellen an die Begebenheiten der KESO-Umgebung angepasst.

So ist, anders als bei der im ursprünglichen Verfahren betrachteten Zielsprache C, in Java die dynamische Auflösung (engl. *dynamic dispatch*) virtueller Aufrufe zulässig und üblich. Dies heißt, dass beim Ausführen einer polymorphen Methode in Abhängigkeit des Typs des Zielobjekts zwischen mehreren Möglichkeiten ausgewählt wird. In der hier umgesetzten Variante der Methodensignatur teilen sich alle Implementierungen einer polymorphen Methode den gleichen Signaturwert. Dies ermöglicht allerdings Aliasierung, eine Erkennung eines fehlerhaft aufgelösten Aufrufes ist aufgrund der Mehrdeutigkeit der einzelnen Signatur nicht mehr möglich. Eine solche falsche Auflösung geschieht immer dann, wenn sowohl die korrekte als auch die fehlerhaft angesprungene Methode Implementierungen der gleichen polymorph ererbten Methode darstellen.

Weiterhin gibt es in KESO immer wieder aus einer einzigen Instruktion bestehende Basisblöcke, wobei die einzige Instruktion einen Sprung darstellt. In diesen Fällen würde der Epilog unmittelbar auf den Prolog folgen, d.h. der durch die Signatur geschützte Abschnitt wäre leer. In diesem Falle wurde in der Implementierung auf das Einfügen der eingrenzenden Signaturüberprüfung verzichtet. Lediglich das Setzen der Signatur und das eventuelle Testen der Methodensignatur wird hier durchgeführt, da nur so eine Veränderung des Ausführungsindikators für jeden neu besuchten Basisblock garantiert werden kann.

3.3.3 Enhanced Control flow Checking using Assertions – ECCA

3.3.3.1 Verfahren

Das von Alkhalifa et al. in [3] vorgestellte Verfahren *Enhanced Control-flow Checking using Assertions (ECCA)* ist der Nachfolger des *Control-flow Checking using Assertions (CCA)* [17], welches sich aufgrund von Ressourcensparsamkeit und geringen Latenz besonders für Echtzeitsysteme eignen soll. Dabei werden in dem zugrundeliegenden Papier zwei Varianten auf unterschiedlicher Sprachebene angeboten, einerseits auf Hochsprachenebene (ECCA-HL), andererseits auf der Ebene einer Zwischensprache (ECCA-IL). Da KESO im eigentlichen Sinne einen *Quelltext-zu-Quelltext-Übersetzer* (engl. *source to source compiler*) darstellt, wäre je nach Einstufung eine Anwendung von ECCA auf beiden Sprachebenen denkbar, für ECCA-IL spräche die Anwendung auf die JINO-interne, an den Java-Bytecode angelehnte Zwischensprache, für ECCA-HL die Zielsprache C, für die die Veröffentlichung ECCA-HL beschreibt. Allerdings wird durch die Autoren in [3] lediglich die Umsetzung von ECCA-IL für die Register Transfer Language (RTL) der GNU Compiler Collection [18, S.309] beschrieben und durch Generalisierung daraus ein allgemeines Verfahren entwickelt. Allerdings baut diese Implementierung auf der Eigenschaft der RTL auf, dass jede Instruktion und somit auch jeder Basisblock nur maximal zwei Nachfolger besitzt. Da jedoch wie in Abschnitt 3.2 dargelegt die Zwischensprache aufgrund von virtuellen Methodenaufrufen diese Eigenschaft nicht besitzt, ist eine Umsetzung von ECCA-IL nicht möglich, im Folgenden wird dementsprechend nur ECCA-HL behandelt.

Dem ECCA-HL Verfahren liegt die Idee zugrunde, die *Laufzeitfehler-Mechanismen* (engl. *traps*) der Hardware im Falle eines entdeckten Fehlers automatisch auszulösen. Dieser Ansatz hat unter anderem den Vorteil, dass die vom Verfahren erzeugte Instruktionssequenz keine neuen Sprünge im Programm erzeugt. Stattdessen löst das Verfahren bei einem unzulässigen Kontrollflussübergang einen Laufzeitfehler, um genau zu sein eine *Teilung durch Null* (engl *Division by Zero*), aus, welche auf nahezu allen gängigen Prozessoren bei einer ungültigen Ganzzahldivision mit Divisor null automatisch ausgelöst wird und oft auch durch eine Routine in Software behandelt werden kann.

Dazu wird zunächst jedem Basisblock eine eindeutige Signatur, die sogenannte *Blockidentität* (BID, engl. *Block Identifier*), in Form einer Primzahl größer zwei zugewiesen und eine globale Variable *id* eingeführt, welche während der Ausführung eines Basisblocks mit dem zugehörigen Signaturwert übereinstimmt.

Zur eigentlichen Überprüfung des Kontrollflusses wird nun jeweils eine Operation in Prolog und Epilog eingefügt. Die im Prolog eines Basisblocks BB_x eingefügte SET-Operation sieht dabei wie folgt aus:

$$id = \frac{BID_x}{\overline{(id \mod BID_x)} \cdot (id \mod 2)} \tag{3.1}$$

Die Semantik von $\overline{\cdot}$ entspricht hier der des aus C bekannten !-Operators [19]:

$$\overline{x} = \begin{cases} 1 & \text{falls } x = 0 \\ 0 & \text{sonst} \end{cases} \tag{3.2}$$

Somit sind beide Komponenten des Nenners in (3.1) Ganzzahlen mit den Werten 0 oder 1. Sollte also beim Blockeintritt der Wert von *id* nicht durch die Signatur des aktuellen Blocks teilbar sein oder es sich um eine gerade Zahl handeln, so ergibt sich der Nenner zu 0, die oben erwähnte Teilung durch Null tritt ein und eine Ausnahmebehandlung wird ausgelöst. Ansonsten ergibt sich der Wert des Nenners zu 1, *id* wird somit auf den Wert der Blocksignatur des jeweiligen Blocks aktualisiert.

Die im Epilog eines Blockes durchgeführte Operation stellt nun sicher, dass bei einer fehlerfreien Durchführung keine der oben geschilderten Bedingungen eintreten kann. Diese TEST-Operation ist wie folgt gestaltet:

$$id = \underbrace{\left(\prod_{i \in Nachfolger(j)} BID_i \right)}_{NEXT} + \underbrace{\overline{\overline{(id - BID_x)}}}_{CHECK} \tag{3.3}$$

Da es sich bei den Blocksignaturen um Primzahlen ungleich zwei handelt, ist das Produkt dieser Werte, $NEXT$, eine ungerade Zahl. Während eines korrekten Durchlaufs gilt in einem Block, dass *id* den Wert der jeweiligen Signatur enthält, $CHECK$ ist somit 0, in allen anderen Fällen ergibt die Doppelnegation für $CHECK$ den Wert 1, die Summe und somit *id* wird gerade. In der SET-Operation des Nachfolgers würde in diesem Fall eine Division durch Null ausgelöst, es werden somit Sprünge in den Basisblock erkannt.

Zur Absicherung der Übergänge zwischen Blöcken wird die Kombination aus $NEXT$ und der zugehörigen Überprüfung $\overline{(id \mod BID_x)}$ verwendet. Da es sich bei den Blocksignaturen um Primzahlen handelt, gilt die Teilbarkeit nur, falls der Wert von *id* vorher von einem der Vorgänger des Blocks gesetzt wurde. In allen anderen Fällen ergibt die Modulooperation einen von 0 abweichenden Wert, aufgrund der Negation erfolgt eine Division durch Null und der Laufzeitfehler wird ausgelöst.

In Algorithmus 3.2 findet sich ein Skelett eines mit ECCA gehärteten Basisblocks.

Require: BID_i ist Signatur des Basisblocks BB_x
Require: $Nachfolger(x)$ ist die Menge der Nachfolger von Basisblock BB_X
1: $id \leftarrow \frac{BID_x}{\overline{(id \mod BID_x)} \cdot (id \mod 2)}$
2: ...{Regulärer Körper des Basisblocks ohne abschließenden Sprung}
3: $id \leftarrow \left(\prod_{i \in Nachfolger(j)} BID_i \right) + \overline{\overline{(id - BID_x)}}$
4: {Optional: abschließende Verzweigungsinstruktion des Basisblocks}

Algorithmus 3.2 – Code eines Basisblocks BB_x in ECCA

3.3.3.2 Einstufung

Durch den Epilog, welcher in Verbindung mit dem Prolog des Nachfolgers die Blocksignatur des aktuellen Blockes prüft, werden alle Sprünge von außen in den Rumpf des Blocks, also Fehler der Typen drei und vier erkannt [6]. Typ vier umfasst auch solche fehlerbedingten Sprünge die einen Basisblock vor dem Epilog verlassen und zum Beginn eines anderen springen. In diesem Fall entspricht die Laufzeitsignatur *id* der Signatur des Blocks, in dem der Fehler auftrat. Da es sich dabei um eine Primzahl handelt, schlägt die Teilbarkeitsprüfung im nächsten Prolog fehl. Somit werden alle Fehler des Typs vier erkannt.

Tritt ein Fehler am Ende eines Basisblocks auf und lenkt den Kontrollfluss mittels eines Fehlers des Typs eins auf den Beginn eines Basisblocks, welcher keinen regulärer Nachfolger des jeweiligen Knotens darstellt, so enthält der in *id* gespeicherte $NEXT$-Wert keinen Faktor, der der jeweiligen Signatur des Zielblocks entspricht, der Fehler wird abgefangen.

Da das Verfahren keine Plausibilitätsprüfung legaler Übergänge vorsieht, bleiben Fehler des Typs zwei unerkannt. Gleiches gilt für Fehler der Kategorie fünf, da Überprüfungen nur auf Blockgranularität stattfinden.

3.3.3.3 Anpassungen auf KESO

Das vorliegende Verfahren besitzt einige Eigenschaften, die eine Umsetzung in KESO oder allgemein auf eingebetteten Systemen unattraktiv machen. Einerseits stellt die Division eine sehr teuere Operation dar, besonders vor dem Hintergrund, dass einige von KESO unterstützte Prozessorarchitekturen, wie bspw. der Infineon Tricore, keine vollständige Divisionsoperation besitzen, sondern lediglich Unterstützung für ein effiziente Softwareimplementierung bieten [20].

Andererseits ergibt sich ein viel grundlegenderes Problem aus der geforderten Primzahleigenschaft und der Art der Verwendung der Primzahlen während des Verfahrens. Die bloße Menge der benötigten Primzahlen als solche ist unbedenklich. So bietet der Zahlenraum der nicht vorzeichenbehafteten, natürlichen Zahlen einer Länge von 16bit 6542 Primzahlen, die größte momentan von KESO übersetzte Anwendung, eine Ausprägung der CD_x Anwendungsfamilie [21], verfügt je nach Konfiguration über ca. 1500 Basisblöcke. Ein Problem ergibt sich jedoch aus der Produktbildung über den Signaturen der Nachfolger eines Knotens zur Bestimmung von $NEXT$.

Sei $P = p_0, p_1, p2, \ldots$ die aufsteigend sortierte Folge aller Primzahlen, so gilt:

$$\prod_{i=0}^{9} p_i = 6469693230 > 2^{32} \tag{3.4}$$

$$\prod_{i=0}^{15} p_i = 33640421653873594560 > 2^{64} \tag{3.5}$$

Dies bedeutet, dass im Fall einer 32bit Laufzeitsignatur ein Basisblock nicht mehr als neun Nachfolger besitzen darf, bei einer 64bit Signatur erhöht sich dieser Wert auf 15. Es

sei angemerkt, dass diese Zahlen aus dem Produkt der jeweils kleinsten verschiedenen Primzahlen entstehen, in der Realität sinken diese Zahlen in Abhängigkeit der Signaturwerte der Nachfolger weiter. Für einen rein imperativen Kontrollfluss ohne das Konzept von Funktionsaufrufen, bei dem jeder Basisblock über lediglich maximal zwei Nachfolger, ein Sprungziel einerseits und den direkten Nachfolger andererseits, verfügt, stellt diese Einschränkung kein Problem dar. Da jedoch das Verfahren keine speziellen Mechanismen für Funktionsaufrufe vorsieht, diese also folglich in den Kontrollflussgraphen eingebettet sind, ergeben sich aus den oben hergeleiteten Zahlen tatsächliche Einschränkungen für die Häufigkeit der Benutzung einer Funktion, da so jede Aufrufstelle einen validen Nachfolger des eine Methode abschließenden Endbasisblocks darstellt. Besonders vor dem Hintergrund virtueller Funktionsaufrufe ist diese Grenze in der Realität schnell erreicht. So konnte der oben erwähnte CD_j aufgrund dieser Einschränkung nicht durch das Verfahren gehärtet werden. Auch die von den Autoren angedachte Aufwandsreduktion durch Verringerung der Granularität auf Basisblockgruppen ist hier nur sehr begrenzt hilfreich, da sich Aufrufe üblicherweise über das gesamte Programm verteilen und so nur sehr große, unpraktikable Gruppen entstehen würden, da die geschilderten Gruppen nur aus im Graphen verbundenen Knotenmengen mit einem einzigen Eingang und einem einzigen Ausgang gebildet werden können.

Da sich jedoch die üblichen Bitbreiten als zu gering erweisen und sich die Verwendung einer BigInt-Bibliothek für diesen Anwendungszweck aus verschiedenen Gründen, sowohl dem hohen Laufzeitbedarf einerseits als auch der Notwendigkeit dynamischer Realokationen und einer Speicherverwaltung andererseits, auf eingebetteten Systemen verbietet, wurde die Implementierung an dieser Stelle nicht mehr weiter verfolgt.

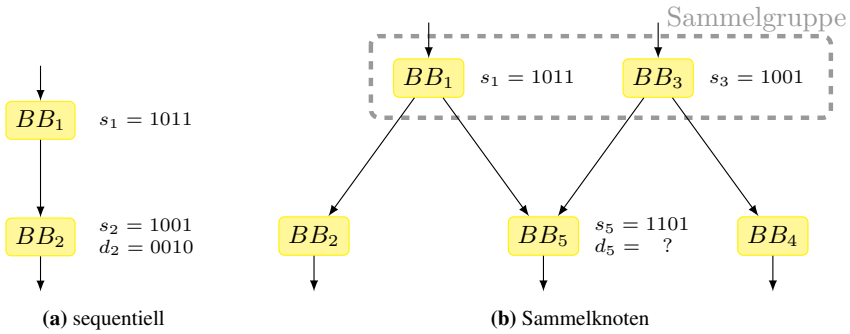
3.3.4 Control Flow Checking by Software Signatures – CFCSS

3.3.4.1 Verfahren

Auch der *Control Flow Checking by Software Signatures (CFCSS)* genannte, durch Oh et al. in [4] geschilderte, Ansatz stellt ein signaturbasiertes Verfahren dar. Wie auch bei ECCA und YACCA wird hier die Kontrollflussinformation über Vorgänger und Nachfolger verwendet, um die korrekte Reihenfolge der Ausführung der Basisblöcke zu überprüfen.

Dazu wird jedem Basisblock BB_i eine eindeutige Signatur s_i zugewiesen. Das Verfahren stellt sicher, dass während einer fehlerfreien Ausführung des Programmes die mitgeführte *Laufzeitsignatur* in der neu eingefügten Variable G zu jedem Zeitpunkt die Signatur des momentan ausgeführten Basisblockes beinhaltet. Um am Basisblockanfang von der alten Signatur auf die neue Signatur zu wechseln, wird hier die `xor`-Operation verwendet.

Die Autoren begründen die Wahl dieses Operators im Wesentlichen mit der einfachen Umsetzung der Operation in Hardware. Durch die Verwendung nur weniger grundlegender Operationen soll die Wahrscheinlichkeit eines Fehlers in den einzelnen Berechnungsschritten des Verfahrens gesenkt werden.



(a) sequentiell **(b)** Sammelknoten

Abbildung 3.2 – Knotenübergänge im sequentiellen Fall und bei Sammelknoten im Sinne von [4]

Für jeden Block BB_i wird die Signaturanpassungsfaktor d_i bestimmt, welche den Übergang von einem Vorgänger auf den aktuellen Basisblock sicherstellt. Die eigentliche Anpassung wird dann durch die folgende Formel beschrieben, welche dem Prolog jedes Basisblocks hinzugefügt wird:

$$G = G \oplus d_i \tag{3.6}$$

Im in Abbildung 3.2a geschilderten Fall, also wenn der Basisblock BB_i über genau einen Vorgänger BB_{prev} mit der Signatur s_{prev} erreicht wird, ist $d_i := s_i \oplus s_{prev}$, denn dann wird die Laufzeitsignatur vom alten Wert s_{prev} zum neuen Wert s_i überführt:

$$G' = G \oplus d_i \tag{3.7}$$
$$= s_{prev} \oplus \left(s_i \oplus s_{prev}\right) \tag{3.8}$$
$$= s_i \tag{3.9}$$

Damit ist nach Ablauf des Prologs in BB_i wieder die geforderte Übereinstimmung der Laufzeitsignatur G mit der Signatur des sich in Ausführung befindlichen Basisblocks hergestellt, welche nach der Signaturaktualisierung im Prolog des Basisblocks überprüft wird.

Diese einfache Berechnungsweise schlägt jedoch für den Fall eines *Sammelknotens* (engl. *branch fan in node*), also eines Basisblocks mit mehr als einem Vorgänger, wie in Abbildung 3.2b für Knoten BB_5 gezeigt, fehl, denn jeder der beiden Vorgänger BB_1 und BB_3 benötigt einen eigenen Anpassungsfaktor, da sich die jeweiligen Signaturen s_1 und s_3 unterscheiden.

Die Entwickler des Verfahrens diskutieren zur Auflösung dieser Problematik zwei Möglichkeiten [4]:

Einerseits könnte die Disjunktheitsanforderung für Blocksignaturen aufgehoben werden. In diesem Fall würden den Knoten BB_1 und BB_3 die gleiche Signatur zugewiesen, wodurch der Anpassungsfaktor d_5 wieder wie oben beschrieben mittels $\oplus$ aus der gemeinsamen Signatur der Vorgänger und der des aktuellen Basisblocks errechnet werden könnte. Gleichzeitig werden die Blöcke dadurch jedoch ununterscheidbar, alle Basisblöcke mit gemeinsamen Nachfolgern, eine *Sammelgruppe*, im obigen Beispiel umrandet dargestellt, treten im Kontrollflussgraphen dann wie ein einziger Knoten auf. Besonders diese letzte Eigenschaft schränkt die Leistungsfähigkeit des Algorithmus deutlich ein, da hierdurch illegale Übergänge ermöglicht werden. So wäre im Beispiel ein unerkannter Übergang von BB_1 nach BB_4 möglich, da BB_1 und BB_4 sich eine Signatur teilen und für die Kontrollflussüberwachung somit nicht unterscheidbar wären, obwohl dieser Übergang im Kontrollflussgraphen nicht vorhanden ist.

Ein Teil dieser Fehler lässt sich mittels der zweiten diskutierten Variante abfangen. Dazu wird aus jeder Sammelgruppe ein *Repräsentant* BB_r ausgewählt, dessen Signatur in allen im Kontrollfluss auf die Gruppe folgenden Sammelknoten, stellvertretend für alle Vorgänger innerhalb der Gruppe, zur Berechnung des Anpassungsfaktors herangezogen wird. Weiterhin wird eine Differenzsignatur D_x für jeden Knoten BB_x in der Gruppe hinzugefügt, welche sich aus der $\oplus$-Differenz der Signatur von s_x zur Signatur des Repräsentanten ergibt:

$$D_x = s_x \oplus s_r \tag{3.10}$$

Weiterhin wird neben der Laufzeitsignatur G eine weitere globale Variable, die *Laufzeitdifferenz* D eingeführt. Diese transportiert die Information über den tatsächlichen Vorgänger aus der Gruppe in die Nachfolger. Dazu wird ihr in jedem Basisblock, der Teil einer Sammelgruppe ist, die jeweilige Differenzsignatur des Blocks zugewiesen. Damit gilt am Ende eines jeden Blocks BB_x einer Sammelgruppe:

$$G \oplus D = s_x \oplus (s_x \oplus s_r) = s_r \tag{3.11}$$

In jedem Sammelpunkt i mit dem Anpassungsfaktor $d_i = s_i \oplus s_r$ wird dementsprechend die Aktualisierung von G um D erweitert:

$$G = G \oplus D \oplus d_i \tag{3.12}$$

In Abbildung 3.3 findet sich für das obige Beispiel eine mögliche Belegung mit Differenzsignaturen.

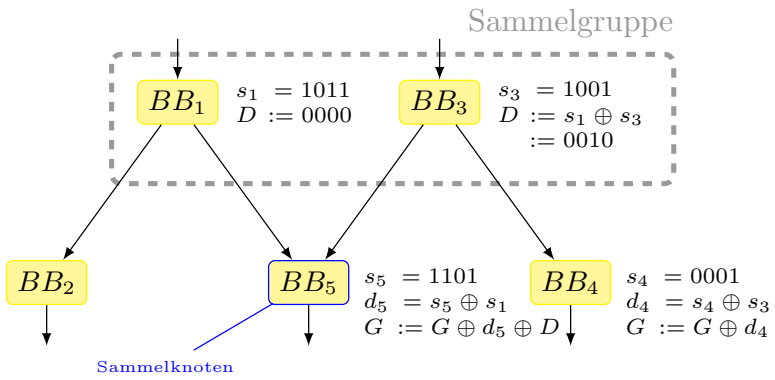


Abbildung 3.3 – Übergänge and Sammelpunkten mittels Differenzsignaturen

Damit ergibt sich für einen Basisblock BB_x der folgende Aufbau:

Require: Anpassungsfaktor $d_x = s_{pred} \oplus s_x$, mit s_{pred} Signatur des Vorgängers oder, falls BB_x Sammelknoten ist, die Signatur des Repräsentanten der Vorgängergruppe

1: $G \leftarrow G \oplus d_x$
2: **if** x ist Sammelknoten **then**
3: $\quad G \leftarrow G \oplus D$
4: **end if**
5: **if** G != s_x **then**
6: $\quad$ signalError()
7: **end if**
8: **if** x ist Teil einer Sammelgruppe **then**
9: $\quad$ Sei s_r Signatur des Repräsentanten der Gruppe
10: $\quad D \leftarrow s_r \oplus s_x$
11: **end if**
12: ...{Regulärer Code des Basisblocks}

Algorithmus 3.3 – Code eines Basisblocks BB_x in CFCSS

Wichtig ist hier, dass sowohl die Berechnung von d_x, als auch die Entscheidungen in den Zeilen zwei und acht, ob es sich bei BB_x um einen Sammelknoten handelt und ob BB_x Teil einer Sammelgruppe ist, jeweils statisch, zum Übersetzungszeitpunkt getroffen

und ausgewertet werden. Es entstehen dementsprechend an diesen Stellen keine neuen Verzweigungen. Auch s_r wird vom Übersetzer bestimmt und als Konstante eingefügt.

Da in Abbildung 3.3 für den Basisblock BB_4, welcher nur einen Vorgänger hat und somit keinen Sammelknoten darstellt, die Laufzeitdifferenz D nicht auf die Laufzeitsignatur G angewendet wird, können fälschlicherweise auftretende Verzweigungen von anderen Knoten aus der Sammelgruppe erkannt werden.

Diese beschriebene Modifikation des Algorithmus stellt also eine Optimierung für all diejenigen Knoten dar, welche einen einzigen Vorgänger besitzen, welcher Teil einer Sammelgruppe ist. Dennoch sind weiterhin illegale Übergänge zwischen Knoten der Gruppe und auf die Gruppe folgenden Sammelknoten möglich, wie es in dem in Abbildung 3.4 gezeigten Übergang der Fall ist. Da hier BB_6 kein Nachfolger von BB_1 ist, aber durch die Laufzeitdifferenz D die mit BB_2 und BB_3 in der Sammelgruppe geteilte, gültige Signatur des Repräsentanten erzeugt wird, ist der Fehler nicht erkennbar. Die Autoren sprechen in [4] hier von *Aliasierung* (engl. *Aliasing*).

3.3.4.2 Einstufung

Gemäß der in Abschnitt 2.2.4 aufgestellten Klassifikation kann das geschilderte Verfahren einen Großteil der Fehler der Typen eins, drei und vier erkennen [6], also diejenigen Fehler, welche einen Wechsel des Basisblocks auslösen und dabei keiner der im Kontrollflussgraphen vorhandenen Kanten folgen. Da das Verfahren lediglich eine Signatur je Basisblock vorsieht und sowohl der Übergang als auch die Überprüfung dieses Signaturwertes ausschließlich im Prolog stattfinden, sind selbstreferentielle Sprünge, welche aus dem Basisblock zurück in dessen Körper führen, unabhängig ob diese am Ende des Basisblocks (Typ vier) oder in der Mitte des Blocks (Typ fünf) entspringen, nicht detektierbar. Da keine semantische Überprüfung, bspw. der Sprungbedingungen oder Objekttypen bei dynamisch aufgelösten Methodenaufrufen stattfindet, werden weiterhin keine Fehler des Typs zwei erkannt. Abgesehen

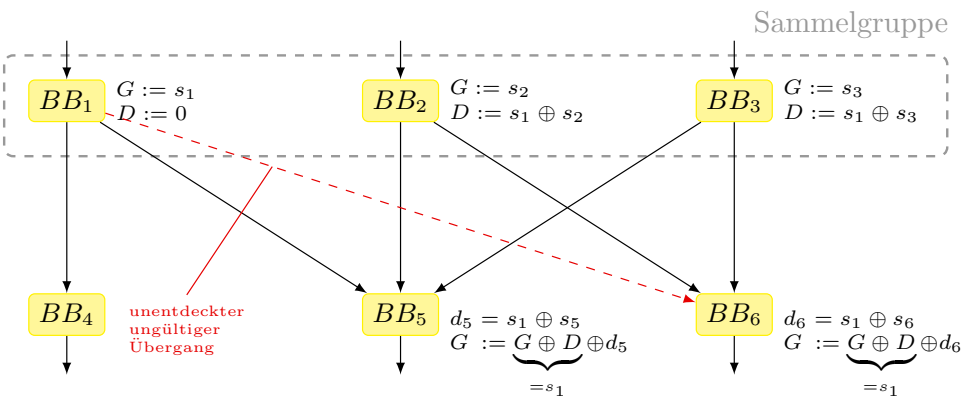


Abbildung 3.4 – Aliasierung bei CFCSS gemäß [4]

davon schränken die in Abbildung 3.4 gezeigten Aliasierungserscheinungen die Erkennung eigentlich illegaler Übergänge zwischen Blöcken einer Sammelgruppe, also Fehler des Typ eins, weiter ein.

3.3.4.3 Anpassungen auf KESO

Bei der Umsetzung ergaben sich keine speziellen Probleme, die Umsetzung erfolgte wie hier geschildert. Allerdings erfolgte die im Papier durchgeführte Umsetzung auf der Ebene der Maschinenprogramme bzw. der Assemblersprache, in dieser Arbeit erfolgte hingegen eine Implementierung auf Hochsprachenebene. Die strukturelle Äquivalenz sollte zu vergleichbaren Detektionskapazitäten führen. Allerdings verhindert diese Art der Umsetzung Mikrooptimierungen bspw. bei der Auswahl der verwendeten Assemblerinstruktionen und kann nicht für eine minimale Anzahl an neu eingefügten Verzweigungen im für das Verfahren erzeugte Maschinenprogramm garantieren.

3.3.5 Yet Another Controlflow Checking using Assertions – YACCA

3.3.5.1 Verfahren

Das *Yet Another Control flow Checking Approach (YACCA)*-Verfahren [5] wird von den Autoren als speicher- und laufzeiteffizientes Verfahren zur Absicherung von Programmen auf der Ebene der Hochsprache beschrieben, welches sich als Verbindung und Verbesserung von ECCA und CFCSS versteht. Die wesentliche Neuerung ist einerseits die zweimalige Signaturprüfung und -anpassung sowohl im Prolog als auch im Epilog, was die Erkennungsrate verbessern soll, als auch andererseits das Verfahren den Signaturanpassung, welche bei CFCSS für die in Abschnitt 3.3.4 geschilderte Aliasierungverantwortlich war und hier mittels einer veränderten Implementierung der Gruppensignaturen gelöst wird. Auch bemüht sich das Verfahren, die Anzahl neu eingeführter Verzweigungen gering zu halten.

Neben der Publikation „Soft-error detection using control flow assertions“ [5] durch Goloubeva et al. aus dem Jahr 2003 existiert eine weitere Publikation „Improved software-based processor control-flow errors detection technique“ [22] der gleichen Autoren von 2005, welche auf YACCA aufbauend eine Optimierung YACCA+ entwickelt. Dort findet sich darüber hinaus eine Definition und Beschreibung von YACCA. Soweit nicht anders angegeben wird hier die in dem späteren Papier [22] gegebene Implementierung von YACCA verwendet, insbesondere da sich in dieser Publikation im Gegensatz zu der Veröffentlichung von 2003 [5] eine tatsächliche, präzise Definition der im Verfahren zu verwendenden Operationen findet, während vorher die Transformation des Quelltextes lediglich exemplarisch beschrieben wurde [5].

Wie alle signaturbasierten Verfahren der Kontrollflussüberwachung verwendet YACCA eine globale Variable `code`, welche die dem momentanen Ausführungszustand zugeordnete Signatur enthält. Im Gegensatz zu den bisher vorgestellten Verfahren werden jedem Basisblock jedoch zwei Signaturen zugewiesen, eine Eingangs- und eine Ausgangssignatur. Der Prolog vollzieht dabei den Übergang von der Ausgangssignatur eines der Vorgänger hin zur Eingangssignatur des aktuellen Basisblocks, welche über den eigentlichen Körper des Basisblocks hinweg in `code` gespeichert ist. Erst beim Verlassen des Blockes, im Epilog, wird dann der Wechsel hin zur Ausgangssignatur des aktuellen Basisblocks vollführt. Hierdurch wird die Erkennung fehlerhafter Übergänge aus dem Epilog des aktuellen Basisblocks zurück in den Körper des gleichen Basisblocks erkennbar, da sich Ausgangs- und Eingangssignatur unterscheiden, eine genauere Analyse findet sich in Abschnitt 3.3.5.2. Im Folgenden bezeichnet die Abkürzung $I1_j$ die Eingangs- und $I2_j$ Ausgangssignatur des Basisblockes BB_j.

Jeder Signaturübergang wird dabei in zwei Schritte untergliedert, die *Überprüfung* (*test assertion*) und die eigentliche *Aktualisierung* (*set assertion*).

Zur Überprüfung werden zwei Möglichkeiten diskutiert. Die erste Variante basiert auf einem zu ECCA analogen Mechanismus, bei dem für jeden Block BB_j das Produkt der Ausgangssignaturen seiner Vorgänger gemäß der folgenden Regel gebildet wird:

$$PREVIOUS_j = \prod_{x \in vorgaenger(j)} I2_x \tag{3.13}$$

Damit lässt sich die Zulässigkeit eines Übergangs wie bei ECCA per Division prüfen:

---

if $PREVIOUS_j \mod code \neq 0$ **then**
  signaError()
end if

Allerdings wird in den dem Verfahren zugrundeliegenden Papieren anders als bei ECCA für die verwendeten Signaturen lediglich die Eindeutigkeit, jedoch keine Primeigenschaft gefordert. Damit würden allerdings ungültige Übergänge möglich, bei einer Nutzung von Primzahlen entstünden die gleichen Probleme wie bei ECCA bezüglich der Größe von *PREVIOUS*. Darüber hinaus wird diese Variante aufgrund der Divisionsoperation durch die Autoren selbst als „besonders kritisch vom Berechnungsstandpunkt aus“ (engl. „*particularly critical from the computational point of view*“) eingestuft [5]. Eine Erläuterung unterbleibt, aus den für YACCA+ getroffenen Optimierungen wird jedoch deutlich, dass es sich bei der Division im Sinne der Autoren um ein *j-Instruktion* handelt, also einen Befehl, der auf bestimmten Architekturen vom Übersetzer in eine Befehlsfolge übersetzt wird, welche selbst Sprünge beinhaltet.

Die zweite Variante hingegen vergleicht naiv den Wert von `code` mit den Ausgangssignaturen der Vorgänger:

Require: $I2_0 \ldots I2_k$ Ausgangssignaturen der Vorgänger von j
1: **if** $(I2_0! = \texttt{code})$ AND $(I2_1! = \texttt{code})$ AND ...AND $(I2_k! = \texttt{code})$ **then**
2: signalError()
3: **end if**

Weiterhin schlagen die Autoren vor, die Anzahl der Verzweigungen durch Einführung einer weiteren Variable, `ERR_CODE`, welche das Auftreten eines Fehlers aufzeichnet, weiter zu reduzieren:

Require: $I2_0 \ldots I2_k$ Ausgangssignaturen der Vorgänger von j
1: ERR_CODE |= $(I2_0! = \texttt{code})$ AND $(I2_1! = \texttt{code})$ AND ...AND $(I2_k! = \texttt{code})$

Diese wird in regelmäßigen Abständen auf von null abweichende Werte überprüft, welches das Auftreten eines Fehlers in dem der Überprüfung vorangegangenen Programmabschnitt anzeigen würde.

Während die Autoren in dem Papier von 2003 [5] noch beide Varianten der Überprüfung anführen, findet sich in der späteren Veröffentlichung nur noch die zweite der hier geschilderten Varianten, welche auch in dieser Arbeit implementiert wurde.

Die *Zuweisung* aktualisiert nun nach der Überprüfung den Wert der `code` Variable auf den der gewünschten Eingangs- oder Ausgangssignatur. Die naive Implementierung $code \leftarrow IX_i$ erlaubt jedoch keine Entdeckung von Kontrollflussfehlern, welche diese Instruktion anspringen, da im Anschluss die korrekte Signatur gesetzt ist. Um dieses Problem zu lösen, transformiert YACCA die momentane Signatur mittels logischer Bitoperationen, um den neuen Wert von `code` auf die aktuelle Signatur zu überführen. Dazu wird folgende Struktur verwendet:

1: `code` $\leftarrow (code \,\&\, M1) \oplus M2$

Dabei handelt es sich bei $M1$ und $M2$ um Bitmasken. $M1$ schränkt hier die Menge der für den Übergang relevanten Bits von `code` ein, $M2$ erfüllt eine zum Signaturanpassungsfaktor in CFCSS analoge Rolle, und ist für die eigentliche Anpassung auf die neue Signatur verantwortlich. Diese sind dabei in der Veröffentlichung ohne weiterführende Erläuterungen zur Herkunft wie folgt definiert [6]:

$$M1_i = \overline{\left(\bigwedge_{j \in Vorgaenger(IY_i)} IX_j \right) \oplus \left(\bigvee_{j \in Vorgaenger(IY_i)} IX_j \right)} \tag{3.14}$$

$$M2_i = \left(IX_j \,\&\, M1_i \right) \oplus IY_i \tag{3.15}$$

X und Y sind dabei jeweils abhängig davon, ob die Masken im Prolog oder Epilog zum Einsatz kommen, passend zu definieren, IY bezeichnet die Signatur, auf welche gewechselt werden soll. Soll also im Epilog von der Eingangssignatur $I1$ auf die Ausgangssignatur $I2$ gewechselt werden, ist $X = 1$ und $Y = 2$, im Prolog ist diese Zuordnung umgekehrt.

Hier ist für die Berechnung eine Erweiterung der Vorgängerrelation für Epiloge notwendig, dabei ist der Vorgänger einer Ausgangssignatur die Eingangssignatur des jeweiligen Blockes, für den Eingang ist es die Ausgangssignatur der Vorgängerblöcke im Kontrollflussgraphen:

$$Vorgaenger(IX_i) = \begin{cases} \{I2_j | BB_j \text{ ist Vorgänger von } BB_i\} & \text{falls } X = 1, IX_i \text{ ist Eingangssignatur} \\ \{I1_i\} & \text{falls } X = 2, IX_i \text{ ist Ausgangssignatur} \end{cases} \tag{3.16}$$

Hierdurch wird logisch gesehen, wie in Abbildung 3.5 abgebildet, im Kontrollflussgraphen jeder Basisblock in zwei Unterknoten aufgespalten. Die Eingangssignatur wird dabei im oberen Knoten dem Prolog und dem Basisblockrumpf zugeordnet, der mit der Ausgangssignatur verknüpfte Knoten umfasst den Basisblock vom Epilog abwärts.

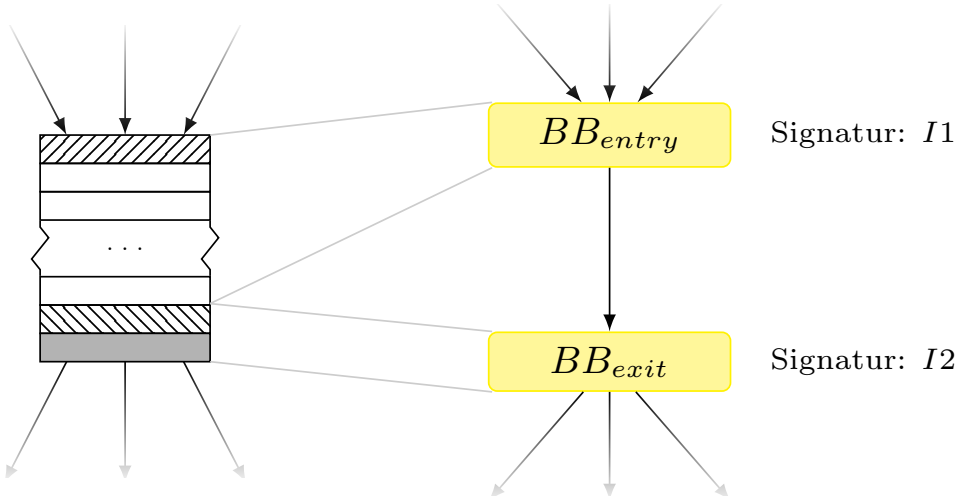


Abbildung 3.5 – Eingangs und Ausgangssignaturen bei YACCA

Mit diesen Definitionen kann nun die Funktionsweise und Herkunft der Masken erklärt werden. Zur Verdeutlichung werde zunächst der Epilog eines Basisblocks BB_i betrachtet. Hier gibt es durch die Auftrennung genau einen Vorgänger, während einer regulären Ausführung enthält `code` bei der Ankunft am Epilog also die Eingangssignatur $I1_i$ des Basisblocks. Es gilt $\&\{I1_i\} = I1_i = \vee\{I1_i\}$ und somit:

$$M1_i = \overline{\left(\mathop{\&}_{j \in Vorgaenger(I2_i)} I1_i\right) \oplus \left(\mathop{\bigvee}_{j \in Vorgaenger(I2_i)} I1_i\right)} \tag{3.17}$$

$$= \overline{I1_i \oplus I1_i} \tag{3.18}$$

$$= \overline{0} \tag{3.19}$$

Die $\bar{\cdot}$-Operation bezeichnet an dieser Stelle eine bitweise Negation, dementsprechend erhält man eine vollständig gesetzte Bitmaske, welche im Zweierkomplement einer -1 entspricht. Damit ist der Signaturübergang zu dem in CFCSS für Blöcke mit nur einem Vorgänger äquivalent:

$$\texttt{code} = (\texttt{code}\&M1_i) \oplus M2_i \tag{3.20}$$

$$= (\texttt{code}\&M1_i) \oplus \big((I1_i\&M1_i) \oplus I2_i\big) \tag{3.21}$$

$$= \texttt{code} \oplus (I1_i \oplus I2_i) \tag{3.22}$$

Eine analoge Formel trifft auch auf den Übergang von der Ausgangssignatur des Vorgängers zur Eingangssignatur eines gegebenen Basisblockes zu, solange dieser nur über einen einzigen Vorgänger verfügt.

Dementsprechend ist die Signaturzuweisung zu der in CFCSS äquivalent, der Unterschied zwischen den Verfahren liegt wie oben bereits angedeutet in der Behandlung von Sammel-

punkten. Diese wird aus einer Analyse der Komponenten von $M1$ offensichtlich (n sei hier die Bitlänge der Signaturen, b_k das k-te Bit der Bitmaske):

$$\mathop{\&}_{j \in Vorgaenger(IY_i)} IX_j = \left\{ b_{n-1} b_{n-2} \ldots b_0 | b_n = 1 \Leftrightarrow \forall IX_j . b_n \left(IX_j \right) = 1 \right\} \tag{3.23}$$

$$\mathop{\bigvee}_{j \in Vorgaenger(IY_i)} IX_j = \left\{ b_{n-1} b_{n-2} \ldots b_0 | b_n = 0 \Leftrightarrow \forall IX_j . b_n \left(IX_j \right) = 0 \right\} \tag{3.24}$$

Die Formeln besagen dabei, dass die aus der Verundung entstehende Bitmaske alle die Bits anzeigt, an denen in allen Signaturen der Vorgänger das jeweilige Bit übereinstimmend gesetzt ist, die Veroderung zeigt mittels einer 0 all die Stellen an, an denen in allen Vorgängern das jeweilige Bit nicht gesetzt ist. Die $\oplus$-Verknüpfung der beiden Mengen aus Gleichung (3.23) und Gleichung (3.24) selektiert somit alle diejenigen Stellen, bei denen es in den Vorgängern zu Abweichungen untereinander kommt. In diesen Fällen ist in der &-Verknüpfung das Bit nicht gesetzt, in der $\vee$-Verknüpfung hingegen schon, da mindestens eine der Signaturen das Bit gesetzt hat. Die bitweise Negation erzeugt dann diejenige Maske aller Bits, welche in allen zulässigen Vorgängersignaturen einen einheitlichen Wert besitzen. Abbildung 3.6 zeigt diese Berechnungsschritte und die Ausgangssituation für ein Beispiel, auch dort sind in $M1$ am Ende alle Bits gesetzt, welche in allen Vorgängern einen identischen Wert besitzen.

Damit handelt es sich bei der Verknüpfungsoperation im Prolog eines Basisblocks BB_i um eine auf diese einheitlichen Bits eingeschränkte Variante des regulären Übergangs für einen einzelnen Vorgänger mittels $\oplus$:

$$\texttt{code} = (\texttt{code} \& M1_i) \oplus M2_i \tag{3.25}$$

$$= (\texttt{code} \& M1_i) \oplus \left(\left(I1_{pred} \& M1_i \right) \oplus I2_i \right) \tag{3.26}$$

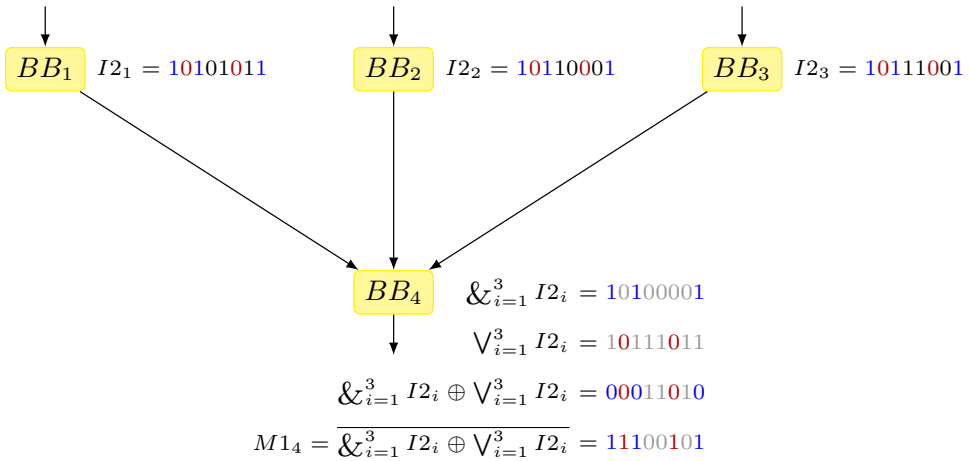


Abbildung 3.6 – Die Bitsets für $M1$ bei einem Prolog. In $M1$ sind abschließend alle Bits gesetzt, die in allen Vorgängern in ihrem Wert übereinstimmen

Für einen gültigen Signaturübergang muss `code` also in allen der über die Vorgänger hinweg identischen Bits mit diesen übereinstimmen.

Das aus Abschnitt 3.3.4 bekannte, dort aliasierende Beispiel lässt sich damit wie folgt in YACCA abbilden:

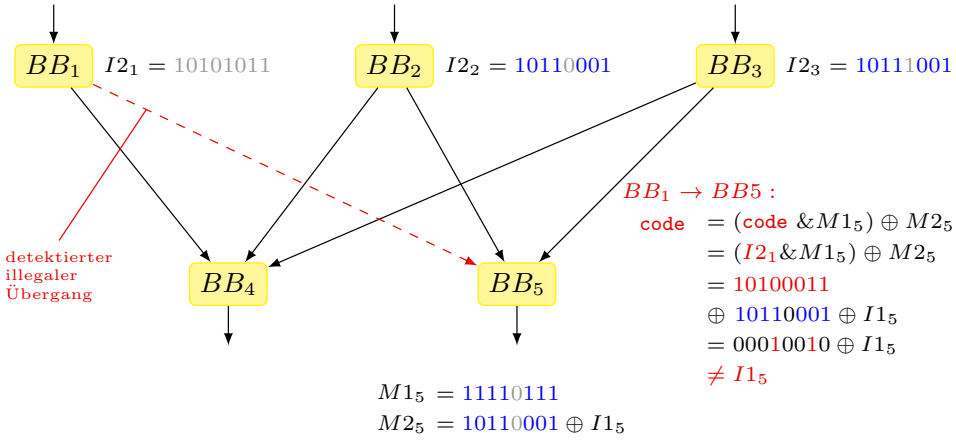


Abbildung 3.7 – Eine Beispielbelegung für einen in CFCSS aliasierenden Kontrollflussgraphen. Die Berechnung für den illegalen Übergang BB_1 nach BB_5 resultiert in einem ungültigen Wert der `code`-Variable

Falls sich der Ursprungsknoten des illegalen Übergangs wie im Beispiel im von $M1$ bezeichneten Bitset in mindestens einem Bit unterscheidet lässt sich somit Aliasierung verhindern.

Der Code eines durch YACCA modifizierten Basisblocks BB_x besitzt somit die folgende Struktur:

Require: $I2_0, \ldots, I2_k$ Ausgangssignaturen der Vorgänger

1: ERR_CODE |= $(I2_0! = \texttt{code})$ AND $(I2_1! = \texttt{code})$ AND ...AND $(I2_k! = \texttt{code})$

2: $M1_{x,in} \leftarrow \overline{\left(\&_{j=0}^{k} I2_j\right) \oplus \left(\bigvee_{j=0}^{k} I2_j\right)}$

3: $M2_{x,in} \leftarrow \left(I2_0 \,\&\, M1_{x,in}\right) \oplus I1_x$

4: $\texttt{code} \leftarrow \left(\texttt{code} \,\&\, M1_{x,in}\right) \oplus M2_{x,in}$

5: ...{Regulärer Code des Basisblocks}

6: ERR_CODE |= $(I1_x! = \texttt{code})$

7: $\texttt{code} \leftarrow \texttt{code} \oplus (I1_x \oplus I2_x)$

Algorithmus 3.4 – Code eines Basisblocks x in YACCA

Überprüfungen des Zustandes von ERR_CODE werden nach Bedarf eingefügt. Die Bitsets hängen nur von den zur Übersetzungszeit bekannten, konstanten Eingangs- und Ausgangssignaturen ab und können dementsprechend bereits vorausberechnet werden.

Allerdings ist wichtig zu bemerken, dass der in dieser Arbeit für den Epilog verwendete Algorithmus von der durch Goloubeva et al. in [6] angegebenen Belegungen der Sets abweicht. Dort werden für den Übergang am Epilog folgende Werte für die Bitmasken postuliert:

$$M1_i = 1 \tag{3.27}$$

$$M2_i = I1_i \oplus I2_i \tag{3.28}$$

Diese Werte sind jedoch laut der Semantik des Verfahrens nicht korrekt. Sei o.B.d.A. $I1_i := 4 = (100)_2$ und $I2_i := 5 = (101)_2$. Dann gilt gemäß der im Papier beschriebenen, fehlerhaften Regeln:

$$M1 = 1 \tag{3.29}$$

$$M2 = I1_i \oplus I2_i = (100)_2 \oplus (101)_2 = (001)_2 \tag{3.30}$$

Im Falle einer korrekten Ausführung müsste nun die definierte Zuweisungsfunktion den Wert von `code` von $I1_i$ zu $I2_i$ verändern. Dies ist jedoch nicht der Fall:

$$\texttt{code'} = (\texttt{code}\&M1) \oplus M2 \tag{3.31}$$

$$= \big((100)_2 \& 1\big) \oplus (001)_2 \tag{3.32}$$

$$= (000)_2 \oplus (001)_2 \tag{3.33}$$

$$= (001)_2 \tag{3.34}$$

$$\neq I2_i \tag{3.35}$$

Somit ist die im Papier angebotene Implementierung fehlerhaft. In KESO wurde dementsprechend die oben gezeigte, gültige Belegung der Bitsets umgesetzt.

3.3.5.2 Einstufung

Die hier geschilderte YACCA-Variante ermöglicht eine vollständige Erkennung der Fehler der Typen eins, drei und vier, durch die im Rahmen dieser Arbeit vernachlässigte, eigentlich vorgesehene Vergleichsverdopplung nach bedingten Sprüngen ist weiterhin die Detektion von Abweichungen der zweiten Kategorie möglich [6]. Die Verbesserungen des Verfahrens im Bezug auf CFCSS in den Typen eins und vier werden hier primär durch die Verwendung der zusätzlichen Ausgangssignatur erzielt, wodurch eine Differentiation zwischen dem Basisblockrumpf und dem Basisblockende ermöglicht wird, ansonsten weist das Verfahren große Parallelen auf. Durch die explizite Überprüfung der Vorgänger und das vermeiden gemeinsamer Vorgängersignaturen wird jedoch Aliasierung mehrheitlich durch eine günstige Wahl der Ausgangssignaturen und Bitsetmengen verhindert. Es ist weiterhin hervorzuheben, dass fehlerbedingte Sprünge in die Folge der Vergleichsoperationen unproblematisch sind, da hier lediglich die Akzeptanzmenge der Vorgänger noch weiter eingeschränkt wird, und somit ein erfolgreicher Ausgang noch unwahrscheinlicher wird. Die vom Verfahren eingeführte

Befehlsfolge ist also auch mehrheitlich gegenüber Sprünge in den zu Überwachung gehörigen Programmteil robust. Durch das Vermeiden einer expliziten Wertzuweisung durch die Berechnung aus der Vorgängersignatur mittels Bitoperationen wird das Risiko einer falschen Akzeptanz weiter reduziert.

3.3.5.3 Anpassungen auf KESO

Das Verfahren sieht in der ursprünglichen Form eine wie bereits eingangs in Abschnitt 3.3.1 erwähnte Vergleichsverdopplung in den Nachfolgern bedingter Sprünge vor, welche jedoch auch hier aus Gründen der besseren Vergleichbarkeit der Verfahren untereinander und dem Fokus auf die statischen Kontrollflussfehler nicht umgesetzt wurde.

Weiterhin werden in der Literatur unterschiedliche Varianten der Überprüfungsoperation geschildert. Da die erste der beschriebenen Optionen analog zu ECCA auf Produkten von Primzahlen basiert und dementsprechend von den in Abschnitt 3.3.3.3 geschilderten Problemen betroffen ist, wurde hier die zweite Variante der expliziten Folge von Vergleichsoperationen umgesetzt.

Wie oben beschrieben wurde darüber hinaus bei der Zuweisungsanweisung in den Epilogen die fehlerhaft definierte Bitmaske angepasst.

Ansonsten wurde das Verfahren in Übereinstimmung mit den zugehörigen Veröffentlichungen umgesetzt.

3.3.6 Dominatorbasierter Ansatz

Anders als die übrigen betrachteten Verfahren basiert das durch Christian Dietrich in [7, S.46ff] beschriebe Verfahren nicht auf Basisblocksignaturen, sondern überprüft den Kontrollfluss auf der Ebene von Basisblockgruppen. Durch die gröbere Granularität ist eine höhere Effzienz zu Lasten der Detektionfähigkeiten möglich. Die gewählte Ebene der Überwachung entspricht der der *Dominanzregionen*.

3.3.6.1 Grundlagen: Dominanz und Dominanzgrenzen

Die Dominanzrelation entstammt ursprünglich dem Übersetzerbau und dient dort neben der Platzierung von ϕ-Funktionen [23] unter anderem der Definition von *Regionen* [16, S.672]. Dabei handelt es sich um Bereiche des Kontrollflussgraphen, welche nur durch einen einzigen Eingangsknoten betreten werden können, wodurch auf diesem Abschnitt Optimierungen wie beispielsweise die Elimination gemeinsamer Ausdrücke in der *Region* möglich werden. Der Eingangsknoten *dominiert* alle Knoten seiner Region. Formal ist die Dominanzrelation für Basisblöcke wie folgt definiert [16, S.656]:

Definition 3. Ein Basisblock x dominiert einen Basisblock y gdw. jeder in der Wurzel des Kontrollflussgraphen beginnende Pfad zu Block y auch x erreicht.

Aus dieser Definition folgen trivialerweise Reflexivität und Transitivität der Relation: Zum einen kann jeder Block nur erreicht werden, wenn er erreicht wird, zum anderen gilt, falls alle Pfade zu Basisblock x durch y und alle Pfade zu y durch z führen, so führen auch alle Pfade zu x durch z. Auch ist die Relation antisymmetrisch, d.h. zwei unterschiedliche Basisblöcke x und y können sich niemals gegenseitig dominieren. Dementsprechend definiert die Dominanzrelation eine Halbordnung [24, S.338] auf der Menge der Basisblöcke. $dom(x)$ bezeichnet dabei die Menge aller Knoten im Kontrollflussgraphen, welche x dominieren.

Auf dieser Halbordnung lässt sich damit auch die *unmittelbare Dominanz* definieren [25]:

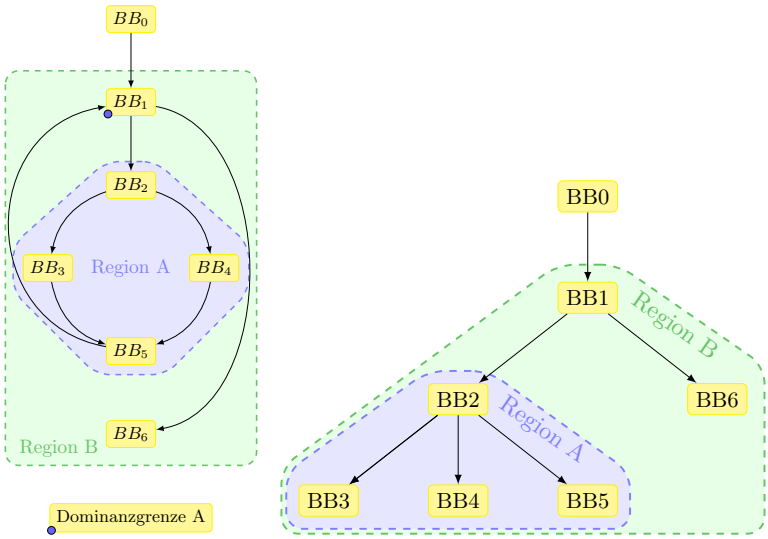
Definition 4. Ein Basisblock x *dominiert* den Basisblock y *direkt* bzw. *unmittelbar*, falls gilt:
$$x \in dom(y) \wedge \forall v \in dom(y).v \neq x \Rightarrow v \in dom(x)$$

d.h. es gibt gemäß der *dom*-Relation keinen weiteren Basisblock zwischen x und y, wobei x von y dominiert wird. y ist also der *unmittelbare Dominator*.

Dabei bezeichnet $idom(x)$ den unmittelbaren Dominator von x, aus dieser Relation ergibt sich ausgehend von der Wurzel des Kontrollflussgraphen ein *Dominatorbaum* [25]. Die Knoten sind dabei die Basisblöcke, die Kantenmenge K ist wie folgt durch *idom* bestimmt:

$$K = \left\{ \left(idom\left(w\right), w \right) | w \text{ ist Basisblock und nicht Wurzel} \right\} \tag{3.36}$$

Ein beispielhafter Dominatorbaum für den in Kapitel 2 gezeigten Kontrollflussgraphen findet sich in Abbildung 3.8.



(a) Kontrollflussgraph **(b)** Dominatorbaum

Abbildung 3.8 – Kontrollflussgraph und zugehöriger Dominatorbaum. Unterbäume des Dominatorbaumes bilden dabei Regionen

Jeder Unterbaum in einem Dominatorbaum bildet dabei wie in Abbildung 3.8 gezeigt eine Region. Die eine Region im Kontrollflussgraphen umgebenden Basisblöcken formen hier die *Dominanzgrenze* (engl. *dominance frontier*). Formal lässt sich diese Basisblockmenge für die von x dominierte Region wie folgt beschreiben [23]:

$$DF(x) = \{y | \exists p \in Vorgaenger(y) . x \in dom(p) \wedge p \neq x \wedge x \notin dom(y)\} \tag{3.37}$$

In Abbildung 3.8 ist die Dominanzgrenze der Region A beispielhaft hervorgehoben.

3.3.6.2 Verfahren

Das dominatorbasierte Verfahren zur Kontrollflussüberwachung [7] weist nun einigen dieser Regionen einen Marker zu, welcher in dem die Region dominierenden Eingangsbasisblock gesetzt wird. Erreicht der Kontrollfluss einen der zu dieser Region gehörigen Basisblöcke, ohne dass der zugehörige Marker gesetzt ist, so liegt ein Kontrollflussfehler vor, da einer der zum Block gehörigen Dominatoren nicht durchlaufen wurde. Die Menge der gesetzten Bits wird in der globalen Variable `markers` mitgeführt und durch Bitoperationen angepasst.

Um eine zuverlässige Fehlerdetektion zu gewährleisten sollte jedoch der Marker für alle Basisblöcke, welche nicht zur Region gehören zurückgesetzt werden. Die ursprüngliche Implementierung in [7] sieht dementsprechend für jeden Basisblock drei Bitsets vor, das Bitset der zu setzenden Marker, das der zu überprüfenden Marker und das der inaktiven Marker. Im Set der zu setzenden Marker S_{set} findet sich höchstens ein gesetztes Bit, falls der Block Eingangsblock einer der für die Kontrollflussüberwachung ausgewählten Regionen darstellt, ansonsten keines. Das Set der zu überprüfenden Bits S_{check} des Blocks BB_x ergibt

sich aus $\bigvee \{S_{set}(d) | d \in dom(x)\}$, das dritte Set ergibt sich als bitweises Komplement von S_{check}.

Damit erhält man den für einen Basisblock die folgende Form:

```
1: markers ← markers | S_set
2: if ((markers & S_check) != S_check) then
3:     signalError()
4: end if
5: markers ← markers & ¬S_set
6: ...{Regulärer Code des Basisblocks}
```

Algorithmus 3.5 – Code eines Basisblocks BB_x im dominatorbasierten Verfahren - Naive Variante

Abbildung 3.9 zeigt die Anwendung dieses Verfahrens auf die Region A des Kontollflussgraphen.

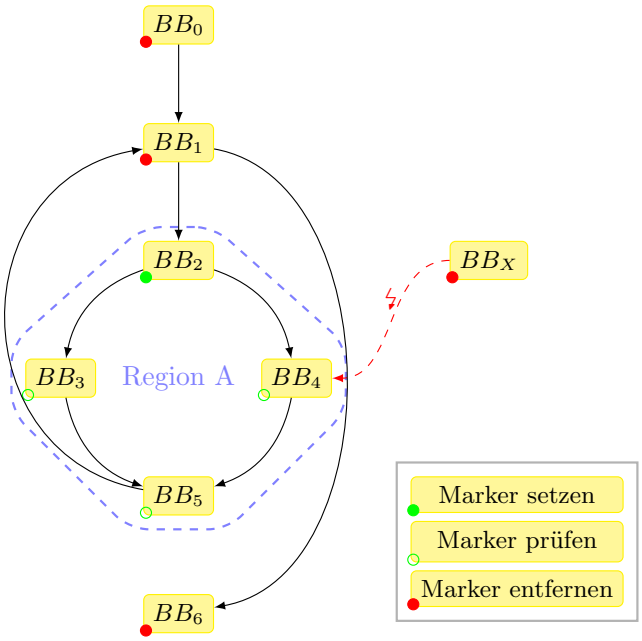


Abbildung 3.9 – Schematische Abbildung des instrumentierten Kontrollflussgraphen in Anlehnung an [7]

Dieser naive Algorithmus lässt sich jedoch durch die Verwendung von Dominanzgrenzen verbessern [7, S.81]. Statt in jedem Basisblock sämtliche inaktiven Marker zurückzusetzen ist das einmalige Zurücksetzten eines jeden Markers beim Übertreten einer Dominanzgrenze ausreichend. Dies steigert einerseits durch die wegfallende Rücksetzinstruktion in allen Basisblöcken, welche nicht zu einer Dominanzgrenze gehören, die Laufzeiteffizienz, andererseits wird die Prüfung auf die fälschliche Anwesenheit eines Markers möglich. Ist in

diesem Falle ein Marker einer Region gesetzt, zu der der Basisblock nicht gehört, so muss ein Kontrollflussfehler aufgetreten sein, welcher einen Sprung über die Dominanzgrenze hinweg hervorgerufen hat. Die Struktur eines Basisblocks ist dabei wie folgt gegeben:

1: **if** $(S_{frontier} != \emptyset)$ **then**
2: `markers` $\leftarrow$ `markers` & $\overline{S_{frontier}}$
3: **end if**
4: `markers` $\leftarrow$ `markers` $|\ S_{set}$
5: **if** $((\texttt{markers}\ \&\ S_{check})\ != S_{check})$ **then**
6: signalError()
7: **end if**
8: ...{Regulärer Code des Basisblocks}

Algorithmus 3.6 – Code eines Basisblocks x im Dominatorbasierten Verfahren - Optimierte Variante

Für die beispielhafte Region A aus dem hier verwendeten Beispiel ergibt sich damit der in Abbildung 3.10 gezeigte Ablauf. Zusätzlich zu den in der unoptimierten Variante erkannten Fehlern ist hierbei auch der illegale Übergang aus der Region hinaus, bspw. nach BB_6 zusätzlich zu den in die Region führenden Kontrollflussfehlern möglich.

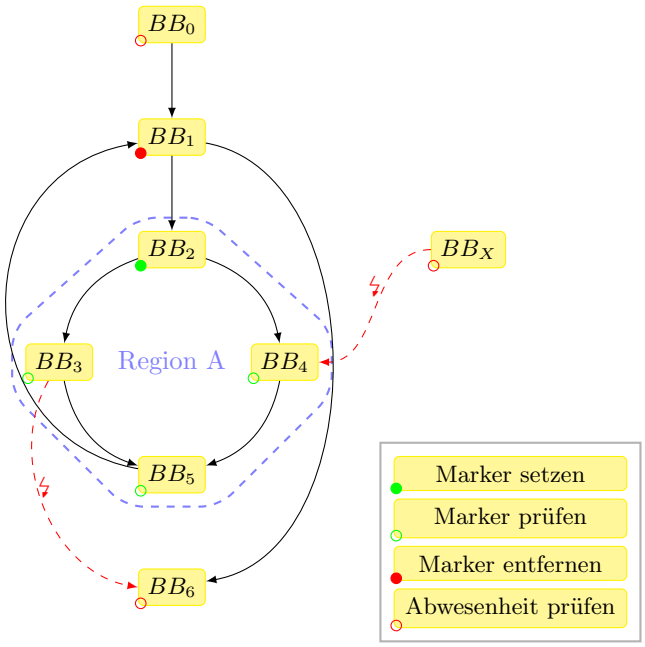


Abbildung 3.10 – Schematische Abbildung des instrumentierten Kontrollflussgraphen nach Optimierung, in Anlehnung an [7]. Durch die Optimierung wird auch der Übergang $BB_3 \rightarrow BB_6$ möglich.

Das so gebildete Verfahren ähnelt dabei dem Plain Inter-Block Error Detection-Verfahren, jedoch auf einer gröberen Granularitätebene und mit einer Vermeidung expliziter Zuweisungen durch den Einsatz von Bitoperationen.

3.3.6.3 Einstufung

Aufgrund der unterschiedlichen Granularität ist die in Abschnitt 2.2.4 aufgestellte Klassifikation nicht unmittelbar anwendbar, davon abgesehen ist die Beschaffenheit der Regionen von der jeweiligen Verteilungsstrategie der Bits an die Regionen abhängig. Dementsprechend wird hier auf eine Einordnung verzichtet.

Generell ist das Verfahren in der Lage, alle illegalen Übergänge zwischen unterschiedlichen Regionen zu erkennen. Die einzige Ausnahme bilden diejenigen eine Region verlassenden Sprünge, die zum Beginn eines der zur Dominanzgrenze der aktuellen Region gehörigen Knoten führt, da hier die korrekte Bitsetanpassung durchgeführt werden würde.

3.3.6.4 Umsetzung in KESO

Wie oben beschrieben, sieht der Autor in [7] eine Optimierung des Verfahrens mittels Dominanzgrenzen vor, welche jedoch im ursprünglichen Projekt aufgrund technischer Schwierigkeiten nicht umgesetzt wurde, da im in der ursprünglichen Veröffentlichung diskutierten Fall lediglich ausgewählte Bestandteile des Quelltextes manipuliert werden konnten. Zwar existieren auch in KESO aufgrund des in Abschnitt 1.2 beschriebenen *Weavelet*-Mechanismus einige Programmbestandteile, welche nicht mittels der in JINO umgesetzten Verfahren unmittelbar modifizierbar sind. Allerdings bewirken diese Bestandteile keinen Wechsel des Kontrollflusses zwischen unterschiedlichen Basisblöcken der Anwendung, somit sind sie für die Bestimmung von Dominanzgrenzen vernachlässigbar. Dementsprechend wurde die hier implementierbare, optimierte Variante in dieser Arbeit umgesetzt.

Wie in Abschnitt 3.3.6.3 angedeutet ist die Granularität abhängig von der Anzahl der verfügbaren Marker. Diese ist wiederum durch die Größe des den Zustand speichernden Speicherwortes limitiert. Für diese Arbeit wurde, da es sich bei den für die Messungen in Kapitel 4 verwendeten Prozessorarchitekturen um einen 32-Bitter handelt, ein `uint32_t` Datentyp gewählt, es stehen somit Bits für 32 Regionen zur Verfügung.

Weiterhin stellt sich vor diesem Hintergrund die Frage der Regionsauswahl für die Markervergabe. Die ursprüngliche Veröffentlichung schlägt hier zur Erhöhung der Genauigkeit vor, alle nichttrivialen Unterbäume des Dominatorbaums, also alle Unterbäume mit mehr als einem Basisblock, nach der Größe zu sortieren und die kleinsten Unterbäume mit Markern zu versehen. Der ursprüngliche Einsatzzweck der Kontrollflussüberwachung für die Absicherung der Systemaufrufe im Kontext eines statisch konfigurierbaren, *maßgeschneiderten* (engl. *tailored*) Betriebssystem deutet jedoch auf eine geringe, begrenzte Größe des dem Verfahren zugrundeliegen Kontrollflussgraphen an. Der primär von KESO abgedeckte Aufgabenbereich umfasst hingegen den Bereich der auf eingebetteten Systemen ausgeführten Anwendungen.

Dementsprechend weist die Größe der Kontrollflussgraphen in Abhängigkeit vom jeweiligen Programm eine deutlich höhere Variabilität auf, tendenziell ist eine höhere Größe zu erwarten.

In diesem Falle wirkt sich die beschriebene Zuweisung der Markerbits nach der Regionengröße aufsteigend jedoch negativ auf die Detektionsmöglichkeiten des Verfahrens aus, da die Markerbits nur auf die Randbereiche des Dominatorbaumes entfallen, aufgrund der geringen Markerzahl bleibt ein Großteil des Kontrollflussgraphen zwischen Wurzel und markierten Randbereichen ungekennzeichnet. Kontrollflussfehler zwischen Blöcken dieses Bereichs würden dementsprechend nicht erkannt, da sie sich eine Bitmaske teilen. Deshalb wurde im Laufe dieser Arbeit eine weiteres Verfahren zur Markervergabe entwickelt, welches eine möglichst gleichmäßige Abmessung der von den jeweiligen Bitmustern bezeichneten Gebiete anstrebt, d.h. die Anzahl an Basisblöcken, welche sich ein Muster teilen ist möglichst äquivalent. Die Problemstellung weist hier deutliche Ähnlichkeiten zur NP-Vollständigen min-max oder auch max-min Baumpartition auf [26]. Zur Implementierung der Auswahlstrategie wurde eine iterierte Variante des durch Kudu et al. in [27] beschriebenen Algorithmus zur optimalen K-partition von Bäumen verwendet. Die optimale K-partition eines Baumes mit gewichteten Knoten bezeichnet dabei die Unterteilung eines Baumes, sodass keiner der Unterbäume eine Knotensumme größer K besitzt. Mit einem uniformen Knotengewicht von eins lässt sich mit dem in Algorithmus 3.7 dargelegten iterativen Algorithmus eine möglichst uniforme Verteilung erreichen, welche als Heuristik eine gute Näherung an die ideale Partition liefert.

1: $partitions = \emptyset$
2: $partitionsize \leftarrow \left\lceil \frac{\left|\{b | b \text{ ist Basisblock}\}\right|}{\#_{Markerbits}} \right\rceil - 1$
3: **repeat**
4: $\quad partitionsize \leftarrow partitionsize + 1$
5: $\quad partitions \leftarrow optimalKPartition(dominatortree, partitionsize)$
6: **until** $|partitions| \leq \#_{markerbits}$

Algorithmus 3.7 – Pseudocode für die Baumpartition bei der Abdeckungs-Strategie

Die Berechnung sowohl des Dominatorbaumes, als auch der Dominanzgrenzen erfolgt in Anlehnung an die ursprüngliche Veröffentlichung [7] mittels der von Cytron et al. in [23] dargelegten Algorithmen.

In dieser Arbeit wurden beide Auswahlstrategien umgesetzt und sind durch die Optionen `cfm_domination_selection_smallest` für die nach der Größe sortierten Auswahl und `cfm_domination_selection_coverage` für die auf gleichmäßige Abdeckung bedachte Variante auswählbar.

Weiterhin lässt sich die Effizienz des Verfahrens, wenn auch auf Kosten der Detektionsfähigkeit, steigern, indem man die Überprüfung der Marker innerhalb einer Region lediglich einmal durchführt. Da eine Überprüfung sowieso am nächsten Regionsübergang durchgeführt wird, ist diese Auslassung nur problematisch, wenn dieser Übergang das Überqueren einer

Dominanzgrenze darstellt und sich die fehlerhafte Signatur nur genau im zur aktuellen Region gehörigen Bit von der idealen Signatur unterscheidet, ein Vorgang der hier optimistisch als eher unwahrscheinlich angenommen wird. Diese Optimierung wurde optional umgesetzt und kann durch die Option `cfm_domination_loopdetect` aktiviert werden. Die Auswertung in Kapitel 4 wird zeigen, ob die Detektionsmöglichkeiten durch diese Optimierung maßgeblich beeinträchtigt werden oder nicht.

3.4 Resümee

In diesem Kapitel wurden mehrere Kontrollflussüberwachungsverfahren vorgestellt. Dabei ist ein Anstieg des Aufwandes, aber auch der Detektionsmöglichkeiten der Verfahren von der Plain Inter-Block Error Detektion über ECCA und CFCSS hin zu YACCA zu beobachten. Aufgrund struktureller Probleme ist jedoch eine Umsetzung von ECCA im KESO-Projekt nicht möglich. Neben diesen recht feingranularen, aber dementsprechend auch teuren Verfahren auf Basisblockebene stellt weiterhin das dominatorbasierten Verfahren eine leichtgewichtige, grobgranularere, jedoch somit auch potentiell in den Detektionsmöglichkeiten eingeschränkte Alternative dar, welche vor allem durch ihre Effizienz bestechen sollte. Diese hier bisher rein theoretischen Einschätzungen müssen im nächsten Schritt experimentell überprüft werden, um die tatsächliche Eignung der Verfahren zu verifizieren.

Kapitel 4

Evaluation und Diskussion

Die theoretische, anhand der Kriterien in Abschnitt 2.2.4 durchgeführte Einstufung der vorgestellten Verfahren besitzt nur eine begrenzte Aussagekraft, da einerseits keine belastbaren Statistiken bezüglich der Wahrscheinlichkeit der unterschiedlichen Fehlerklassen existieren, als auch andererseits das durch den Übersetzter erzeugte Maschinenprogramm, bspw. durch die Einführung neuer Sprünge zur Übersetzung bestimmter Sprachkonstrukte, Abweichungen zum theoretisch analysierten Kontrollfluss aufweisen kann. Dementsprechend ist die experimentelle Überprüfung der Wirksamkeit der unterschiedlichen Verfahren zwingend nötig.

In Abschnitt 4.1 werden zunächst die verwendeten Testanwendungen vorgestellt, im Anschluss werden die Verfahren sowohl bezüglich Laufzeit- und Speicheraufwand in Abschnitt 4.1.2 als auch der Leistungsfähigkeit der erzielten Kontrollflussüberwachung in Abschnitt 4.3 getestet. Zuvor findet sich in Abschnitt 4.2.1 eine kurze Beschreibung der zur dieser Analyse verwendeten FAIL*-Fehlerinjektionsumgebung und zur Interpretation der damit gewonnenen Ergebnisse.

Zur Notation: Während diesem Kapitel müssen wiederholt die einzelnen Verfahren und im Falle des in Abschnitt 3.3.6 geschilderten dominatorbasierten Ansatzes auch auf die unterschiedlichen Konfigurationen dieses Verfahrens referenziert werden. Hierbei wird *Plain Inter-Block Error Detection* auch verkürzt als `Plaininterblock` bezeichnet, gleichsam bezieht sich `domination` auf den bereits erwähnten, dominatorbasierten Ansatz, die Variante `domination:small` bezeichnet die kleine Regionen bei der Auswahl bevorzugende Ausführung, während `domination:cover` die eine Gleichverteilung der Regionsgrößen anstrebende Implementierung angibt. Ein `:loop`-Suffix bezeichnet abschließend die in Abschnitt 3.3.6.4 beschriebene Optimierung durch Reduktion der Markerüberprüfung innerhalb der Regionen.

4.1 Testaufbau

Zur Bewertung dienen drei Beispielanwendungen:

- eine Matrixmultiplikation zweier 5x5 Matrizen
- das Sortieren eines Feldes von Ganzzahlen mit 42 Einträgen
- ein TCP-Zustandsautomat (nur reine Zustandslogik)

Die Anwendungen wurden nach verschiedenen Gesichtspunkten ausgewählt. Sowohl bei der Matrixmultiplikation als auch bei Sortieralgorithmen handelt es sich um gängige, in den zu den Verfahren gehörigen Papieren häufig verwendete Testalgorithmen, wodurch ein gewisser Grad an Vergleichbarkeit der Ergebnisse gewährleistet wird.

Gleichzeitig handelt es sich sowohl bei der verwendeten Implementierung des Sortieralgorithmus als auch bei dem Zustandsautomaten um verbreitete Implementierungen. Die Sortierfunktion entstammt dem GNU-Classpath Version 0.99, die Implementierung des Zustandsautomaten wurde mittels des State Machine Compilers, Version 6.3.0 erstellt.

Die Matrixmultiplikation verwendet dabei den naiven Standardalgorithmus mit drei verschachtelten Schleifen, die Matrizen werden als zweidimensionale Felder realisiert. Die Sortierung verwendet eine modifizierte Variante des Quicksort-Algorithmus, welcher für kleine Längen von unter sieben Einträgen auf Insertionsort zurückfällt. Die Zustandmaschine beschreibt den in Abbildung 4.1 abgebildeten Zustandsautomaten einer TCP-Verbindung, auf dem durch die Testanwendung 17 Zustandsübergänge ausgeführt werden.

Trotz der weiten Verbreitung der ersten beiden Anwendungen sowohl als Testanwendung als auch in produktivem Einsatz ist ihre Eignung für die Zwecke einer Evaluation von Verfahren zur Kontrollflusshärtung aufgrund der hohen Datenabhängigkeit nur bedingt gegeben. Als Gegenpol dient hier der im Wesentlichen kontrollflussabhängige Zustandsautomat.

Gleichsam unterscheiden sich die drei Programme bezüglich ihrer Verwendung von Funktionen. Während die Matrixmultiplikation keine Funktionsaufrufe tätigt, besteht die Implementierung des gewählten Sortieralgorithmus aus sieben Funktionen unterschiedlicher Länge. Anders als bei der Zustandsmaschine wird dabei jedoch auf virtuelle Aufrufe verzichtet. Die vom SMC erzeugte Implementierung des Automaten baut sehr stark auf Vererbung und der Überschreibung virtueller Methoden in Unterklassen auf, dabei werden 54 Methoden in 18 Klassen verwendet.

Dementsprechend bilden die ausgewählten Programme aufgrund ihrer Heterogenität eine repräsentative Auswahl an kurzen Java-Programmen, so es sich hierbei auch nicht um typische Anwendungen aus der Domäne der eingebetteten Systeme handelt.

In Tabelle 4.1 findet sich eine Auflistung einiger für die Kontrollflussüberwachung wichtigen Metriken der Anwendungen. Dabei überrascht die trotz der hohen algorithmischen und strukturellen Unterschiede zwischen den Anwendungen nur sehr geringe Varianz. So ist die durchschnittliche Blocklänge der datenintensiven Programme, Matrixmultiplikation und Sortierung, vergleichbar zu der des Zustandsautomaten. Während sich dies noch mit der

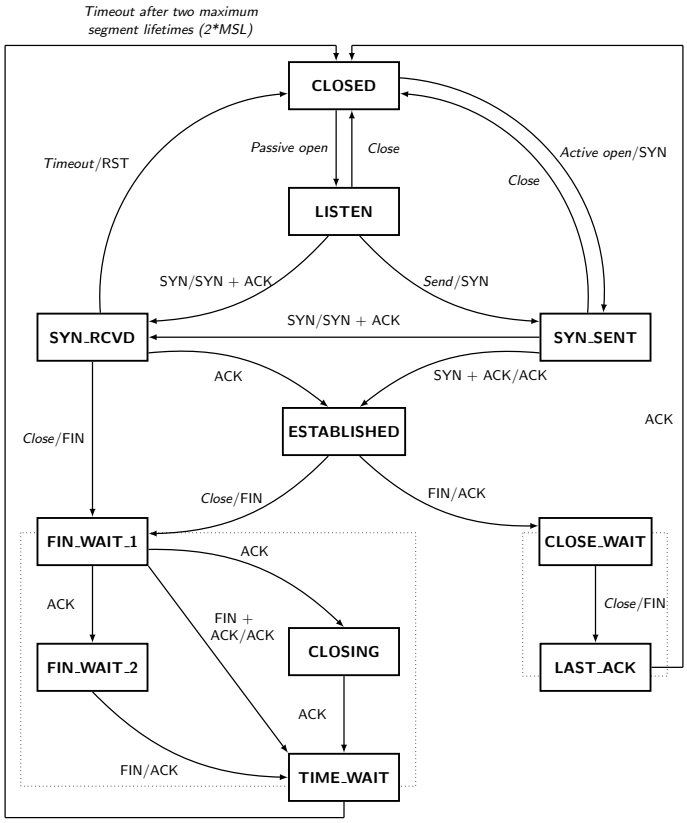


Abbildung 4.1 – Abbildung der umgesetzten TCP-Zustandsmaschine (Quelle: [28])

großen Menge an schleifenbedingten Sprüngen im Falle der Multiplikation als auch mit der allgemein hohen Zahl an Vergleichen für die Sortierfunktion erklären lässt, so ist die große Ähnlichkeit bei den Vorgänger- und Nachfolgerzahlen doch ungewöhnlich, da diese für die stark auf virtuellen Methodenaufrufen basierende Implementierung der Zustandsmaschine bedeutend höher liegen sollte. Aufgrund der durch die statische Konfiguration ermöglichten, vollständigen Daten- und Kontrollflussanalyse ist es JINO jedoch in vielen Fällen möglich, den korrekten Typ des zugrundeliegenden Objektes zu inferieren und die Zahl der potentiellen Nachfolger damit deutlich einzuschränken und somit auch die Zahl der Sammelknoten im Kontrollflussgraphen zu reduzieren. Begleitend ist bei der Zustandsmaschine innerhalb der Methoden die Kontrollflussdichte gering und somit werden dort mehrheitlich Basisblöcke mit nur einem einzigen Vorgänger und Nachfolger erzeugt, wodurch der hier betrachtete geringe Wert zustandekommt. Deutliche Unterschiede ergeben sich also lediglich bezüglich der Basisblockzahl und der Menge der Methoden und Methodenaufrufe.

	Matrixmultiplikation	Sortierung	Zustandsautomat
Anzahl Basisblöcke	123	203	226
Anzahl Methoden	2	5	42
Anzahl Funktionsaufrufe	6	10	67
∅ Blöcke/Methode	61,5	40,6	5,38
∅ JVM-Befehle/Block	4,89	6,25	5,10
∅ Vorgänger/Block	1,25	1,35	1,19
∅ Nachfolger/Block	1,30	1,38	1,30

Tabelle 4.1 – Gängige Metriken der Testanwendungen

4.1.1 Rahmenbedingungen

Eingebettete Systeme stellen üblicherweise eine besonders kosten- und somit auch aufwandssensitive Anwendungsdomäne, vor allem im Bezug auf Speicherbedarf und Rechenkapazität, dar. Aus diesem Grund ist der Einsatz optimierender Übersetzer in dieser Anwendungsdomäne üblich. Gleichzeitig wirken sich die hier verwendeten Konfigurationsparameter der beteiligten Programme direkt auf die erzeugten Maschinenprogramme und somit auch auf die zu beobachteten Messwerte aus, im Folgenden werden dementsprechend die in dieser Arbeit verwendeten Werte dokumentiert.

Im vorliegenden Fall wurden alle verwendeten Anwendungen mit dem JINO-Übersetzer, Revision 4298, übersetzt. Dabei wurden unnötige *Anweisungen zur Fehlerprotokollierung und Konsistenzprüfung* (engl. *Debugging Code*) mittels der `production`-Option deaktiviert und javatypische Laufzeitüberprüfungen, bspw. die Überprüfung von Feldindizes, durch `omit_safechk` abgeschalten. Dies soll eine isolierte Betrachtung der Kontrollflussüberwachung frei von den Effekten anderer Fehlerdetektionsmechanismen sicherstellen. Innerhalb des der Injektion unterzogenen Zeitraumes finden in keiner der betrachteten Anwendungen Allokationen statt, auch sonst wurden keine *Einwebungen* (engl. *Weavelets*) in nativen Programmtext aufgerufen.

Die so entstanden C++-Quelltexte der jeweiligen Applikation wurde im Anschluss mit dem die OSEK-Spezifikation umsetzenden CIAO Bibliotheksbetriebssystem [29], Commit `14defd2`, zu einem Gesamtsystem verbunden. Nach einer Vorverarbeitung durch den AspectC++-Übersetzer `ag++`, Version 0.8, erfolgten Übersetzung und Bindung mittels des zur GNU Compiler Collection gehörigen `g++`-Kompilierers auf den Standardeinstellungen der Optimierungsstufe `-O2`.

Für die Messung der Artefaktgrößen und der eigentlichen Fehlerinjekton wurde der Quelltext hierzu durch die `g++` Version 4.9.1 für die IA32-Architektur übersetzt. Da sich diese Architektur jedoch aufgrund diverser nur schwer kontrollierbarer Parameter nur bedingt für Laufzeitmessungen eignet wurde diese Metrik auf einem RISC-basierten Infineon TriBoard TC1796.302 erhoben. Die hierzu benötigten Binärdateien wurden dementsprechend

abweichend mit dem Tricore-g++-*Fremdübersetzter* (engl. *cross compiler*), Version 4.6.3, erzeugt.

Da sich in der zu den jeweiligen Verfahren gehörigen Literatur in der Regel bei den Messungen nur unzureichende Informationen zur Konfiguration der Arbeitsumgebung und der verwendeten Programme finden, ist die Vergleichbarkeit der Messungen im Allgemeinen nicht gegeben. So finden sich lediglich im Falle von Plain Inter-block Error Detection Angaben bzgl. der Übersetzeroptionen [1], allerdings wurde in diesem Falle explizit auf Optimierungen verzichtet.

4.1.2 Laufzeit und Größe

Der Anwendungsbereich der eingebetteten Systeme stellt traditionell ein besonders ressourcensensitives Umfeld dar, dementsprechend kritisch müssen Zunahmen in Speicherbedarf, Programmgröße und auch Laufzeit beurteilt werden.

Bezüglich des Speichermehraufwandes im Sinne von zusätzlich benötigtem Datenspeicher zur Laufzeit zeigen sich alle umgesetzten Verfahren sehr moderat, Speicher ist nur für die Signatur und im Falle von CFCSS noch zusätzlich für die Differenzsignatur nötig. Diese Variablen wurden hier als nicht vorzeichenbehaftete 32bit Ganzzahlen im Taskdeskriptor realisiert. Dies bedeutet einen nur geringen Zuwachs von insgesamt vier bzw. acht Byte pro OSEK-Task.

Deutlich schwerwiegender ist hingegen die Zunahme der Programmgröße durch die zusätzlich für die Verfahren eingeführten Instruktionen zur Verwaltung und Überprüfung der Signatur. In Tabelle 4.2 findet sich eine Aufstellung der Größen der vom Übersetzter erzeugten Binärdateien. Die angegebene Größe ist hierbei die Größe der reinen, von KESO übersetzten Anwendung, ohne das zugrundeliegende CIAO-Basissystem nach *Entfernung* (engl. *stripping*) überflüssiger Symbole und entsprechen ungefähr der Größe des Textsegmentes der Anwendung.

Während für die dominatorbasierten Verfahren und Plaininterblock noch moderate Zunahmen von etwas über 10 % bis zu 18,9 % bei der Sortieranwendung zu verzeichnen sind, und auch die `loop`-Optimierung nochmals eine Verbesserung dieser Werte um knapp die Hälfte erreicht, ist die Zunahme bei CFCSS und YACCA deutlicher. Dies lässt sich im Wesentlichen auf die Überprüfung der Vorgänger- und Nachfolgerinformation bei CFCSS und YACCA zurückführen, welche bei ersterem ungefähr eine Verdopplung des Speicherbedarfs zum kontrollflussagnostischen Plaininterblock-Verfahren darstellt. Bei YACCA finden sich derartige Befehlssequenzen neben dem Prolog auch noch im Epilog der Basisblöcke, was wiederum einen Zuwachs um den Faktor zwei im Vergleich zu CFCSS bedingt.

Vergleicht man diese Werte soweit vorhanden mit Werten aus der Literatur, so zeigt sich ein differenziertes Bild. Die für das CFCSS-Verfahren im zugehörigen Papier [4] vorhanden Vergleichswerte zeigen für die Matrixmultiplikation ein Zusatzaufwand von 41,4 % und für Quicksort von 55,9 % an, welche von der hier gezeigte Umsetzung leicht unterboten

werden, jedoch eine vergleichbare Größenordnung liefern. Diese Abweichung lässt sich dabei auf die Unterschiede zwischen der dort verwendeten MIPS-Architektur im Bezug auf die bei der hier verwendeten Zielarchitektur deutlich kompakteren Instruktionskodierung zurückführen. Auch die zu vermutete leicht größere Basisblockgröße in der hier vorliegenden Implementierung aufgrund des indirekten Speicherzugriffs auf die Felder, welche in Java auf der *Halde* (engl. *heap*) abgelegt sind, stützt diesen Effekt. Anders stellt sich die Situation für die im zu YACCA gehörigen Papier [5] für eine 5x5 Matrixmultiplikation angegebenen Werte mit einem Zusatzbedarf von 261 % für CFCSS und 191 % für YACCA dar. Während sich die Unterschiede um fast eine Größenordnung gegebenenfalls durch abweichende Übersetzeroptionen, vor allem bezüglich des Optimierungsgrades, erklären lassen würden, auch wenn die zugehörigen Informationen im Papier fehlen, so steht zumindest das Verhältnis zwischen den Verfahren in der referenzierten Veröffentlichung und hier in einem deutlichen Missverhältnis. Vor dem Hintergrund des deutlichen Mehraufwandes von YACCA durch zusätzliche Überprüfungen im Epilog ist eine Aufwandsabnahme von CFCSS hin zu YACCA einfach nicht plausibel, gleichzeitig fehlen für eine sinnvolle Analyse hier jedoch weitere Angaben zur Art der Messung und Erzeugung der betrachteten Binärdateien.

Neben der Größe ist weiterhin die zusätzlich aufzuwendende Laufzeit entscheidend, in Tabelle 4.3 finden sich die Ergebnisse der zugehörigen Messung. Wie in Abschnitt 4.1.1 erwähnt, wurden diese Messungen auf einem TriBoard TC1796 durchgeführt. Dieses bietet durch seine für den Einsatz in Echtzeitsystemen konzipierte RISC-Architektur eine im Hinblick auf Zeitmessungen deutlich kontrolliertere Ausführungsumgebung. Die eigentliche Zeitmessung erfolgte über den im Prozessor bereitgestellten *Zeitnehmer* (engl. *System Timer*) `STM0`.

Die verwendete Entwicklerplatine bietet dabei zwei Datenspeicher. Die Programmdaten lagen für die hier durchgeführten Messungen in dem 64KB fassenden internen LDRAM,

	Matrixmultiplikation		**Sortierung**		**Zustandsautomat**	
	Größe (Byte)	Faktor	Größe (Byte)	Faktor	Größe (Byte)	Faktor
Referenz	25934	100,0 %	27558	100,0 %	61320	100,0 %
dom:cover:loop	27122	104,5 %	29302	106,3 %	64892	105,8 %
dom:small:loop	27698	106,8 %	29906	108,5 %	65848	107,3 %
plaininterblock	28714	110,7 %	32378	117,4 %	70756	115,3 %
dom:small	28878	111,3 %	32418	117,6 %	69096	112,6 %
dom:cover	29050	112,0 %	32774	118,9 %	69756	113,7 %
cfcss	31610	121,8 %	37494	136,0 %	74680	121,7 %
yacca	40746	157,1 %	51506	186,9 %	92520	150,8 %

Tabelle 4.2 – Artefaktgrößen der Anwendung nach Härtung mittels unterschiedlicher Verfahren, sowohl absolut als auch relativ zur unmodifizierten Referenzanwendung

	Matrixmultiplikation		Sortierung		Zustandsautomat	
	Laufzeit (Ticks)	Faktor	Laufzeit (Ticks)	Faktor	Laufzeit (Ticks)	Faktor
Referenz	4576	100,0 %	4750	100,0 %	1437	100,0 %
dom:cover:loop	10617	232,0 %	16175	340,5 %	4908	341,5 %
plaininterblock	13993	305,7 %	15367	323,5 %	5166	359,4 %
dom:cover	15122	330,4 %	20733	436,4 %	6605	459,6 %
dom:small:loop	19102	417,4 %	23835	501,7 %	3521	245,0 %
dom:small	21048	459,9 %	27484	578,6 %	5578	388,1 %
cfcss	26707	583,6 %	43549	916,8 %	10007	696,3 %
yacca	63579	1389,4 %	96589	2033,4 %	25587	1780,5 %

Tabelle 4.3 – Laufzeiten der verschiedenen Anwendungen nach Härtung mittels unterschiedlicher Verfahren, Datenspeicher im internen LDRAM

welcher schnelle Zugriffe in nur zwei Prozessor-Zyklen bietet [20]. Dieser schnelle Zugriff ist wichtig, da hier die gewählte Umsetzung der Laufzeitsignaturen als `volatile` Variable bei jedem Zugriff die Neuauswertung und somit auch einen erneuten Abruf der Werte aus dem Speicher erzwingt. Diese Umsetzung stellt jedoch die einzige Möglichkeit dar, portabel auf der Sprachebene von C eine Elimination der im korrekten Kontrollfluss redundanten Überprüfungen zu verhindern. Gleichzeitig ist der erneute Abruf auch für all diejenigen Sprünge wichtig, welche in die zur Kontrollflussüberwachung gehörigen Instruktionsgruppen springen, da nur so die tatsächliche Überprüfung der Signatur im Gegensatz zum Vergleich zufälliger Registerwerte sichergestellt werden kann.

Diese Zugriffe schlagen sich auch deutlich in den Laufzeiten nieder, selbst das je nach Anwendung leichtgewichtigste Verfahren bedingt einen Zusatzaufwand von mindestens 130 %. Generell zeigen sich Unterschiede sowohl zwischen den jeweiligen Anwendungen als auch zwischen den Verfahren. Generell sind die Zunahmen für die Sortieranwendung mit am höchsten. Dieser Umstand lässt sich auf die unterschiedlichen Größen der Basisblöcke zurückführen. Zwar sind die mittleren Blockgrößen über alle Anwendungen hinweg ungefähr gleich, jedoch wurde in der Laufzeitmessung nur ein geringer Teil der Anwendung, die eigentliche Sortierung und nicht das befüllen oder die abschließende Überprüfung der Werte betrachtet. Da bei der Sortierung algorithmusbedingt viele kleine Basisblöcke entstehen, ist diese hier messbare Zunahme durch diesen Effekt schlüssig erklärbar.

Auch zwischen den Verfahren zeigen sich deutliche Unterschiede. Generell existiert ein deutlicher Unterschied zwischen den noch eher leichtgewichtigen Verfahren Plaininterblock und den dominatorbasierten Ansätzen sowie den aufwandsintensiven Vertretern CFCSS und YACCA. Während CFCSS mit Zunahmen von 480 % bis ca. 910 % möglicherweise gerade noch vertretbar scheint, wird besonders YACCA für die wiederholten, häufigen Signaturabrufe bestraft und scheint dementsprechend mit einem Zusatzaufwand in der Laufzeit von 1290 %

bis 1900 % im praktischen Einsatz nicht mehr vertretbar. Diese hohen Werte wirken vor dem Hintergrund der in der dem YACCA-Papier [5] von 2003 angegebenen 147 % unnatürlich hoch. Gegebenenfalls wurde in diesem Fall die hohe Zugriffszahl durch Verwendung eines dedizierten Registers für die Laufzeitsignatur abgefangen, gleichwohl handelt es sich hier um eine reine Spekulation, da im Papier die nötigen Informationen fehlen.

Für die dominatorbasierten Varianten lässt sich keine feste Ordnung bezüglich der Laufzeiten für die unterschiedlichen Selektionsverfahren angeben. Der Aufwand bei diesem Verfahren hängt maßgeblich davon ab, wie viele Regionen in die "heißen" Schleifen der Multiplikation oder Sortierung gelegt werden, da hier der größte Teil der Laufzeit verbracht wird. Dies schwankt je nach genauer Form und Regionsgröße der Schleifen doch deutlich, woraus sich die über die Anwendungen hinweg zu beobachtenden Schwankungen ergeben. Jedoch wird durch die `loop`-Optimierung auch hier eine deutliche Verbesserung erzielt.

Allgemein weisen die Messungen beider Metriken, Laufzeit und Größe, einen hohen Korrelationsgrad auf, welcher sich dadurch erklärt, dass jede durch die Verfahren in das Programm eingefügte Instruktion in der Regel auch mindestens einmal ausgeführt wird. Während jedoch die Artefaktgröße in diesem Falle einen linearen Zusammenhang zur Zahl der zusätzlichen Anweisungen aufweist, so können die Befehle, z.B. in Schleifen auch mehrfach ausgeführt werden, wodurch der Effekt deutlich verstärkt wird.

Somit ergibt sich generell bezüglich des Aufwandes eine Ordnung der Verfahren mit Plaininterblock und den dominatorbasierten Verfahren ungefähr gleichauf am einen Ende über CFCSS hin zu YACCA, welches mit je nach Anwendung 50 % bis 87 % Zunahme in der Größe und 1290 % bis 1900 % Zusatzaufwand in der Laufzeit eher wenig praktikabel wirkt.

4.2 Fehlerinjektion

Da die Art der Experimentdurchführung und die Wahl der Injektionsstellen maßgeblich die Ergebnisse und ihre Interpretation beeinflussen, wird zur besseren Einordnung im Folgenden ein kurzer Überblick über die Funktionsweise von FAIL* gegeben, bevor auf einer klaren Definition des Fehlermodells aufbauend in Abschnitt 4.2.3 daraus mittels eines Gedankenexperiments eine Interpretationssemantik für die anfallenden Messwerte abgeleitet wird.

4.2.1 Die FAIL* - Fehlerinjektionsumgebung

Fault Injection Leveraged (FAIL*) ist eine Programmsammlung, welche die deterministische Durchführung von Fehlerinjektions-Experimenten auf einer Vielzahl unterschiedlicher Hardwarevarianten mittels Emulatoren ermöglicht [30]. Die in dieser Arbeit genutzte Ausführung, eine modifizierte Variante von FailBochs, verwendet dabei den Bochsemulator um diese Funktionalität für x86-basierte Programme bereitzustellen.

Die Durchführung einer Injektionskampagne ist dabei in zwei Schritte gegliedert, die *Ablaufprotokollierung* (engl. *tracing*) und die eigentliche *Fehlerinjektion* (engl. *fault injection*). Im ersten Schritt wird eine unbeeinträchtigte Ausführung des Programms inklusive aller Speicher- und Registerzugriffe aufgezeichnet, sie stellt die sog. *Referenzfolge* (engl. *golden run*) dar. Im nächsten Schritt wird festgelegt, an welchen Punkten in der Referenzfolge in welchen Datensätzen welche Arten von Fehlern erzeugt werden sollen. Da diese Entscheidung maßgeblich vom zugrundeliegenden Fehlermodell abhängig ist, wird später in Abschnitt 4.2.2 das in Abschnitt 2.2.5 definierte Modell nochmals, vor allem in Hinblick auf den Einsatz in FAIL*, konkretisiert. Für jedes ausgewählte, zu injizierende Fehlermuster startet FAIL* im Folgenden eine neue Ausführung des Programms, stoppt am Injektionszeitpunkt, führt die dem Fehlerbild entsprechenden Veränderungen am Programmzustand durch und setzt die Ausführung fort. Ab diesem Zeitpunkt ist der Experimentverlauf ereignisgesteuert, d.h. das Experiment läuft weiter bis eine der im Vorfeld festgelegten Befehlssequenzen, im Folgenden *Marker* genannt, ausgeführt wird. Das Auftreten eines Markers signalisiert dabei das entsprechende Ergebnis der jeweiligen Fehlerinjektion.

Zur Auswahl der Injektionspunkte sind zwei Vorgehensweisen üblich, das *Stichprobenverfahren* (engl. *sampling*) und die *vollständige Erfassung* (engl. *full scan*) [31]. Während erstere eine Menge zufälliger Programmpunkte auswählt, wird bei der in dieser Arbeit verwendeten vollständigen Erfassung für jede ausgeführte Instruktion jeder gemäß dem Fehlermodell mögliche Fehler injiziert.

Um die dabei entstehende Zustandsexplosion abzufangen wird hierbei eine *Definitions-Benutzungs-Reduktion* (engl. *def/use pruning*) verwendet. Ihr liegt die Idee zugrunde, dass eine bestimmte fehlerbedingte Veränderung eines Datums, z.B. ein Bitkipper an einer bestimmten Position, unabhängig vom Zeitpunkt innerhalb des Intervalls zwischen dem letzten Schreib- und dem nächsten Lesezugriff auf dieses Datum die gleiche Auswirkung auf den Anwendungsverlauf besitzt, alle Fehlerereignisse in diesem Zeitrahmen bilden somit eine Äquivalenzklasse. Eine Injektion eines genauen Fehlermusters muss für jede Äquivalenzklasse eines Datums nur einmal durchgeführt werden. Durch Gewichtung der Resultate mit der Größe der Äquivalenzklasse ergeben sich dabei dennoch die gleichen Ergebnisse wie bei einer tatsächlichen vollständigen Abtastung. Diese Gewichtung ist dabei ein Maß für die Wahrscheinlichkeit der einzelnen Ereignisse, welche genutzt werden kann, um eine belastbare Interpretation der Injektionsergebnisse zu gewährleisten.

4.2.2 Fehlermodell und Auswertung

Wie bereits in Abschnitt 2.2.5 erwähnt beruht das verwendete Fehlermodell auf einer uniformen Verteilung von stochastisch unabhängigen Einbitfehlern. Dabei wurden die Fehler in drei Gruppen eingeteilt, diejenigen, welche den Befehlsszeiger direkt modifizieren, solche welche andere Register betreffen und solche, welche Veränderungen im Speicher auslösen. Für den größten Teil der Auswertung wird die Injektion des Befehlszeigers, die *IP-Benchmark*,

betrachtet, da sie der in Abschnitt 2.2 dargelegten Definition eines Kontrollflussfehlers entspricht. Das Textsegment soll dabei von der Injektion ausgenommen sein.

Die unterschiedlichen Ergebnisarten eines Experimentdurchganges ergeben sich dabei aus den Eigenschaften der zugrundeliegenden Hardware einerseits und den semantisch je Anwendung festgelegten Regel bezüglich der Korrektheit eines Ergebnisses andererseits. Zu erster Kategorie gehören im hier vorliegenden Fall die Ereignisse `TRAP`, für vom simulierten Prozessor ausgelöste Ausnahmesituationen, `WRITE_TEXTSEGMENT` im Falle eines versuchten Schreibzugriffs auf das als rein lesbar angenommene Textsegment und `WRITE_OUTERSPACE` für Schreibzugriffe auf im Adressraum nicht vorhandenen, aber adressierbaren Speicher. Falls der injizierte Fehler am Befehlszähler zu einer versuchten Befehlsausführung aus einem solchen nicht verfügbaren Speicherbereich führt, wird dies mit `INJECTION_FAILURE` registriert.

Daneben wird für jedes Experiment eine Reihe von Funktionen bereitgestellt, deren Ausführung ein bestimmtes Ergebnis des Experiments darstellt. Diese *Marker* werden am Ende des eigentlichen Programms, in einem von der Injektion ausgenommen Programmteil abhängig vom Programmzustand aufgerufen. So wird dort für die Sortieranwendung das zugrundeliegende Feld beispielsweise auf korrekte Anzahl, Position und Wert der Einträge geprüft, bei der Matrixmultiplikation werden die korrekten Werte der Ergebnismatrix und für die Zustandmaschine der Endzustand und die Anzahl der durchgeführten Zustandsübergänge abgeglichen. Im Fall eines erfolgreichen Abschlusses wird dann mittels des `OK_MARKERS` signalisiert, dass der Fehler keine Auswirkung auf das Berechnungsergebnis besaß. Sollte der Kontrollfluss diesen Punkt hingegen mit einem falschem Berechnungsergebnis erreichen, wird der unentdeckte Fehler mit dem Ergebnis `FAIL_MARKER` signalisiert.

Falls eines der hier umgesetzten Verfahren den Fehler erkannt hat, so wird durch das Verfahren die Fehlerbehandlung aufgerufen, dieser Vorgang wird durch das Resultat `DETECTED_MARKER` vermerkt.

Eine Sonderstellung nehmen jedoch die Fälle ein, in denen der Kontrollfluss aufgrund des Fehlers weder eine hardwarebedingte Ausnahmesituation auslöst noch das Programmende erreicht, sondern sich bspw. in einer Endlosschleife verfängt. Um derartige Fälle auszuschließen wurden alle Anwendungen, welche nach fünf Sekunden, einem Vielfachen der normalen Programmlaufzeit, keinen der oben beschriebenen Zustände erreichten mit dem Ergebnistyp `TIMEOUT` beendet. Vor dem Hintergrund eines Einsatzes in einem harten Echtzeitsystem sind derartige Vorkommnisse jedoch nur von untergeordneter Bedeutung, da hier diese Fehler durch die *Laufzeitüberwachungseinheit* (engl. *Watchdog*) erkannt würden.

Da mit Ausnahme der `FAIL_MARKER`-Ereignisse alle Ergebnisse in einen für die Anwendung sicheren Zustand führen, muss das Ziel der Kontrollflussüberwachung sein, die Anzahl dieser Ereignisse zu reduzieren.

4.2.3 Metriken: Fehlerrate und -abdeckung

Bei der Durchführung von Experimenten stellt sich immer auch die Frage der Auswertung und Interpretation. In der Literatur ist hier für Fehlerinjektionsexperimenten die *Fehlerabdeckung* (engl. *fault coverage*) gebräuchlich, so wurde sie zur Evaluation von Plain Inter-Block Error Detection [1] [2], CFCSS [4], ECCA [3] und YACCA [5] [22] herangezogen. Auch in der den dominatorbasierten Ansatz beschreibenden Veröffentlichung, welche jedoch einen Schwerpunkt auf die Datenintegrität legt, findet sich ein vergleichbares Konstrukt unter dem Begriff der *Silent Data Corruption Rate*. Diesen Metriken gemein ist die Betrachtung der jeweiligen Ereignisse ausschließlich relativ zur Gesamtzahl aller für die jeweilige Variante durchgeführten Injektionsexperimente. Diese Praxis maskiert jedoch die unterschiedlichen Dimensionsgrößen der der Messung zugrundeliegenden Fehlerräume, welche sich aus den Unterschieden zwischen den unmodifizierten und gehärteten Programmen bezüglich Programmgröße und Laufzeit ergeben. Schirmeier et al. führen in [31] eine Analyse dieser Problematik vor dem Hintergrund der Datenfehler durch, die Aussagekraft der Fehlerabdeckungsmetrik wird dort mittels eines einfachem Gedankenexperimentes widerlegt. Die dort geschilderte Überlegung lässt sich in abgewandelter Form auch auf die vorliegende Problematik der Härtung gegen Kontrollflussfehler übertragen:

Dazu sei zunächst ein hypothetisches, signaturbasiertes oder funktional äquivalentes, Verfahren zur Kontrollflussüberwachung, *HYPO*, gegeben, welches die Fehlertypen 1-4 gemäß der Kategorisierung aus Abschnitt 2.2.4 zuverlässig erkennt. Das heißt, es werden sämtliche Kontrollflussfehler erkannt, welche vom Ende eines Basisblocks zum Beginn eines anderen (Typ 1 oder Typ 2), vom Ende eines Basisblocks in die Mitte eines anderen (Typ 3) oder von der Mitte eines Basisblocks zum Anfang oder der Mitte eines anderen führen (Typ 4). Unerkannt bleiben somit ausschließlich fehlerhafte Übergänge von Typ 5, also Übergänge innerhalb des Basisblocks selbst.

Über einem mit diesem Verfahren gehärteten Programm P werde nun eine Fehlerinjektion von Kontrollflussfehlern vorgenommen. Ohne Beschränkung der Allgemeinheit werden hierzu in jedem Ausführungsschritt alle Einbitfehler in den Instruktionszeiger injiziert, hierbei wird also der gesamte Fehlerraum der Kontrollflussfehler überprüft (*full scan*). Vor diesem Hintergrund ergibt sich die Fehlerrate F_{HYPO} zu:

$$F_{HYPO}(P) = \frac{\#_{HYPO}(\text{unerkannter Fehler})}{\#_{HYPO}(\text{injizierter Fehler})} = \frac{\#_{HYPO}(\text{Fehler von Typ 5})}{\#_{HYPO}(\text{injizierter Fehler})} \tag{4.1}$$

$\#(X)$ bezeichnet hier die Anzahl des Ereignisses X. Die Fehlerabdeckung FC ergibt sich als das Gegenereignis der Fehlerrate wie folgt:

$$FC_{HYPO}(P) = 1 - F_{HYPO} = 1 - \frac{\#_{HYPO}(\text{unerkannter Fehler})}{\#_{HYPO}(\text{injizierter Fehler})} \tag{4.2}$$

Nun werde HYPO wie folgt modifiziert: Am Beginn eines jeden Basisblocks, noch vor den zu HYPO gehörigen Instruktionen im Prolog, wird nun eine Folge von leeren Anweisungen, sogenannten Nulloperationen oder auch NOPs, eingefügt. In Anlehnung an

Schirmeier et al. [31] wird das so entstandene Verfahren im Folgenden als *diluted HYPO* (*DHYPO*, verwässertes/gestrecktes HYPO) bezeichnet.

Das oben betrachtete Programm P wird anschließend mittels DHYPO gehärtet und erneut dem wie oben beschriebenen Fehlerinjektionsexperiment unterzogen. Da innerhalb der Basisblöcke von DHYPO keine Veränderungen durchgeführt wurden, die Größe also konstant ist, gilt trivialerweise:

$$\#_{DHYPO}\,(\text{Fehler von Typ 5}) = \#_{HYPO}\,(\text{Fehler von Typ 5}) \tag{4.3}$$

Einzig durch die aufgrund der eingefügten Instruktionen veränderte Ausrichtung könnten leichte Abweichungen entstehen, diese lassen sich jedoch durch eine angepasste Binderkonfiguration beheben.

Ansonsten müssen alle neu entstandenen Sprünge aus dem von den eingefügten leeren Instruktionen beanspruchten Bereich, im Folgenden auch als *Rutsche* (engl. *NOP sled, NOP slide, NOP ramp*) bezeichnet, heraus oder in diesen hinein betrachtet werden. Diese lassen sich grob in vier Kategorien einordnen. Zunächst existieren Kontrollflussfehler, die aus der Rutsche wieder direkt in diese selbst zurückführen, es werden also lediglich einige der Nulloperationen übersprungen oder mehrfach ausgeführt. Da diese Befehle jedoch keinen Einfluss auf die Daten der eigentlichen Berechnung nehmen, erzeugen diese Abweichungen im Kontrollfluss keine beobachtbaren Fehler im Ergebnis. Lediglich die Laufzeit wird leicht verändert.

Weiterhin sind injizierte Fehler zu betrachten, welche aus der Rutsche zum Beginn oder mitten in die Befehlsfolge eines Basisblocks springen. Für die Zwecke der Kontrollflussüberwachung entspricht das erste Szenario einer fehlerhaften Verzweigung ausgehend vom Ende des unmittelbaren Vorgängerbasisblocks, also einen Fehler der Ordnung Typ eins oder Typ zwei, welche von HYPO und dementsprechend auch DHYPO erkannt werden. Der zweite Fall, also der Sprung in die Instruktionsfolge eines Basisblocks, entspricht einem Fehler des Typs drei, auch dieser wird detektiert.

Sprünge in den Bereich der durch HYPO eingefügten Nulloperationen sind semantisch äquivalent zu Sprüngen an den Beginn eines Basisblocks, der Kontrollfluss läuft gewissermaßen schlicht die NOP-Rutsche entlang. Je nach Ursprungsort des Sprunges entspricht dies einem Fehler des Typs eins, zwei oder vier. Auch hier wird der fehlerhafte Übergang durch HYPO zuverlässig erkannt.

Der vierte Fall, also der Übergang aus einer Rutsche in den NOP-Prolog eines anderen Basisblocks ist eine spezielle Ausprägung der letzten beiden geschilderten Fälle. Es handelt sich also semantisch um den Übergang vom Ende eines Basisblocks zum Anfang eines anderen.

Somit bleiben beim Injektionsexperiment alle Ereignisse, welche die von DHYPO eingefügten Nulloperationen betreffen, entweder als ohne Einfluss auf das Ergebnis, da sie eine selbstreferentiellen Sprunges aus einer NOP-Region zurück in die Rutsche selbst repräsentieren, oder werden ansonsten vom Verfahren als Kontrollflussabweichung detektiert.

Die Anzahl der unerkannten Fehler beim Übergang von HYPO zu DHYPO bleibt also konstant. Da jedoch mehr Instruktionen ausgeführt werden, und sich für jede der zusätzlichen Instruktionen im Injektionsexperiment die Zahl der injizierten Fehler erhöht gilt:

$$F_{HYPO}(P) = \frac{\#_{HYPO}(\text{unerkannter Fehler})}{\#_{HYPO}(\text{injizierter Fehler})} = \frac{\#_{HYPO}(\text{Fehler von Typ 5})}{\#_{HYPO}(\text{injizierter Fehler})} \tag{4.4}$$

$$> \frac{\#_{DHYPO}(\text{Fehler von Typ 5})}{\#_{DHYPO}(\text{injizierter Fehler})} = \frac{\#_{DHYPO}(\text{unerkannter Fehler})}{\#_{DHYPO}(\text{injizierter Fehler})} \tag{4.5}$$

$$= F_{DHYPO}(P) \tag{4.6}$$

Der relative Anteil der unentdeckten Fehler wurde durch DHYPO im Vergleich zu HYPO gesenkt, und somit die Fault-Coverage erhöht. Dieser Effekt lässt sich durch Verlängerung der NOP-Rutschen beliebig verstärken, die Fehlerabdeckung sich also beliebig nah zum Ideal der vollständigen Abdeckung von 100 % rücken, obwohl das Verfahren keine funktionale Verbesserung der Detektionsfähigkeiten gegenüber $HYPO$ erbringt.

Somit verbleibt lediglich der Nachweis, dass ein zu HYPO idempotentes Detektionsverfahren existiert. Das in Abschnitt 3.3.5 vorgestelle YACCA, mit der zusätzlichen Vergleichsverdopplung nach bedingten Sprüngen, besitzt in der Theorie die benötigten Eigenschaften. Die Konstruktion des oben geschilderten Gedankenexperimentes ist somit korrekt, die Schlüsse folglich zulässig. Der Vollständigkeit halber sei angemerkt, dass auf Architekturen, welche Instruktionskodierungen variabler Länge verwenden, YACCA die beschriebenen Detektionsfähigkeiten nicht vollständig besitzt. Im Falle eines Sprunges in die Mitte der Kodierung einer Instruktion kann sich dort die Interpretation des Programmes oder zumindest eines Programmabschnittes soweit verändern, dass der Fehler unerkannt bleibt. Aufgrund dieser Vorgänge ist der hier beschriebene Effekt somit mittels FailBochs nicht beobachtbar. Dies ist teilweise auch auf die geringen Länge der `nop`-Befehlskodierung für die IA32-Architektur von einem Byte [32] zurückzuführen, da somit in der Region hinter einer Rutsche alle Adressen in Einbyteinkrementen injiziert werden, die mittlere Länge der Instruktionskodierungen jedoch bei den hier betrachteten Anwendungen oberhalb von drei Byte pro Instruktion liegt, zwei von drei injizierten Sprüngen sind also im Mittel unausgerichtet. Dies ändert jedoch nichts an der allgemeinen Gültigkeit des Gedankenexperiments.

Dieses Gedankenexperiment zeigt deutlich, dass die Fehlerabdeckung als Metrik zur Evaluation von Verfahren zur Kontrollflusshärtung ungeeignet ist, da sich die Dimensionen der Fehlerräume der zu vergleichenden Programmvarianten unterscheiden, die durch eine längere Programmlaufzeit steigende Fehlerwahrscheinlichkeit jedoch unberücksichtigt bleibt. Dementsprechend werden in dieser Arbeit, wie auch durch Schirmeier et al. in [31] empfohlen, die absoluten Fehlerzahlen aus einer vollständigen Injektion der Fehlerraumes der unterschiedlichen Varianten verwendet. Da bei der vollständigen Abstastung für jede Instruktion eine Fehlerinjektion stattfindet, wird für jeden über die eigentliche Laufzeit der ungehärteten Anwendung hinausgehenden zusätzlichen Befehl eine neue Fehlermöglichkeit angenommen, die Metrik ist somit implizit mit der Laufzeit gewichtet. Somit stellt dieses

Vorgehen sicher, dass die durch die veränderte Laufzeit beeinflusste Fehlerwahrscheinlichkeit berücksichtigt wird.

4.3 Messergebnisse und Auswertung

Auch wenn, wie in Abschnitt 4.2.3 dargelegt, die Fehlerrate als solche keine Vergleichsgrundlage für die Wirksamkeit der unterschiedlichen Verfahren liefert, so ist sie doch eine valide Metrik zur Beurteilung der allgemeinen Anfälligkeit der ungehärteten Anwendung gegenüber Kontrollflussfehlern. In Abbildung 4.2 finden sich die Verteilungen der unterschiedlichen Ergebnistypen für die hier betrachteten Anwendungen bei einer Injektion des Befehlszählers. Dabei ist für die Zustandsmaschine eine deutlich geringere Fehlerrate als bei der Matrixmultiplikation oder auch der Sortierung zu beobachten; dieser Zusammenhang legt eine höhere Anfälligkeit der eher datenabhängigen Anwendungen nahe. Allerdings ist anzumerken, dass diese Werte deutlich von den dem Experiment zugrundeliegenden Definitionen des jeweiligen Berechnungsergebnisses abhängen. Während für die Zustandsmaschine lediglich zwei Daten, der Endzustand und die Anzahl der Zustandsübergänge betrachtet wurden, ist die Datenmenge für Multiplikation und Sortierung in Form der Ergebnismatrix und dem zu sortierenden Feld deutlich größer. Im vorliegenden Fall entspricht diese Unterscheidung aber auch genau der zwischen Daten- und Kontrollflussabhängigkeit bei den Anwendungen. Die zugrundeliegenden Korrektheitskriterien trennen dabei zwischen unentdeckten Fehlern und Fehlern ohne Einfluss auf das Berechnungsergebnis (`OK_MARKER`), insofern gelten für dieses Ergebnis die gleichen Überlegungen.

Die Aufteilung zwischen der Summer aus `OK`- und `FAIL_MARKER` zu den übrigen Ergebnistypen entspricht dabei der Eigenschaft der Anwendung nach einem Kontrollflussfehler wieder auf den regulären, von der fehlerfreien Anwendung verwendeten Kontrollflusspfad, wenn auch an einer veränderten Position, zurückzukehren und das Programmende zu erreichen. Dieser Umstand ist maßgeblich von der genauen Struktur, den vorhandenen Instruktionssequenzen und der Anordnung des geladenen Programmes im Speicher abhängig. Allgemeine Aussagen sind dementsprechend schwierig und inhärent anwendungsabhängig. So sind beispielsweise für die beiden ungültigen Schreiboperationen auf nicht vorhandenen (`WRITE_OUTERSPACE`) oder zum Textsegment gehörigen (`WRITE_TEXTSEGMENT`) Speicher maßgeblich von der Menge und verwendete Adressierungsart der in der Anwendung vorhandenen Schreibzugriffe abhängig, welche mit einer ungünstigen Registerbelegung durch einen Kontrollflussfehler angesprungen werden könnten.

Lediglich der Anteil der Injektionsfehler (`INJECTION_FAILURE`) ist experimentbedingt über alle Anwendungen hinweg konstant, dies impliziert einen linearen Zusammenhang zur Laufzeit der Anwendung. Die vollständige Übereinstimmung der relativen Anteile der Injektionsfehler ergibt sich dabei direkt aus den angenommenen Eigenschaften des simulierten Rechnersystems und der Art der Injektion. Das Zielsystem verfügt hier über 16 MiB Speicher, welcher am unteren Ende des Adressraums eingeblendet ist und sowohl das Text-

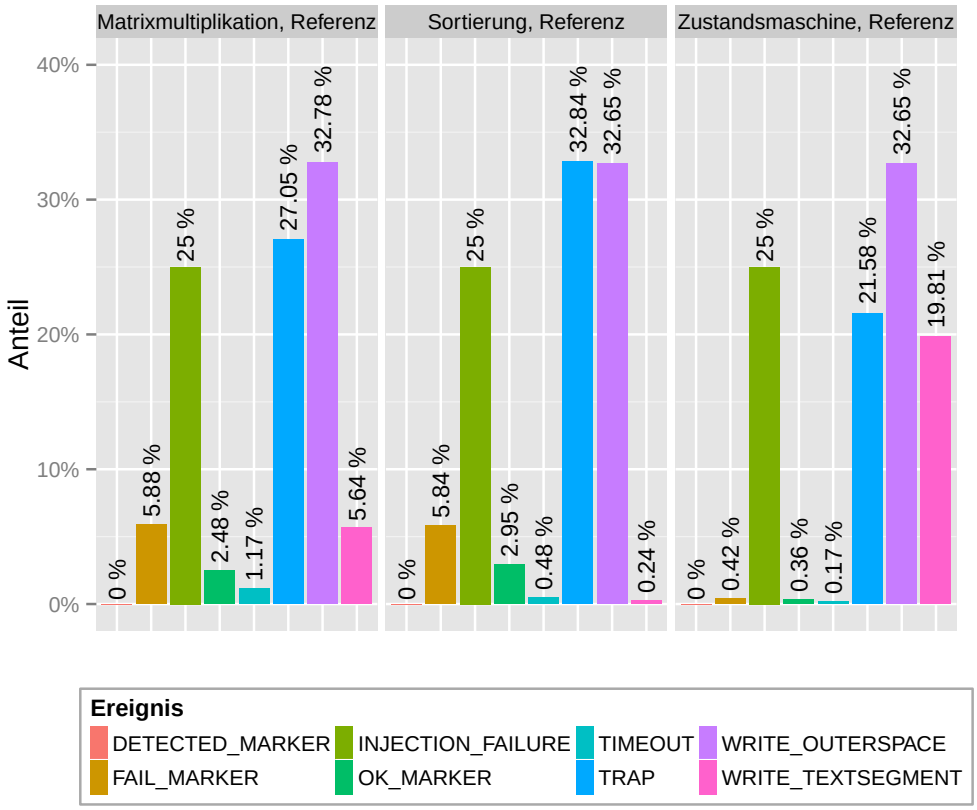


Abbildung 4.2 – Verteilung der Ereignistypen für die ungehärteten Anwendungen

als auch das Datensegment beinhaltet. Dies umfasst die unteren 24 Bit oder auch drei Byte einer 32-bittigen Adresse. Somit verlassen je ausgeführtem Befehl genau ein Viertel der am Befehlszähler injizierten Fehler den vorhandenen Adressraum und löst einen Injektionsfehler aus. Da dieser Wert somit ausschließlich von den Rahmenbedingungen und der Methodik des Experiments abhängt, wird er in der folgenden Betrachtung vernachlässigt.

4.3.1 Kontrollflussfehler

Im Folgenden werden zunächst die Ergebnisse der reinen Kontrollflussinjektion für die jeweiligen Anwendungen betrachtet. Hierzu wurden für jeden Ausführungsschritt am Befehlszähler jeweils alle Einbitfehler injiziert. Für die Auswertung ist hier besonders die Anzahl der `FAIL_MARKER`-Ereignisse, also die Existenz unentdeckter Fehler relevant. Ziel der Kontrollflussüberwachung muss sein, diesen Wert zu senken.

4.3.1.1 Matrixmultiplikation

In Abbildung 4.3 finden sich die Ergebnisse des Injektionsexperiments für die Matrixmultiplikation.

Der Ablauf der Anwendung gliedert sich dabei in drei Schritte. Im ersten Abschnitt werden zwei 5×5-Matrizen allokiert und zeilenweise mit den Werten zwischen 0 und 24 bzw. 42 und 66 befüllt, auch die Allokation der Ergebnismatrix findet hier statt. Im nächsten Schritt wird mittels des naiven Algorithmus die Matrixtrixmultiplikation durchgeführt. Abschließend erfolgt die Überprüfung des Rechenergebnisses durch elementweisen Vergleich. Die Injektion erstreckte sich lediglich auf die in Abschnitt 2 durchgeführten Operationen also die eigentliche Berechnung, die Härtung umfasste jedoch den gesamten Quelltext der Anwendung.

Betrachtet man die Anzahl der unentdeckten Fehler, so fällt auf, dass die Härtung mittels Plaininterblock sogar zu einer Zunahme der Fehlerzahl führt, d.h. die Vergrößerung des Fehlerraumes aufgrund der für das Verfahren zusätzlich eingefügten Befehle erzeugt mehr Fehlerereignisse, als durch die eingefügten Überprüfungen zusätzlich abgefangen werden. Dies ist durch eine Kombination unterschiedlicher Faktoren leicht erklärbar. Wie Tabelle 4.1 zu entnehmen ist, besitzt die Testanwendung nur eine geringe Basisblöckgröße von durchschnittlich $4,89$ JVM-Instruktionen pro Block. Gleichzeitig führt der Algorithmus eine Vielzahl von Feldzugriffen durch. Die JVM benötigt dabei für jeder Zugriff drei Instruktionen, die Übergabe des Arrays, die Benennung des Index und den eigentlichen Abruf des Wertes. Da bei der Übersetzung der Anwendung die üblichen, Java-eigenen Indexüberprüfungen deaktiviert wurden, lässt sich dieser Vorgang durch die umfassenden Addressierungsmodi der IA32-Architektur sehr kompakt abbilden, falls Index und Feldbasisadresse bereits in einem Register vorliegen. Dies ist für die Multiplikationsschleifen hier überwiegend der Fall, d.h. die tatsächliche Blockgröße ist nochmals geringer als der obige Wert erwarten lässt. Die Anwendung besitzt hier keine Methodenaufrufe, die im Plaininterblock-Verfahren für diesen Fall vorgesehenen Maßnahmen besitzen also keinen Einfluss auf das Ergebnis des Experiments. Weiterhin verwendet Plaininterblock zum Setzen des jeweiligen Signaturwertes eine direkte Zuweisung, Sprünge auf diese Instruktion sind prinzipbedingt nicht detektierbar. Alle anderen Verfahren vermeiden deshalb derartige Zuweisungen und nutzen stattdessen die statische Kontrollflussinformation um die Anpassung basierend auf dem alten Wert differentiell durchzuführen, die Autoren von YACCA benennen dies explizit als Entwurfentscheidung für die Wahl der dortigen *SET*-Operation [5]. Diese beiden Faktoren, die geringe Basisblockgröße, welche die von Plaininterblock erkannten Sprünge in die Mitte eines anderen Basisblocks hinein unwahrscheinlich macht, gepaart mit den schlechten Detektionseigenschaften der eingebrachten Überwachungsinstruktionen selbst, sorgen für die beobachtbare Zunahme.

Wie am Ende von Abschnitt 3.3.6.2 angedeutet weist das dominatorbasierte Verfahren strukturell eine gewisse Ähnlichkeit zu Plaininterblock auf, fängt jedoch die oben beschriebenen Problempunkte etwas ab: Statt explizite Zuweisungen zu verwenden, werden die Marker differentiell am Übergang der Dominanzgrenzen angepasst und durch die gröbere Granularität wird das Verhältnis zwischen den zur Überprüfung eingebrachten Befehlen und der eigentlichen Nutzarbeit verbessert, das Verfahren sorgt für eine Senkung der Anzahl unentdeckter Fehler. Dabei weisen die unterschiedlichen Konfigurationen des Verfahrens jedoch deutliche Unterschiede im Verhalten auf. Generell schneiden die kleinere Regionen

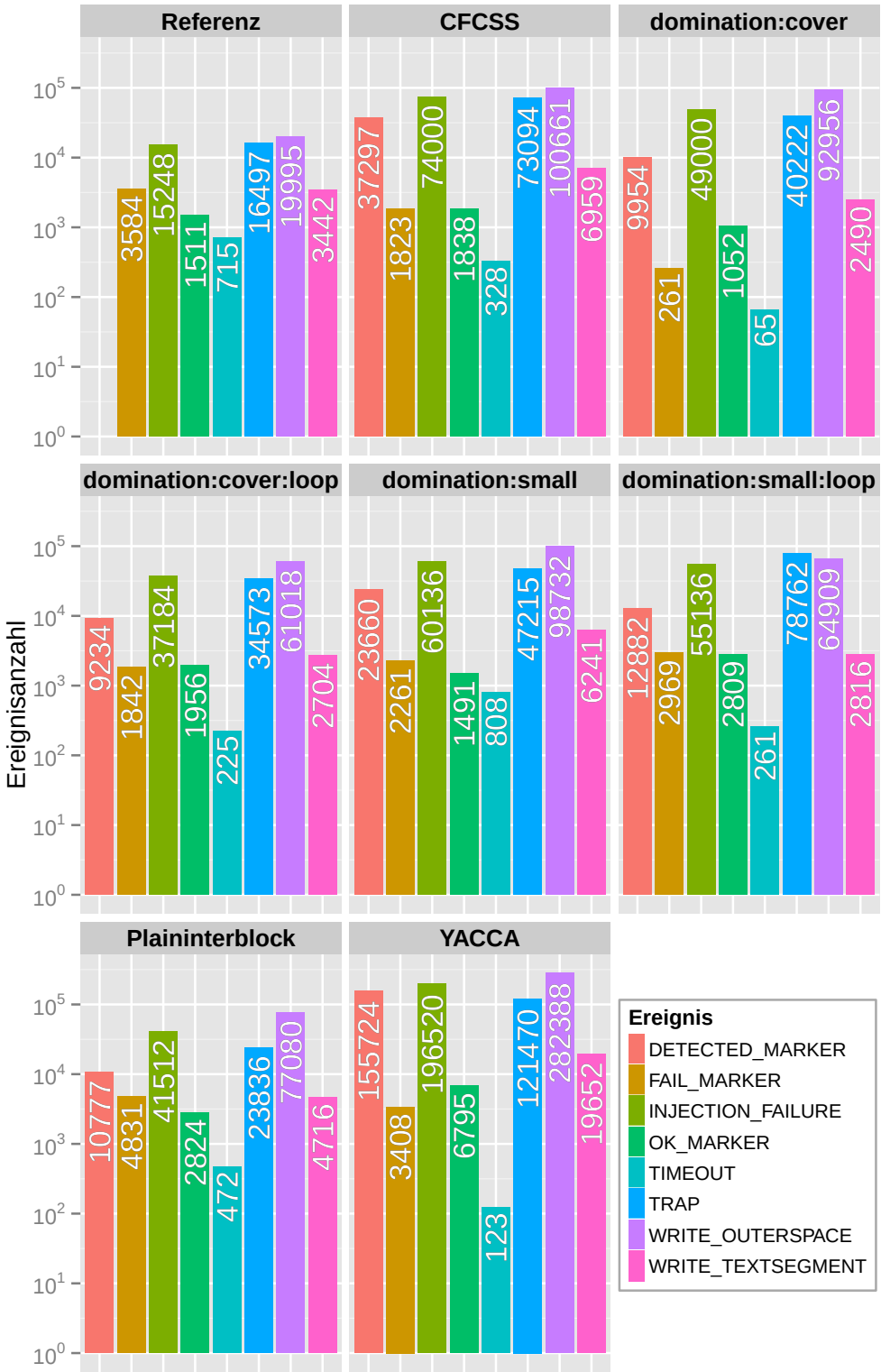


Abbildung 4.3 – Injektionsergebnisse für die Injektion des Befehlszählers bei der Matrixmultiplikation, Darstellung in logarithmischer Skala

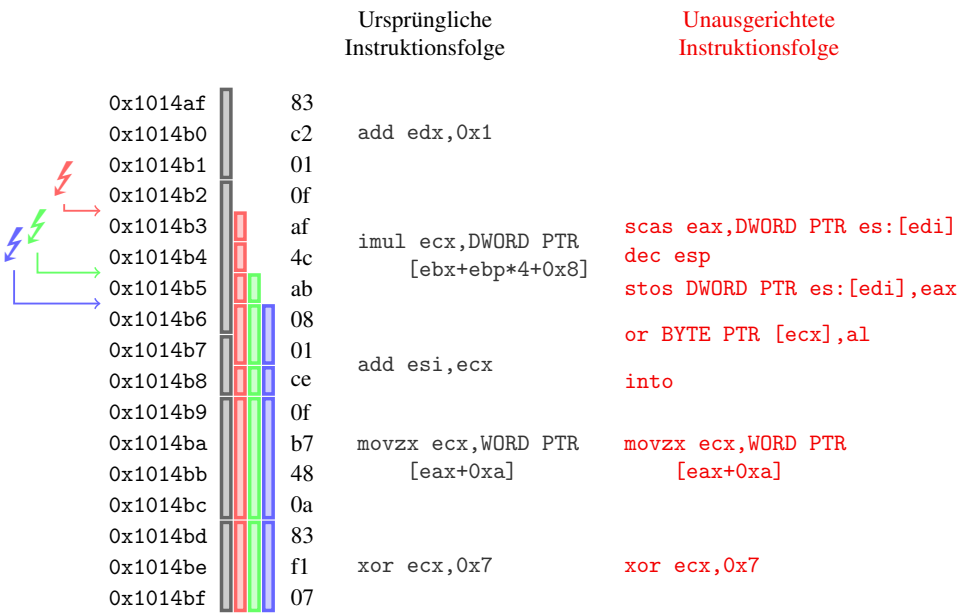


Abbildung 4.4 – Beispiel für drei unausgerichtete Zugriffe an den Adressen `0x1014b3`, `0x1014b5` und `0x1014b6` für die `dom:small` Programmvariante

bei der Auswahl bevorzugenden Verfahren (`:small`) hier schlechter ab als die auf eine gleichmäßige Verteilung (`:cover`) bedachten Gegenstücke. Der der Injektion unterworfene Programmabschnitt ist dabei mit 16 Basisblöcken eher überschaubar. `Domination:small` wählt auf diesem Bereich acht Regionen und vier zugehörige Dominanzgrenzblöcke aus, deutlich mehr als `domination:cover`, welches auf diesen Abschnitt lediglich drei Regionen und drei Grenzblöcke lokalisiert. Hier ist also nicht die Granularität der Überprüfung ausschlaggebend, auch könnte diese Änderung in der Granulartität keinesfalls die deutliche Verschlechterung durch Aktivierung der Schleifenerkennung schlüssig erklären. Vielmehr muss hier die Entstehungsweise der `FAIL_MARKER`-Ereignisse betrachtet werden. Auf der zur Gruppe der CISC Befehlssätzen gehörigen Intel Architektur 32 (IA32) werden wie bereits erwähnt zur Befehlskodierung Bytesequenzen variabler Länge eingesetzt. Wie Tabelle 4.4 zu entnehmen ist, haben alle Anwendungsvarianten eine vergleichbare mittlere Befehlslänge von etwas über drei Byte, somit sind die meisten Befehle länger als ein Byte. Dadurch werden bei der Injektion Sprünge in die Mitte einer Befehlssequenz, im Folgenden auch *unausgerichtete Sprünge* (engl. *unaligned jumps*) genannt, wahrscheinlich. Hier ist das Ergebnis dann von der auf die angesprungene Speicherstelle folgenden Bytesequenz abhängig. Lässt sich diese zu einer validen Instruktionsfolge dekodieren, so fängt sich die falsch ausgerichtete Abweichung üblicherweise nach einiger Zeit und kehrt zum normalen Programmfluss zurück, dort wird je nach Verfahren der Kontrollflussüberwachung dann entweder die Abweichung erkannt (`DETECTED_MARKER`), oder je nach Zustand der Programmdaten am Schluss die Gültigkeit (`OK_MARKER`) oder Ungültigkeit (`FAIL_MARKER`) der Berechnung festgestellt. Findet sich im

Instruktionsstrom hingegen eine nicht als korrekter Befehl dekodierbare Sequenz, so wird ein `TRAP`-Ereignis ausgelöst, ähnliches gilt für Schreibzugriffe auf die geschützten oder nicht vorhandenen Speicherbereiche innerhalb der falsch ausgerichteten Sequenz.

Betrachtet man vor diesem Hintergrund Anzahl und Anteil der unausgerichteten Sprünge für das `FAIL_MARKER`-Ereignis, so wird das vollständige Ausmaß dieses Effektes deutlich. Eine Analyse der internen Verteilung der unausgerichteten unentdeckten Fehler zeigt weiterhin, dass sich, auch bedingt durch die hohe Anzahl an Wiederholungen in den „heißen“ Schleifen der Multiplikation, die Fehler in der Regel in großer Anzahl um einige wenige Instruktionen gruppieren. Eine solche Programmstelle stellt beispielsweise die in Abbildung 4.4 dargestellte Umgebung der `imul`-Instruktion der Schleife für die Variante `dom:small` dar. So ergibt hier der vom Übersetzer nach dieser Instruktion erzeugte Bytestrom bei Sprüngen an das zweite, vierte und fünfte Byte dieses Operationskodes ein kurzes, gültiges Programmstück, welches sich ohne große Veränderungen am Programmzustand vorzunehmen wieder dem eigentlichen Programmfluss annähert. Dabei wurde jedoch mindestens ein Schleifendurchlauf fehlerhaft durchgeführt, das Berechnungsergebnis ist folglich nicht korrekt. Dieser Abschnitt allein ist für 620 der 2261 unentdeckten Fehler für diese Anwendungsvariante verantwortlich. Gleichzeitig wird hier auch die hohe Abhängigkeit vom internen Aufbau der Anwendung deutlich: Dass sich die Veränderung des Stackpointers mittels `dec esp` nur beschränkt auswirkt, ist hier der Tatsache geschuldet, dass im weiteren Programmablauf keine Funktionsaufrufe stattfinden und alle Feldadressen sich zu diesem Zeitpunkt in Registern befinden, somit nur selten auf nun falsch ausgerichtete Werte auf dem *Stapel* (engl. *stack*) zugegriffen wird, bzw. sich diese Zugriffe nicht stark auswirken.

Die Existenz solcher Abschnitte ist jedoch nur in sehr begrenztem Umfang vom betrachteten Verfahren abhängig, ist sie doch im Wesentlichen eine Eigenschaft des vom Übersetzer und Binder erzeugten Maschinenprogramms. Im vorliegenden Fall ist das für die Härtungsvariante `dominaton:cover` erzeugte Artefakt in dieser Hinsicht besonders günstig, wodurch sich die außergewöhnlich geringe Anzahl unentdeckter Fehler erklärt.

Für CFCSS und vor allem YACCA ist der Anteil der durch unausgerichtete Sprünge bedingten Fehler noch gering genug, um zumindest grundlegende Aussagen über die Wirksamkeit der Verfahren zuzulassen. Hierbei fällt besonders bei YACCA auf, dass die Fehlerzahl gemessen an der Explosion des Fehlerraumes noch vergleichsweise gering ist. Dies ist unter anderem auf die guten Eigenschaften der eingefügten Befehle zurückzuführen, so sind die für den Basisblockprolog generierten Assemblerinstruktionen bei YACCA mit einer maximalen Größe von ca. zwei Byte kurz. Wie Tabelle 4.4 zu entnehmen ist, scheint er eher günstige Eigenschaften bezüglich der Auswirkungen unausgerichteter Sprünge zu besitzen. Für ausgerichtete Sprünge scheint die in Abschnitt 3.3.5.2 beschriebene Robustheit zu greifen.

	∅ Opcodelänge (Byte)	Fehler unausgerichtet	Fehler gesamt	Anteil
Referenz	3,06	1908	3584	53,2 %
cfcss	3,38	1045	1823	57,3 %
domination:cover	3,30	185	261	70,9 %
domination:cover:loop	3,14	1424	1842	77,3 %
domination:small	3,23	1737	2261	76,8 %
domination:small:loop	3,14	2482	2969	83,6 %
plaininterblock	3,30	2640	4831	54,6 %
yacca	3,37	1064	3408	31,2 %

Tabelle 4.4 – mittlere Länge der Instruktionskodierung sowie Anteil und Anzahl der durch unausgerichtete Sprünge verursachten Fehler für die Matrixmultiplikation

4.3.1.2 Sortierung

Auch für die Feldsortierung wurde die beschriebene Fehlerinjektion am Befehlszähler durchgeführt, in Abbildung 4.5 findet sich eine Zusammenfassung der Ergebnisse des Experiments.

Zwar existiert hier, wie auch bei der Matrixmultiplikation, aufgrund des gewählten Sortieralgorithmus, einer Variante von Quicksort, eine „heiße“Schleife. Allerdings ist diese aufgrund algorithmischer Optimierungen an der Sortierfunktion deutlich umfangreicher als beim traditionellen Quicksort: So wird für Feldabschnitte kleiner sieben Elemente Insertionsort verwendet, für alle größeren Feldabschnitte wird Quicksort durchgeführt. Dabei wird jedoch das Pivotelement mittels eines Pseudomedian bestimmt, in Falle von mehr als 40 Elementen werden dazu neun, ansonsten drei, Feldeinträgen ausgewählt und dort der Median bestimmt. Mit diesem als Pivotelement wird dann die eigentliche Partition und sortierende Rekursion über dem Feld durchgeführt. Da das im Experiment verwendete Feld mit einer Größe von 42 Einträgen alle drei Fälle anspricht, wird eine starke Optimierung durch den Übersetzer verhindert, auch bleiben die rekursiven Aufrufe erhalten. Somit bildet diese Testanwendung einen wesentlich heterogeneren Kontrollfluss als die Matrixmultiplikation, welcher sich in einer höheren Repräsentativität der Ergebnisse widerspiegeln sollte, da Mikroeffekte in bestimmten Programmabschnitten besser abgefangen werden.

In Tabelle 4.5 findet sich wie auch schon bei der Betrachtung der Matrixmultiplikation eine Aufstellung der mittleren Länge der Befehlskodierung, wie auch Anzahl und Anteil der unausgerichteten Sprünge. Zwar ist hier die mittlere Opcodelänge leicht höher, allerdings scheint dieser Wertebereich von drei bis vier Byte für die mittlere Größe der Befehlskodierung typisch bei Anwendungen der IA32-Architektur, auch die Werte der Zustandsmaschine liegen in diesem Bereich. Allgemein ist der Anteil unausgerichteter Sprünge auch hier mit, CFCSS ausgenommen, Werten von 57,02 % bis 75,88 % hoch.

Beim Vergleich der Zahlen unentdeckter Fehler zeigt sich ein zur Matrixmultiplikation analoges Bild. Für Plaininterblock ist eine deutliche Zunahme der `FAIL_MAKER`-Ereignisse

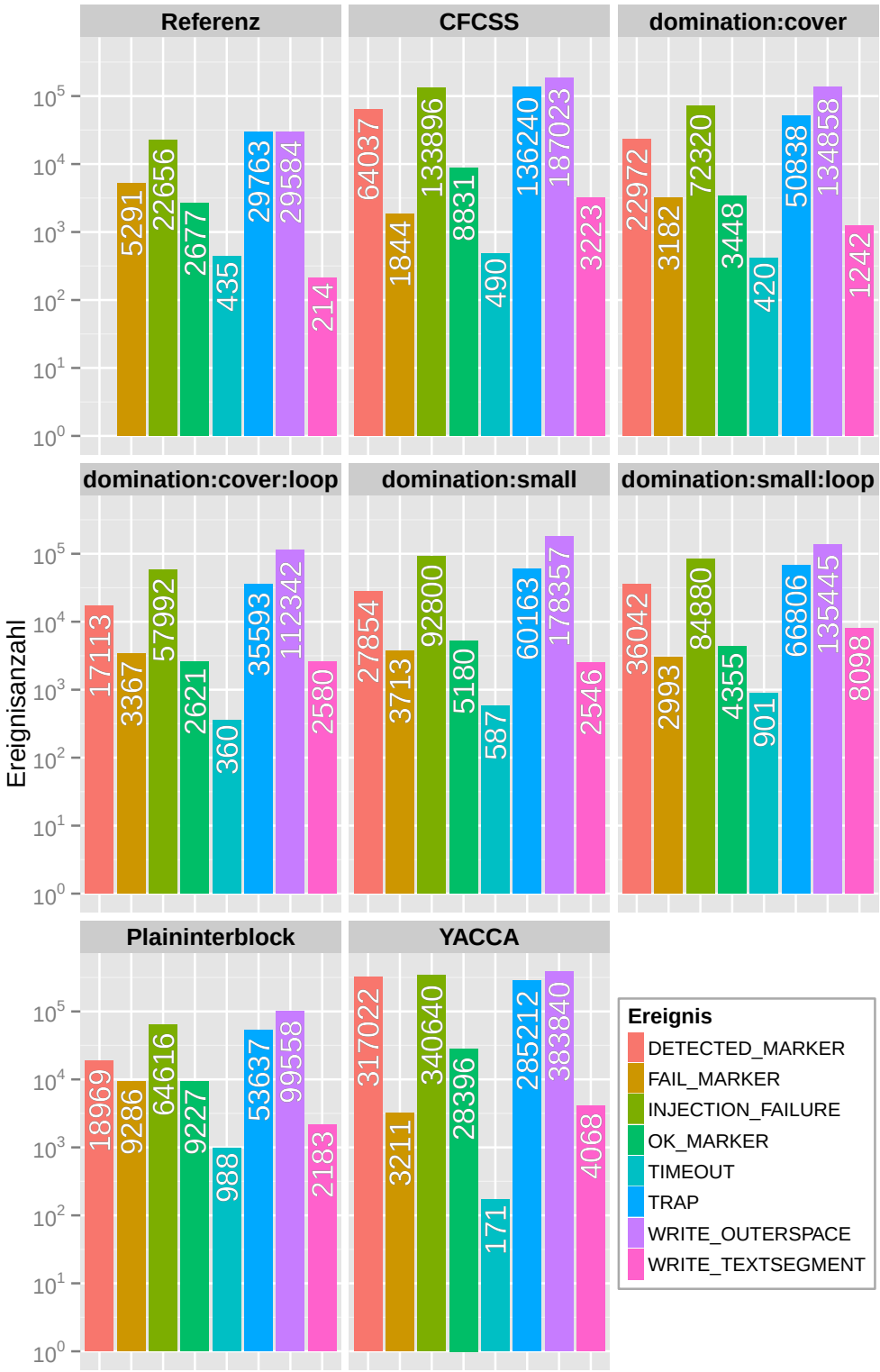


Abbildung 4.5 – Injektionsergebnisse für die Injektion des Befehlszählers bei der Sortieranwendung, Darstellung in logarithmischer Skala

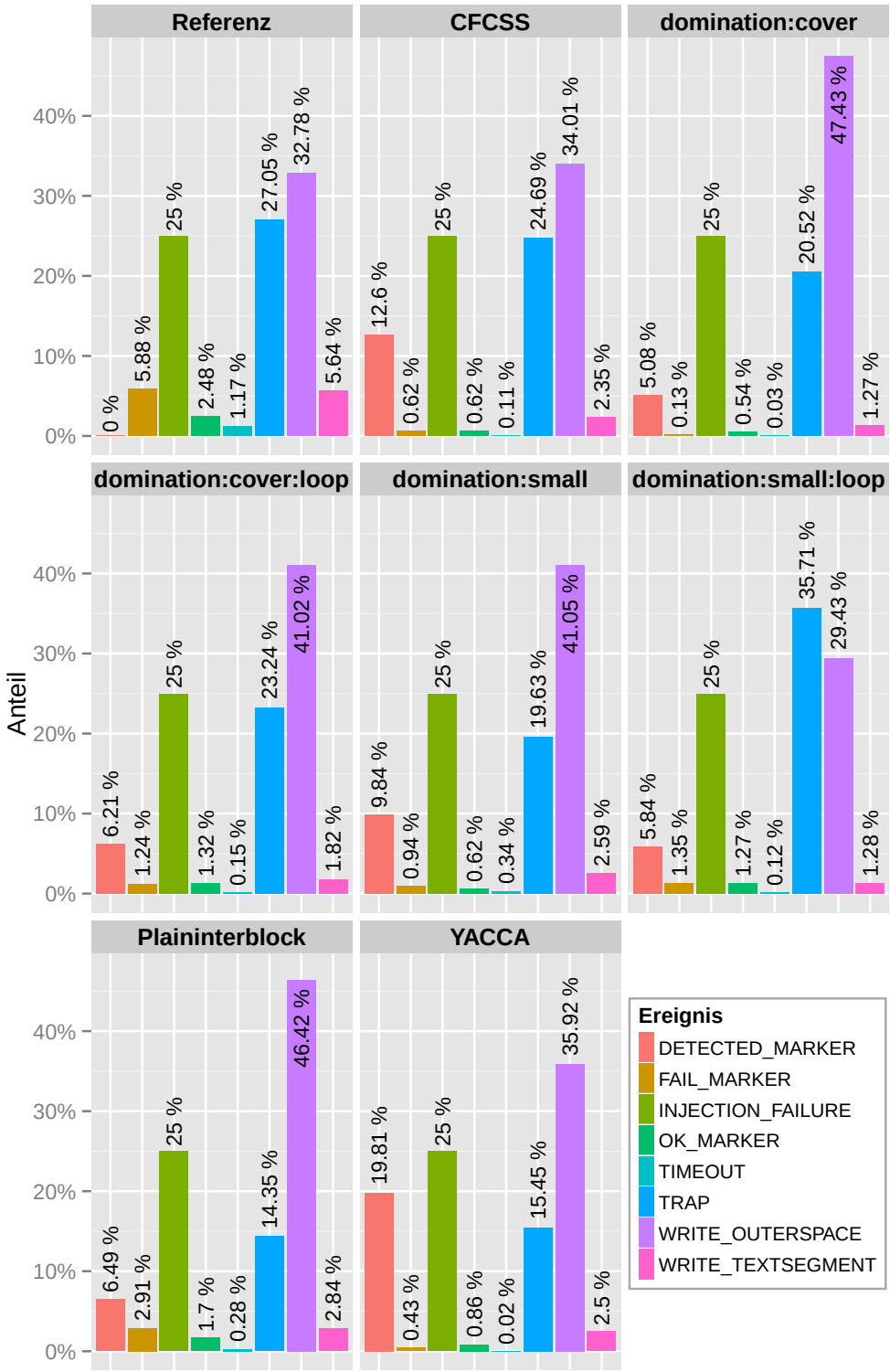


Abbildung 4.6 – Injektionsergebnisse für die Injektion des Befehlszählers bei der Sortieranwendung, Darstellung anteilig bezüglich der pro Variante erzeugten Kontrollflussfehler

	∅ Opcodelänge (Byte)	Fehler unausgerichtet	Fehler gesamt	Anteil
Referenz	3,15	3017	5291	57,02 %
cfcss	3,59	800	1844	43,38 %
domination:cover	3,49	2221	3182	69,80 %
domination:cover:loop	3,21	2154	3367	63,97 %
domination:small	3,36	2277	3713	61,33 %
domination:small:loop	3,21	1811	2993	60,51 %
plaininterblock	3,46	7046	9286	75,88 %
yacca	3,49	1862	3211	57,99 %

Tabelle 4.5 – mittlere Länge der Instruktionskodierung sowie Anteil und Anzahl der durch unausgerichtete Sprünge verursachten Fehler für die Sortieranwendung

zu beobachten, allerdings sind 75% dieser Fehler auf unausgerichtete Sprünge zurückführbar. Ansonsten bewegen sich die Verfahren auf einem ähnlichen Fehlerniveau, lediglich CFCSS stellt einen Ausreißer nach unten dar, welcher sich, wie `domination:cover` bei der Multiplikationsanwendung, durch die Abwesenheit von ungünstigen Bytesequenzen in der zugrundeliegenden Anwendung ergibt. Würde man hier die Zahl der übrigen Anwendungen von ca. 2000 durch unausgerichtete Sprünge verursachte Fehler annehmen, wäre auch die Gesamtzahl mit ca. 3000 Ereignissen im Bereich der übrigen Verfahren. Es fällt hier auf, dass das Auftreten dieser Sprünge in die Mitte eines Befehls, von den Grenzfällen Plaininterblock einerseits und CFCSS andererseits, über alle Härtungsmaßnahmen hinweg vergleichbare Zahlen liefert. Nimmt man für alle Verfahren eine ungefähr gleiche Wahrscheinlichkeit für das Auftreten einer solchen für unausgerichtete Sprünge ungünstigen Bytesequenz an, so wird diese, da alle Verfahren die gleiche Anzahl an Schleifendurchläufen beim Sortiervorgang durchführt, auch nährungsweise gleich oft angesprungen. Die schwankenden Laufzeiten sind hier nur bedingt relevant, da sich die Zahl der für den Einsprung günstigen Ausgangspunkte durch die Erhöhung der Laufzeit nicht automatisch mit erhöhen. Interessanterweise sinkt jedoch die Zahl der unausgerichteten Fehler für alle Verfahren außer Plaininterblock verglichen mit der Referenz. Dies erklärt sich durch all diejenigen Fehler, welche nach der Rückkehr des Kontrollfluss auf eine wieder korrekt ausgerichtete Instruktion aufgrund einer fehlerhaften Signatur erkannt werden. Bei allen Verfahren außer Plaininterblock ist diese Erkennung durch die Verwertung der Kontrollflussinformationen auch an Blockgrenzen möglich, für Plaininterblock lediglich dann, wenn dieser "Einfädelungspunkt" genau im Körper des Basisblockes liegt. Dies erklärt die im Bezug zu den anderen Verfahren unnatürlich hohe Zahl der bei Plaininterblock durch unausgerichtete Sprünge entstandenen Abweichungen, da die durch die Ausweitung des Fehlerraumes zusätzlich entstehenden, derartigen Fehlerereignisse nur mangelhaft detektiert werden. Gleichzeitig wird hier die hohen theoretische Detektionskapazität von YACCA auch in der Praxis deutlich, denn trotz der sehr großen Zunahme in

der Laufzeit und der damit verbundenen Zunahme der Dimension des Fehlerraumes werden durch die feingranulare Überwachung viele so entstandenen Fehler erkannt, dies erklärt auch den geringen Anteil der durch unausgerichtete Sprünge ausgelösten, unentdeckten Fehler, Ähnliches gilt für CFCSS.

Es ist hier zur Einordnung der Verbesserung allerdings wichtig, die Dimensionen des Fehlerraumes zu berücksichtigen. So reduziert das beste Verfahren, CFCSS, die Zahl der unentdeckten Fehler von den in der Referenz vorliegenden 5291 um 3447 auf 1844. Allerdings stellt dieser nun neu erkannte Anteil, relativ zur Gesamtzahl der in die ungehärtete Referenz injizierten Kontrollflussfehler lediglich einen Anteil von 3,8 % dar. Für die übrigen Verfahren, welche ca. 3000 `FAIL_Marker` Ereignissen nach der Härtung zeigen, reduziert sich diese Verbesserung nochmals auf lediglich 2,5 % zusätzlich erkannter Fehler anteilig an der Dimension des ungehärteten Fehlerraumes. Neben dieser moderaten Fehlerreduktion führen die Härtungsverfahren durch die Vergrößerung des Fehlerraumes also zu Zunahmen in den anderen, unproblematischen Ereignisarten der erkannten Fehler (`DETECTED_MARKER`), der Kontrollflussfehler ohne Einfluss auf das Berechnungsergebnis (`OK_MARKER`), der fälschlichen Schreibzugriffe auf das Textsegment (`WRITE_TEXTSEGMENT`) oder auf nicht vorhanden Speicher (`WRITE_OUTERSPACE`), sowie Hardwareausnahmezustände (`TRAP`) und Injektionsfehler (`INJECTION_FAILURE`). Durch diese Zuwächse kommt es also im Wesentlichen zu einer Änderung an der relativen Verteilung der Ereignisarten, wie sie in Abbildung 4.6 dargestellt ist. Dabei entspricht der Anteil des `FAIL_MARKER`-Ereignisses der in Abschnitt 4.2.3 erwähnten Fehlerrate. Hier wird die Nichtigkeit dieser Metrik sehr plastisch deutlich, zeigt sich hier doch das eigentlich aufgrund der theoretischen Einstufung der Verfahren erwartete Bild, mit einer moderaten Senkung der Fehlerrate von der Referenz zu Plaininterblock und von dort über CFCSS hin zu YACCA, welches mit einer Fehlerrate von 0,43 % eine Fehlerabdeckung von 99,57 % erreichen würde.

4.3.1.3 Zustandsmaschine

Unter den getesteten Anwendungen nimmt die Zustandsmaschine vor allem im Bezug auf ihre geringe Datenabhängigkeit eine Sonderstellung ein, auch findet sich hier keine Schleife, wodurch sich die verfälschende Fehlgewichtung von im Schleifenkörper auftretenden Ereignissen verhindern lässt. Allerdings geht damit auch eine Verringerung des Fehlerraumes einher.

Wie jedoch bereits in Abschnitt 4.3 betrachtet ist bei dieser Anwendung auch der Anteil der unentdeckten Kontrollflussfehler mit lediglich 0,42 % sehr gering. Das heißt, zumindest der für die ungehärtete Anwendung erzeugte Instruktionsstrom weist bereits eine gewisse Robustheit gegenüber Kontrollflussfehlern auf. Diese Kombination aus kleinem Fehlerraum und geringer Fehlerrate betont die Auswirkungen von Mikroeffekten.

Dementsprechend sind für eine valide Betrachtung bei diesem Experiment all diejenigen Fehler auszuschließen, welche ins Datensegment oder in den ungehärteten CiAO Anteil

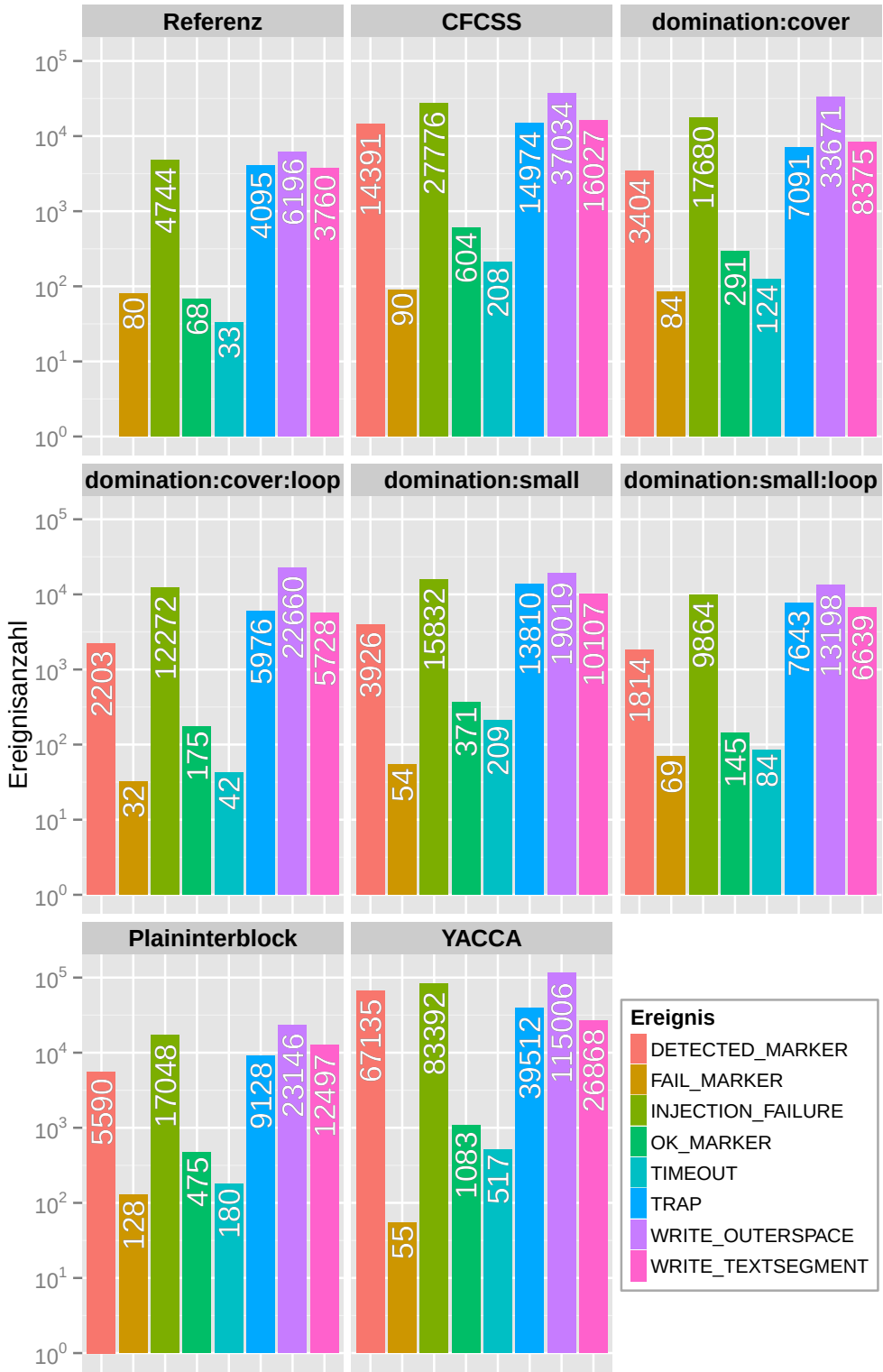


Abbildung 4.7 – Injektionsergebnisse für die Injektion des Befehlszählers bei der Zustandsmaschine, Darstellung in logarithmischer Skala

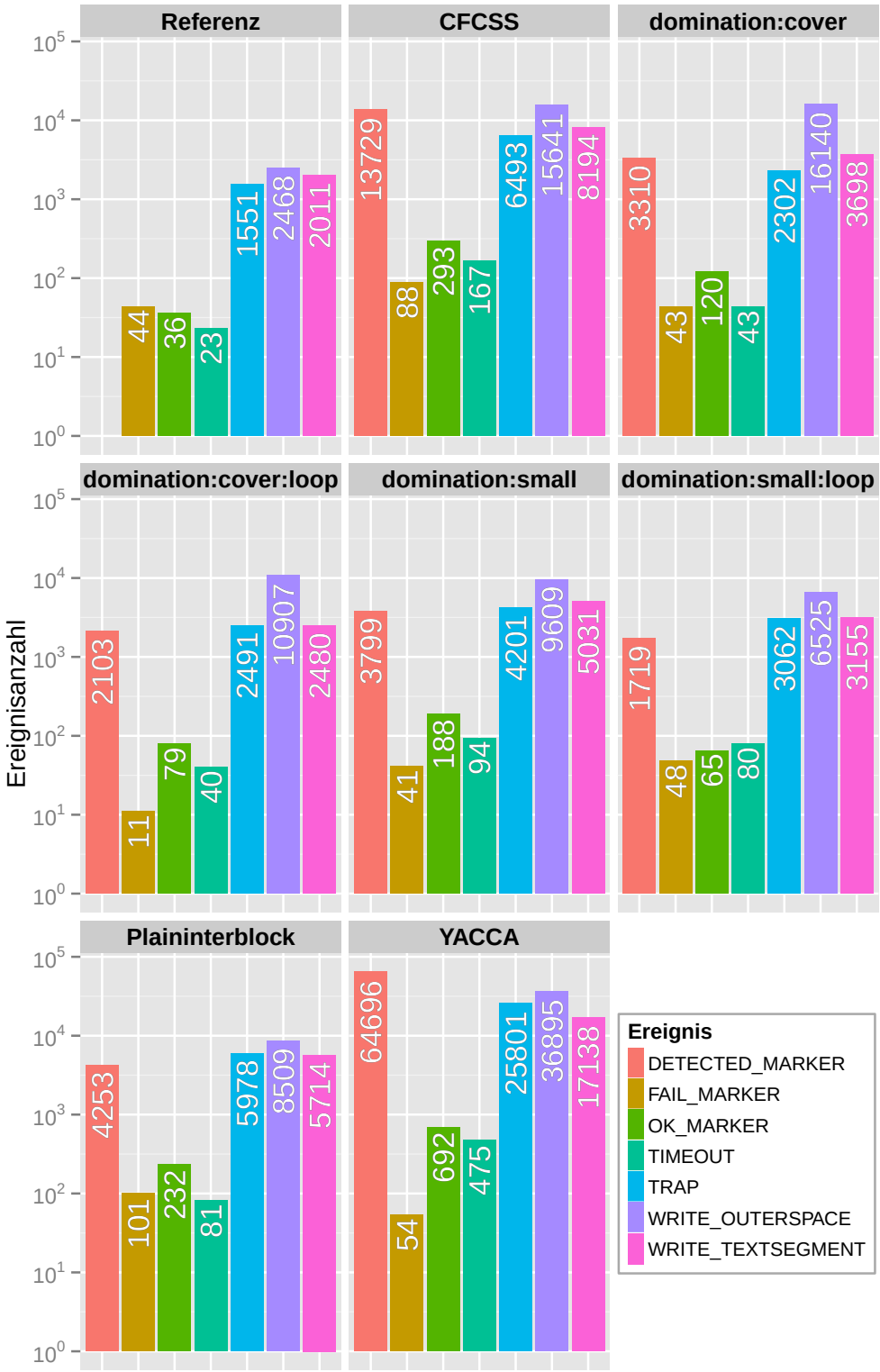


Abbildung 4.8 – Injektionsergebnisse für die Injektion des Befehlszählers bei der Zustandsmaschine beschränkt auf die reine Applikation, Darstellung in logarithmischer Skala

führen. In den anderen Anwendungen konnten diese aufgrund des großen Fehlerraumes und der langen Laufzeit anteilsmäßig noch vernachlässigt werden, für die Zustandsmaschine sind die Auswirkungen jedoch deutlich messbar. Die zugehörige um derartige Erscheinung bereinigte Aufstellung findet sich in Abbildung 4.8. Gleichzeitig ist diese Darstellung in der Auswertung mit besonderer Vorsicht zu behandeln, da sich durch diese Eingrenzung die Größen der zulässigen Injektionsgebiete unterscheiden. Dies wird insbesondere bei den Verfahren CFCSS und YACCA deutlich. Diese erzeugen im Laufe der Härtung die mit Abstand größten Binärobjekte bzw. Textsegmente, dementsprechend ist die Wahrscheinlichkeit, dass ein Sprung in diese Region zurückführt deutlich größer und somit die Menge der gefilterten Sprünge geringer. Qualitative Aussagen müssen in diesem Fall also mit den in der vollständigen Variante vorhandenen Werten abgeglichen werden.

Betrachtet man die Abnahme der `FAIL_MARKER` Ereignisse zwischen Abbildung 4.7 und Abbildung 4.8, so ist die deutlichste Abnahme für die Referenz zu betrachten, je mehr Platz die gehärtete Anwendung im Textsegment einnimmt, desto geringer wird die Wahrscheinlichkeit, dass ein Sprung aus dem Textsegment hinausführt, ohne gleich den gesamten eingeblendeten Speicher zu verlassen oder eine anderweitige, in Hardware behandelte Fehlergattung auszulösen und irgendwann mit einer akzeptierenden Berechnung des Wertes zu enden, also keinen `TIMEOUT` auszulösen, bei YACCA ist dies nur noch in einem einzigen Injektionspunkt der Fall. Dies heißt, die Verfahren sorgen in einem gewissen Umfang selbst für die benötigte Lokalität, um eine Härtung auf dem Niveau der Anwendung selbst ohne Berücksichtigung des zugrundeliegenden Betriebssystems durchführen zu können.

Generell ist die Anwendung für die dominatorbasierten Verfahren aufgrund des sehr flachen, aber dafür breiten Dominatorbaumes eher ungeeignet. Bei den `dom:small` Varianten wählt das Verfahren dabei 31 Regionen mit je zwei Basisblöcken aus, zusätzlich verbleibt eine Restgruppe, welche 177 Basisblöcke umfasst. Dabei haben beinahe alle dieser Regionen eine Dominanzgrenze in einem einzigen Basisblock, ein Umstand, welcher aufgrund der homogenen Programmstruktur und den sehr kurzen Methodenskörpern zu erwarten ist. Die übrigbleibende, große Region stellt für die Kontrollflussüberwachung einen Ungunstpunkt dar, da Fehler in dieser Region nicht entdeckt werden können. Da diese Region keine Dominanzgrenze besitzt, sich also über den gesamten Kontrollfluss von der Wurzel bis zum Programmende erstreckt, wird die Situation durch die Auslassen von Überprüfungsoperationen in diesem Bereich verschlechtert. Da der Bereich seit dem letzten Wechsel aus einer anderen Region zurück in diese Hauptregion bis zum Programmende auf Grund der spezifischen hier gewählten Umsetzung der `loop`-Optimierung so ungeschützt bleibt, steigt hier der Wert von `domination:small` nach `domination:small:loop` an.

Anders ist das Verhältnis bei `domination:cover`. Auch hier sind die Regionen inhomogen, neben elf Regionen einer sinnvollen Größe von zehn bist dreiundzwanzig Größen verbleiben 21 Regionen der Größe zwei, die Graphpartition war also nur bedingt erfolgreich. Da jedoch für diese Variante der Fehlerraum durch die mit den häufigeren Regionsdurchgängen verbundenen höheren Laufzeit um rund 7300 Injektionspunkte breiter ist als bei

`small`, ist die effektive Verbesserung von `domination:cover` in der auf den Anwendungsteil beschränkten Betrachtung nur um einen einzigen Fehler besser als die Referenz, in der uneingeschränkten Betrachtung tritt tatsächlich sogar eine Verschlechterung um vier unentdeckte Fehler ein. Selbstverständlich sind diese konkreten Änderungen nur sehr gering und somit nicht aussagekräftig, generell kann jedoch von einer der Referenz entsprechenden Fehlerzahl gesprochen werden. Interessant ist hier die geringe Fehlerzahl bei der `domination:cover:loop`-Variante. Anders als bei der Matrixmultiplikation oder der Sortieranwendung lassen sich die guten Werte hier jedoch nicht primär mit dem Ausbleiben unausgerichteten Sprüngen erklären. Vielmehr wird durch die häufigen Regionsübergänge eine ausreichende Markerüberprüfung trotz `loop`-Optimierung erreicht, gleichzeitig verringert die Optimierung die Zunahme in den Fehlerraumdimensionen. Das gute Abschneiden ist also im Wesentlichen das Ergebnis der hohen Ökonomie des Verfahrens, gepaart mit einer günstigen Graphpartition, welche häufige Regionsübergänge bedingt und so eine Erkennung trotz der nur spärlichen Überprüfung der Marker in den Regionen selbst ermöglicht.

Auffällig ist jedoch vor allem das schlechte Abschneiden von CFCSS, welches für die anderen beiden Anwendungen immer zu einer moderaten Verbesserung der Fehlerzahlen geführt hatte, hier jedoch sogar höhere Fehlerzahlen als die Referenz liefert. Die Begründung liegt hier in der in Abschnitt 3.3.4 beschriebenen Aliasierung. Da der State Machine Compiler die Zustände und Zustandsübergänge im Wesentlichen mittels Vererbung und Überschreibung derselben abstrakten Schnittstelle umsetzt, finden sich alle Implementierungen einer jeweiligen Methode in einer gemeinsamen Sammelgruppe, soweit der Übersetzer nicht durch statische Kontrollflussanalyse einige der Implementierungen ausschließen kann. Zwischen den Implementierungen dieser Gruppe kann in CFCSS nicht sinnvoll unterschieden werden, dadurch wird die Leistungsfähigkeit dieses Verfahrens im vorliegenden Fall stark eingeschränkt. Hier zeigen sich die Vorteile der in YACCA gewählten Umsetzung, welche diese Problematik prinzipbedingt umgeht: Trotz des deutlich größeren Fehlerraumes ist die Anzahl unentdeckter Fehler hier im Bezug zur Referenz.

4.3.1.4 Zusammenfassung

Über alle Kontrollflussfehlerinjektionsexperimente hinweg sind stets ähnliche Ergebnisse zu beobachten. Generell sinkt für die gehärteten Verfahren, mit Ausnahme von Plaininterblock, die Anzahl der unentdeckten Fehler leicht. Für die Zustandsmaschine ist diese Feststellung auch für CFCSS und die Abdeckungsvariante des dominatorbasierten Ansatzes (`dom:cover`) unzutreffend, hier wird die mangelnde Eignung der Verfahren für die jeweilige Anwendung deutlich. Allerdings ist allgemein eine höhere Wirksamkeit der Verfahren für die datenabhängigen Verfahren zu beobachten: Betrachtet man die mittlere Verbesserung über alle Verfahren jeweils im Bezug zur Fehlerzahl in der Referenz, so wurde im Mittel für die Matrixmultiplikation eine Verbesserung von 30,66 % erreicht, für die Sortieranwendung von 25,49 % und für die Zustandmaschine wurde die Fehlerzahl im Mittel lediglich um 8,57 % gesenkt. Dies

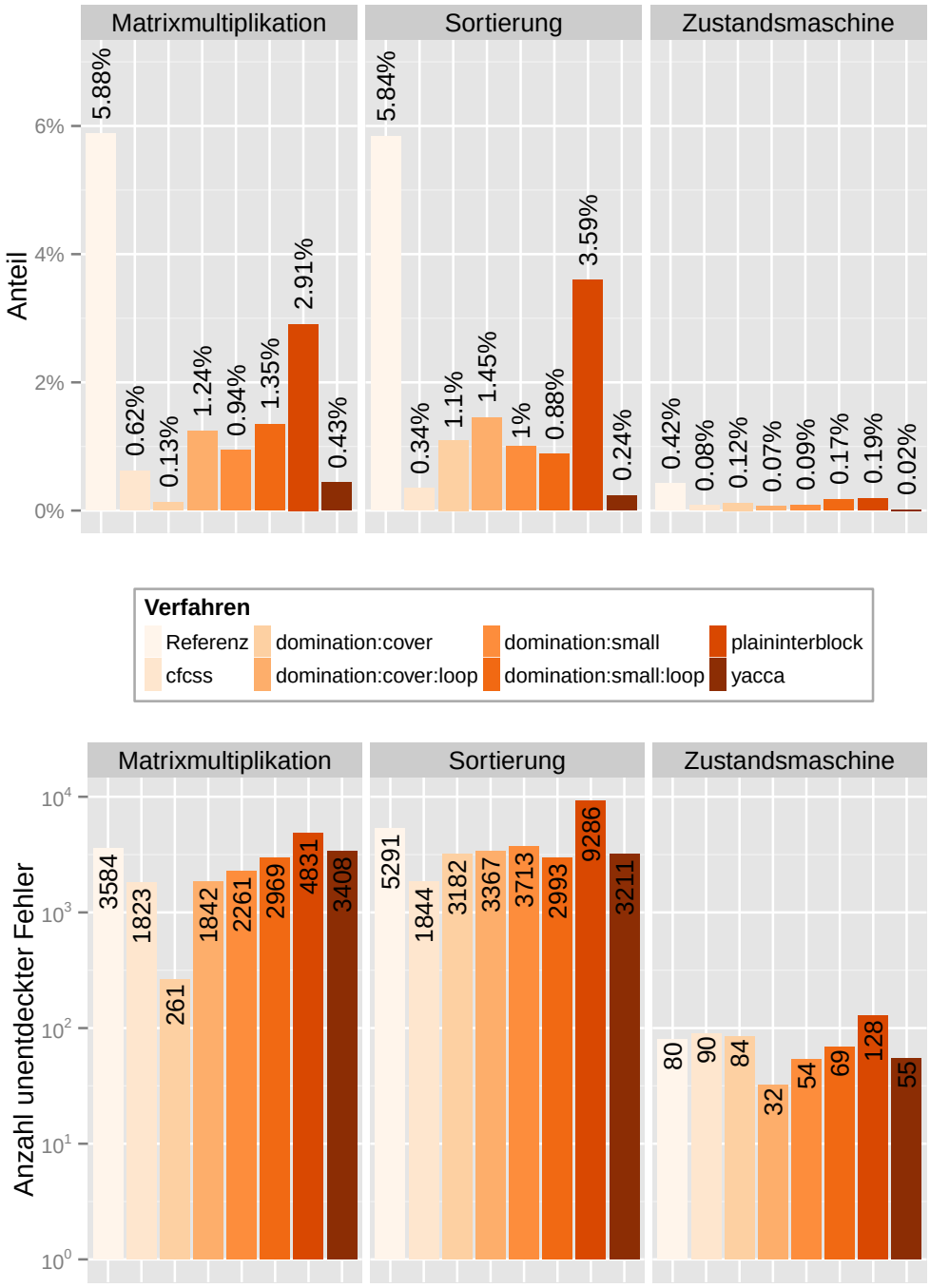


Abbildung 4.9 – Anzahl der `FAIL_MARKER`-Ereignisse nach Verfahren und Anwendung, anteilig zur Menge der injizierten Fehler (oben) und absolut (unten)

deckt sich mit der Beobachtung, dass in letzterer Anwendung bereits eine deutlich geringere Neigung zu unentdeckten Fehlern bereits in der ungehärteten Variante von lediglich 0,42 % zu beobachten war. Die Verfahren hatten also nur noch eine sehr geringe Fehlerzahl als Ausgangsgrundlage, auf welcher die Verbesserungen vorgenommen werden konnten. Der Nutzen und das zu bevorzugende Verfahren beim Einsatz von Kontrollflussüberwachung sind also im Wesentlichen anwendungsabhängig und lassen sich nicht abschließend beurteilen.

Davon abgesehen ist über alle betrachteten Anwendungen hinweg eine große Abhängigkeit von Mikroeffekten in den verfahrensspezifischen Einzelmessungen zu beobachten. Diese begründen sich vor allem in der Struktur des dem jeweiligen Experimentlauf zugrundeliegenden Maschinenprogrammes und seiner Ausrichtung im Speicher, bedingt durch die Effekte der unausgerichteten Sprünge, also Sprünge in die Mitte einer Instruktionskodierung, welche in allen Experimenten zu beobachten sind. Dieser Effekt ist dabei stets für weit über 50 % der betrachteten Fehler verantwortlich, üblich sind Werte in der Größenordung von 70 % bis 80 %. Die Abwesenheit solcher Mikroeffekte ist dann auch jeweils für das gute Abschneiden der jeweils besten Verfahren je Anwendung, also `dom:cover` bei der Matrixmultiplikation, CFCSS bei der Sortierung und in geringerem Umfang auch für das Abschneiden von `dom:small` bei der Zustandsmaschine, verantwortlich.

Auffällig ist hier auch die nur geringe bis nicht vorhandene Verbesserung des YACCA-Verfahrens im Vergleich zum Vorgänger CFCSS, auf welchem das Verfahren eigentlich basiert. Lediglich für ausgesprochen stark aliasierende Implementierungen wie der Zustandmaschine wird eine leichte Verbesserung sichtbar. Dieser Missstand ist größtenteils der Verwendung der in Abschnitt 4.2.3 widerlegten Metrik der Fehlerabdeckung geschuldet. In Abbildung 4.9 findet sich eine kontrastierende Gegenüberstellung dieser Metrik zur hier verwendeten, absoluten Fehlerzahl einer vollständigen Abtastung.

Aufgrund dieser Unzulänglichkeiten dieser Metrik ist ein Vergleich der hier betrachteten Messwerte mit experimentellen Ergebnissen in der Literatur schwierig: Ein Vergleich der Absolutzahlen ist mangels Referenz nicht möglich und ein Vergleich der Fehlerraten ist auch abseits der Fragwürdigkeit der Eignung dieser Metrik auch aufgrund von Unterschieden in der Vorgehensweise der Messungen ohne praktische Aussagekraft. So wurden in den durch Oh et al. [4] und Goloubeva et al. [5] durchgeführten Experimenten jeweils nur stichprobenartige Fehlerinjektionen an zufällig auswählten Sprunginstruktionen im Programm vorgenommen, dabei ist keine Berücksichtung der Äquivalenzintervalle für die Auswahl der Injektionsstellen vermerkt. Weiterhin wurden in beiden Fällen RISC-basierte Architekturen, MIPS im Falle des ersten Papiers und eine SPARC V8 im zweiten Papier genutzt.

Allerdings lässt sich mittels der sich aus den Messungen ergebenden Fehlerraten die schrittweise Fortentwicklung der Verfahren qualitativ nachvollziehen. Zwar verschlechtert Plaininterblock in allen Fällen das Verhalten der Anwendung bezüglich Kontrollflussfehlern, dennoch sinkt durch die Aufblähung des Fehlerraumes die Fehlerrate ab. Auch CFCSS verbreitert durch die zusätzlichen Instruktionen nochmals die Dimensionen des Fehlerraumes deutlich, erreicht aber durch bessere Überprüfungseigenschaften in den meisten Fällen eine

tatsächliche Reduktion der unerkannten Fehler. Somit ist die betrachtete Fehlerrate nochmals deutlich geringer als bei Plaininterblock, das gleiche gilt für das Verhältnis zwischen CFCSS und YACCA.

Abschließend lässt sich festhalten, dass die Verfahren zwar tatsächlich geringe Verbesserungen bewirken. Die Unterschiede zwischen den auf dem Kontrollfluss aufbauenden, explizite Zuweisungen vermeidenden Verfahren, also dem dominatorbasierten Verfahren, CFCSS und YACCA fallen jedoch gering aus und sind stark anwendungs- und maschinenprogrammabhängig.

4.3.2 Allgemeine Einbitfehler

Bisher wurden in Abschnitt 4.3.1 zunächst selektiv die bloßen Kontrollflussfehler betrachtet, welche sich durch die Injektion des Befehlszählers ergeben. Allerdings blendet diese Darstellung hierbei Fehler an Daten in Speicher und Registern bewusst aus. Geht man jedoch von einem uniform verteilten Fehler auf dem gesamten Programmzustand aus, so ist keine getrennte Betrachtung zulässig, besonders da die Anwendung der Härtungsverfahren die Dimensionen des gesamten Fehlerraums vergrößert, also auch den der Datenfehler.

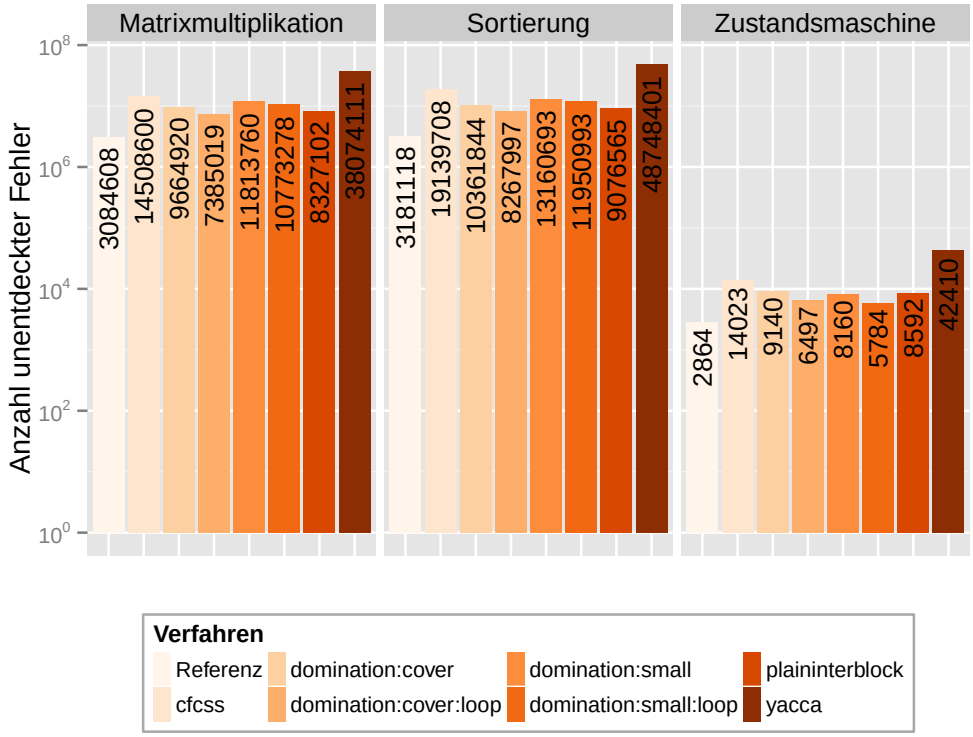


Abbildung 4.10 – Anzahl der `FAIL_MARKER`-Ereignisse bei der Injektion aller Einbitfehler in Programm und Daten

In Abbildung 4.10 finden sich die Injektionsergebnisse für ein Fehlerinjektionsexperiment über dem gesamten Fehlerraum der Einbitfehler. Dabei wurden alle Registerwerte inklusive dem oben betrachten Befehlszähler, alle gelesenen Daten im Speicher als auch die gelesenen *Instruktionskodierungen* (engl. *Opcodes*) inklusive Operanden injiziert und eine Gewichtung mittels der Äquivalenzintervalle, wie in Abschnitt 4.2.3 erläutert, vorgenommen. Diese Messung kann im Sinne der an der Programmierschnittstelle des Prozessors auftretenden Fehler als vollständig angenommen werden.

In der Messung sind mehrere Fakten augenfällig. Zunächst sind die unter Abschnitt 4.3.1 betrachteten, durch die jeweils besten Verfahren erzeugten Verbesserungen von 3323 bei der Matrixmultiplikation, 3447 für die Sortierung und 48 im Falle der Zustandsmaschine gemessen an der Gesamtzahl der `FAIL_MARKER`-Ereignisse für allgemeine Einbitfehler nur von geringer Signifikanz. Dementsprechend wird die Betrachtung im Wesentlichen durch die Menge der Datenfehler dominiert. Deren Anzahl ist dabei eng mit der Programmlaufzeit verknüpft: Für jede zusätzliche Instruktion, welche sich zwischen einer Schreiboperation und der zugehörigen Leseoperation befindet steigt die Wahrscheinlichkeit eines Fehlers, im Experiment äußert sich dies durch ein längeres Äquivalenzintervall, also auch einer höheren Fehleranzahl nach der Gewichtung. Da in allen Fällen wenigstens eine Verdopplung der Programmlaufzeit durch die Härtungsverfahren festzustellen ist, liegt für alle betrachteten Verfahren auch die Absolutzahl der unentdeckten Fehler über dem Wert der Referenz. Allgemein ist die starke Korrelation zwischen den hier verzeichneten Fehlerzahlen mit den in Abschnitt 4.1.2 gemessenen Laufzeiten beobachtbar, die Ordnung der Verfahren bezüglich Laufzeit stimmt im Wesentlichen mit der sich aus der Fehlerzahl ergebenden Sortierung überein.

Aus diesen Ergebnissen lässt sich die zwingende Notwendigkeit einer die Kontrollflussüberwachung begleitenden Datenhärtung ableiten. Vor dem Hintergrund der eher geringen Menge der Kontrollflussfehler in Relation zu der Anzahl der unentdeckten Datenfehler ist die Wirksamkeit dieser Kombination über der einer reinen Datenhärtung hinaus jedoch anzuzweifeln.

4.4 Diskussion

Wie in Abschnitt 4.2.3 dargelegt, ist das in der Literatur bisher übliche Vergleichsverfahren der Fehlerabdeckung nicht nur unzureichend, sondern sogar verzerrend, indem es Werte unterschiedlich dimensionierter Fehlerräume ohne Gewichtung vergleicht. Die in dieser Arbeit genutzte Vergleichsmetrik der absoluten Fehlerzahlen einer vollständigen Abtastung des Fehlerraums korrigiert diese Effekte. Die Ergebnisse der in Abschnitt 4.3 durchgeführten Fehlerinjektion relativieren hier die in der Literatur durch die Nutzung der fehlerhaften Metrik entstandenen Wirksamkeitsaussagen deutlich. Zwar zeigt sich für das Verhältnis zwischen der Anzahl unentdeckter Fehler in den gehärteten Varianten zur Menge der unentdeckten Fehler in der ungehärteten Referenz eine Verbesserung von im Mittel 18,88 %, diese Verbesserung

stellt im Fehlerraum aller in der Referenz möglichen Kontrollflussfehler dennoch nur eine mittlere Verbesserung von 0,9 % dar.

Wird die Beschränkung der Betrachtung auf Kontrollflussfehler gänzlich gelöst und wie in Abschnitt 4.3.2 die Auswirkung aller Einbitfehler, also auch die von Datenfehlern berücksichtigt, schwindet dieser Anteil nochmals. Es wird offensichtlich, dass zumindest für die hier betrachteten Anwendungen auf der IA32 bei allen Verfahren durch die Ausweitung des Fehlerraumes aufgrund der zusätzlichen Instruktionen die Anzahl der unentdeckten Fehler nach der Härtung deutlich ansteigt. Ob vor diesem Hintergrund die Kombination aus Maßnahmen sowohl zur Daten- als auch zur Kontrollflusshärtung einen tatsächlichen Vorteil gegenüber der reinen Datenhärtung bietet bleibt fraglich.

Mit Ausnahme von Plaininterblock, welches durchweg zu einer Steigerung der Fehlerzahl führte, sind die Unterschiede zwischen den betrachteten Verfahren über alle Anwendungen hinweg weder signifikant noch die Eignung für spezifische Anwendungstypen feststellbar. Lediglich für CFCSS ist die Aliasierungsproblematik für stark auf Vererbung basierenden Anwendungen wie der Zustandsmaschine beobachtbar, das Verfahren in solche Fällen also eher ungeeignet. Vor diesem Hintergrund empfiehlt sich in praktischen Anwendungen eine Umsetzung der dominatorbasierten Verfahren, da diese bei gleichbleibender Detektionsfähigkeit die bessere Laufzeiteffizienz besitzen.

In den Messungen wurde weiterin eine hohe Abhängigkeit der experimentell bestimmten Fehleranfälligkeit der gehärteten Verfahren nicht nur vom gewählten Fehlermodell, sondern auch von der Zielarchitektur und dem erzeugten Maschinenprogramm, welches nur unwesentlich durch die Wahl des Verfahrens bzw. des durch JINO erzeugten C(++)-Quelltextes und viel mehr durch die spezifischen Eigenschaften des verwendeten C(++)-Übersetzers bestimmt wird, deutlich. Für die hier betrachtete IA32 äußert sich diese Abhängigkeit wie beschrieben vor allem in der Existenz von Ungunststellen bezüglich unausgerichteter Sprünge. Dieser Zusammenhang macht eine Neubewertung der Verfahren nach jeglicher Änderung an einer beliebigen der am Kompilationsprozess beteiligten Komponenten, also der Anwendung, des JINO-Übersetzers, des im Hintergrund verwendeten C(++)-Übersetzers, des zugrundeliegenden OSEK Systems oder auch nur der Konfigurationsoptionen all dieser Programme, nötig.

Besonders vor dem Hintergrund der in Abschnitt 4.1.2 ermittelten hohen Speicher- aber vor allem auch Laufzeitkosten von je nach Verfahren 120 % bis 2000 % stellt sich hier die Frage des Kosten/Nutzenverhältnisses der Umsetzung, zumal bei Laufzeitkosten von über 100 % eine komplette Replikation der Anwendung zumindest bei einer idealisierten Betrachtung sogar geringere Kosten erzeugen würde.

Zumindest für die in dieser Arbeit getroffene Umsetzung ist die Praktikabilität und die Effektivität einer zusätzlichen Kontrollflussüberwachung somit anzuzweifeln. Allerdings ist für diese Einstufung maßgeblich der große Laufzeitaufwand verantwortlich, der einerseits aufgrund der damit verbundenen Leistungseinbußen die Verfahren unattraktiv macht und andererseits durch die somit ausgelöste Vergrößerung des Fehlerraumes der Kontroll-

und Datenfehler zu einer Verschlechterung des Verhaltens der Anwendung unter der Fehlerinjektion führt. Dieser Effekt ist maßgeblich durch die Realisierung der Signaturen als `volatile`-Variablen verursacht, die Ergebnisse also zunächst nur auf vergleichbare Umsetzungen übertragbar. Eine Realisierung der Signatur durch die Reservierung eines dedizierten Registers könnte hier Milderung verschaffen. Dennoch scheint der Nutzen vor dem geringen Anteil der Kontrollflussfehler an der Gesamtfehlerzahl auch hier immer noch fraglich.

4.5 Resümee

Generell zeigen die Messungen also leichte Verbesserungen in der Anzahl unentdeckter Fehler durch die Härtung für alle Verfahren außer Plaininterblock, welches über alle Anwendungen hinweg die Fehleranfälligkeit der Anwendung erhöht. Die übrigen Verfahren zeigen vergleichbare, leichte Verbesserungen bezüglich der Zahl unentdeckter Fehler nach der Härtung, beschränkt auf die Betrachtung von reinen Kontrollflussfehlern. Für stark auf Polymorphie basierende Umsetzungen zeigt CFCSS jedoch bedingt durch die Aliasierungsproblematik Schwächen. Die Messungen werden jedoch mehrheitlich von Mikroeffekten auf der Ebene der erzeugten Maschinenprogramme dominiert.

In Anbetracht der sehr hohen Laufzeitkosten und der damit verbundenen Aufweitung des Fehlerraumes auch für Datenfehlern, welcher bei kombinierter Betrachtung von Kontrollfluss und Datenfehlern zu einer deutlichen Zunahme der Fehlerzahlen führt, scheint ein Einsatz im KESO-Umfeld eher unattraktiv.

Kapitel 5

Fazit

Nachdem die Nichtigkeit der bisher verbreiteten Metrik der *Fehlerabdeckung* (engl. *fault coverage*) in Kapitel Abschnitt 4.2.3 nachgewiesen worden ist, wurde in dieser Arbeit durch die absolute Anzahl der unentdeckten Fehler bei einer vollständigen Abtastung des Fehlerraumes eine neuartige Vergleichsmetrik für Kontrollflussüberwachungsverfahren eingeführt.

Die in Kapitel 4 vor diesem Hintergrund durchgeführten Messungen finden hier auf die eingangs gestellte Frage nach der Effektivität und dem Kosten-Nutzenverhältnis für einen Kontrollflussüberwachungsdienst im Umfeld des KESO-Projektes eine doch recht deutliche Antwort. Zwar zeigt sich eine generelle Wirksamkeit der betrachteten Verfahren, allerdings fallen die Verbesserungen eher mäßig aus. Im Falle des Plain Inter-Block Error Detection Verfahrens ist sogar eine Verschlechterung der Fehlertoleranzeigenschaften zu beobachten, die Umsetzung des ECCA-Verfahren war aus strukturellen Gründen nicht möglich. Für die übrigen Verfahren, CFCSS, YACCA und den dominatorbasierten Ansatz sind die Unterschiede bezüglich der jeweiligen Menge der unentdeckten Kontrollflussfehler nach Härtung im hier durchgeführten Fehlerinjektionsexperiment insignifikant. Gleichzeitig führen die durch die Härtung zusätzlich in das Programm eingebrachten Instruktionen zu einer deutlichen Zunahme der Datenfehler, da hier die Größe des Fehlerraums mit der Programmlaufzeit steigt. Dieser die Verbesserungen für die Kontrollflussfehler bei weitem überwiegende Effekt, der durch die hohen Laufzeitkosten von 120 % bis 2000 % je nach Verfahren nochmals deutlich verstärkt wird, macht den Produktiveinsatz der Kontrollflusshärtung unattraktiv bis widersinnig. Da diese Wirkung jedoch bei den leichtgewichtigeren Verfahren wie dem dominatorbasierten Ansatz aufgrund der kürzeren Laufzeit deutlich geringer ausgeprägt sind als bei CFCSS oder dem nochmals deutlich teureren YACCA-Verfahren, wäre im Falle einer Umsetzung dieses Verfahren vorzuziehen.

Gleichzeitig ist eine generell hohe Abhängigkeit zwischen dem konkret erzeugten Maschinenprogramm und dem Verhalten der Anwendung unter der Fehlerinjektion zu betrachten. Somit ist die Wirksamkeit von allen Komponenten des Gesamtsystems, also der Anwendung, dem gewählten Verfahren, der in KESO vorhanden Übersetzungskomponente, dem C(++)-

Übersetzer, dem Binder, dem zugrundeliegenden OSEK-Betriebssystem und der Prozessorarchitektur abhängig, Aussagen sind immer nur für die durch diese spezifische Kombination erzeugten Kompilate gültig und müssen bei jedweder Änderung neu evaluiert werden.

Abseits von der Frage der Praxistauglichkeit der Verfahren im Einzelnen zeigen die Messungen auch die begrenzte Aussagekraft der bis dato vorwiegend zu Evaluationszwecken verwendeten *Minimalanwendungen* (engl. *Mircobenchmarks*) aufgrund der sich dort durch die kurzen Laufzeiten stark verstärkenden Mikroeffekte. Gleichzeitig scheinen zumindest die Testanwendungen auf IA32 im Allgemeinen eine doch recht hohe Resistenz gegenüber Kontrollflussfehlern zu besitzen. Dementsprechend ist eine erneute Fehlerinjektion für umfangreichere Anwendungen, auch auf von IA32 deutlich abweichenden Architekturen, bspw. RISC-Systemen anzudenken. Weiterhin ist sowohl für die hohen Laufzeitkosten als auch für die zusätzliche Aufblähung des Fehlerraumes die gewählten Umsetzung der Signaturwerte als `volatile` Variablen maßgeblich. Abweichende Realisierungen der Laufzeitsignatur, wie beispielsweise in Form eines dedizierten Register, welches die teueren Lade und Schreiboperationen abfängt, sollten in Betracht gezogen und evaluiert werden. Auch ist ein Vergleich zwischen einer reinen Datenhärtung und einer Kombination aus Daten- und Kontrollflusshärtung lohnend, gegebenenfalls ergibt sich hier ein tatsächlicher Zugewinn in den Detektionskapazitäten, welcher in der reinen Kontrollflusshärtung aufgrund der stark zunehmenden Datenfehler in dieser Arbeit nicht zu beobachten war.

Auch zeigte sich für die IA32-Architektur in den Messungen eine hohe Abhängigkeit der Ergebnisse von den Eigenschaften des Maschinenprogramms der Anwendung bezüglich unausgerichteter Sprünge in die Mitte von Instruktionskodierungen. Es wäre interessant zu erproben, in wie weit die Erzeugung von in dieser Hinsicht günstiger Sequenzen in der Codegeneration eines Übersetzers sinnvoll berücksichtigt werden kann.

Abkürzungsverzeichnis

KESO	KESO a Multi-JVM for OSEK based Systems
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
ROM	Read Only Memory
CFCSS	Control Flow Checking by Software Signatures
YACCA	Yet Another Control flow Checking Approach
ECCA	Enhanced Control-flow Checking using Assertions
CCA	Control-flow Checking using Assertions
FAIL*	Fault Injection Leveraged
SMC	The State Machine Compiler
JINO	jino
RTL	Register Transfer Language
CIAO	CiAO is Aspect Oriented
IA32	Intel Architektur 32
CISC	Complex Instruction Set Computer

Abbildungsverzeichnis

1.1 Schematischer Überblick des KESO-Systems (Quelle: [11]) 2

2.1 Platzierung von Prolog und Epilog bei verzweigenden und verzweigungsfreien Basisblöcken 7
2.2 Beispiel eines Kontrollflussgraphen, in Anlehnung an Goloubeva et al. [5] . 8
2.3 Fehlerkategorisierung gemäß Goloubeva et al. [6] 9

3.1 Umsetzung von virtuellen Aufrufen, in Anlehnung an [7] 13
3.2 Knotenübergänge im sequentiellen Fall und bei Sammelknoten im Sinne von [4] 23
3.3 Übergänge and Sammelpunkten mittels Differenzsignaturen 25
3.4 Aliasierung bei CFCSS gemäß [4] 26
3.5 Eingangs und Ausgangssignaturen bei YACCA 31
3.6 Die Bitsets für $M1$ bei einem Prolog. In $M1$ sind abschließend alle Bits gesetzt, die in allen Vorgängern in ihrem Wert übereinstimmen 32
3.7 Eine Beispielbelegung für einen in CFCSS aliasierenden Kontrollflussgraphen. Die Berechnung für den illegalen Übergang BB_1 nach BB_5 resultiert in einem ungültigen Wert der `code`-Variable 33
3.8 Kontrollflussgraph und zugehöriger Dominatorbaum. Unterbäume des Dominatorbaumes bilden dabei Regionen 37
3.9 Schematische Abbildung des instrumentierten Kontrollflussgraphen in Anlehnung an [7] 38
3.10 Schematische Abbildung des instrumentierten Kontrollflussgraphen nach Optimierung, in Anlehnung an [7]. Durch die Optimierung wird auch der Übergang $BB_3 \rightarrow BB_6$ möglich. 39

4.1 Abbildung der umgesetzten TCP-Zustandsmaschine (Quelle: [28]) 47
4.2 Verteilung der Ereignistypen für die ungehärteten Anwendungen 59
4.3 Injektionsergebnisse für die Injektion des Befehlszählers bei der Matrixmultiplikation, Darstellung in logarithmischer Skala 61

4.4 Beispiel für drei unausgerichtete Zugriffe an den Adressen `0x1014b3`, `0x1014b5` und `0x1014b6` für die `dom:small` Programmvariante 62
4.5 Injektionsergebnisse für die Injektion des Befehlszählers bei der Sortieranwendung, Darstellung in logarithmischer Skala 65
4.6 Injektionsergebnisse für die Injektion des Befehlszählers bei der Sortieranwendung, Darstellung anteilig bezüglich der pro Variante erzeugten Kontrollflussfehler 66
4.7 Injektionsergebnisse für die Injektion des Befehlszählers bei der Zustandsmaschine, Darstellung in logarithmischer Skala 69
4.8 Injektionsergebnisse für die Injektion des Befehlszählers bei der Zustandsmaschine beschränkt auf die reine Applikation, Darstellung in logarithmischer Skala 70
4.9 Anzahl der `FAIL_MARKER`-Ereignisse nach Verfahren und Anwendung, anteilig zur Menge der injizierten Fehler (oben) und absolut (unten) 73
4.10 Anzahl der `FAIL_MARKER`-Ereignisse bei der Injektion aller Einbitfehler in Programm und Daten 75

Tabellenverzeichnis

4.1 Gängige Metriken der Testanwendungen 48
4.2 Artefaktgrößen der Anwendung nach Härtung mittels unterschiedlicher Verfahren, sowohl absolut als auch relativ zur unmodifizierten Referenzanwendung 50
4.3 Laufzeiten der verschiedenen Anwendungen nach Härtung mittels unterschiedlicher Verfahren, Datenspeicher im internen LDRAM 51
4.4 mittlere Länge der Instruktionskodierung sowie Anteil und Anzahl der durch unausgerichtete Sprünge verursachten Fehler für die Matrixmultiplikation . 64
4.5 mittlere Länge der Instruktionskodierung sowie Anteil und Anzahl der durch unausgerichtete Sprünge verursachten Fehler für die Sortieranwendung . . 67

Literaturverzeichnis

[1] M. Rebaudengo, M. S. Reorda, M. Torchiano und M. Violante, „Soft-error detection through software fault-tolerance techniques," in *Defect and Fault Tolerance in VLSI Systems, 1999. DFT'99. International Symposium on*. IEEE, 1999, S. 210–218.

[2] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda und M. Violante, „Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *IEEE Transactions on Nuclear Science*, Vol. 47, Nr. 6, S. 2231–2236, 2000.

[3] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy und J. A. Abraham, „Design and evaluation of system-level checks for on-line control flow error detection," *Parallel and Distributed Systems, IEEE Transactions on*, Vol. 10, Nr. 6, S. 627–641, 1999.

[4] N. Oh, P. Shirvani und E. McCluskey, „Control-flow checking by software signatures," *Reliability, IEEE Transactions on*, Vol. 51, Nr. 1, S. 111–122, Mar. 2002.

[5] O. Goloubeva, M. Rebaudengo, M. S. Reorda und M. Violante, „Soft-error detection using control flow assertions," in *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*. IEEE, 2003, S. 581–588.

[6] O. Goloubeva, M. Rebaudengo, M. Reorda und M. Violante, *Software-implemented hardware fault tolerance*. Springer, 2006.

[7] C. Dietrich, „Global Optimization of Non Funcitonal Properties in OSEK Real-Time Systems by Static Cross-Kernel Flow Analyses," Masterarbeit, Universität Erlangen, Deutschland, Sep. 2014.

[8] V. Narayanan und Y. Xie, „Reliability concerns in embedded system designs," *Computer*, Vol. 39, Nr. 1, S. 118–120, Jan. 2006.

[9] S. Borkar, „Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, Vol. 25, Nr. 6, S. 10–16, Nov. 2005.

[10] G. Miremadi und J. Torin, „Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection," *Reliability, IEEE Transactions on*, Vol. 44, Nr. 3, S. 441–454, Sep. 1995.

[11] C. Lang, „Compiler-Assisted Memory Management Using Escape Analysis in the KESO JVM," Masterarbeit, Universität Erlangen, Deutschland, 6 2014.

[12] C. Erhardt, M. Stilkerich, D. Lohmann und W. Schröder-Preikschat, „Exploiting static application knowledge in a Java compiler for embedded systems: a case study," in *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*. ACM, 2011, S. 96–105.

[13] OSEK Group *et al.*, „VDX Operating System 2.2.3," Feb. 2005.

[14] X. Castillo, S. McConnel und D. Siewiorek, „Derivation and Calibration of a Transient Error Reliability Model," *Computers, IEEE Transactions on*, Vol. C-31, Nr. 7, S. 658–671, July 1982.

[15] R. Vemu und J. A. Abraham, „Ceda: Control-flow error detection through assertions," in *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*. IEEE, 2006, S. 6–pp.

[16] A. V. Aho, M. S. Lam, R. Sethi und J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2. Aufl. Pearson International, 2007.

[17] G. A. Kanawati, V. S. Nair, N. Krishnamurthy und J. A. Abraham, „Evaluation of integrated system-level checks for on-line error detection," in *Computer Performance and Dependability Symposium, 1996., Proceedings of IEEE International*. IEEE, 1996, S. 292–301.

[18] R. M. Stallman *et al.*, *Using and porting the GNU compiler collection*. Free Software Foundation, 1999, Vol. 86.

[19] I. W. G. 14/N1256, „ISO/IEC 9899:TC3 - Commitee Draft," ISO/IEC, Tech. Rep., Sep. 2007.

[20] U. Infineon Technologies: TriCore Design Group, Bristol. (2005, Feb.) TriCore 1 v1.3: Volume2: Instruction Set.

[21] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer und J. Vitek, „CD x: a family of real-time Java benchmarks," in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*. ACM, 2009, S. 41–50.

[22] O. Goloubeva, M. Rebaudengo, M. S. Reorda und M. Violante, „Improved software-based processor control-flow errors detection technique," in *Reliability and Maintainability Symposium, 2005. Proceedings. Annual*. IEEE, 2005, S. 583–589.

[23] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman und F. K. Zadeck, „Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 13, Nr. 4, S. 451–490, 1991.

[24] I. Bronstein, K. Semendjajew, G. Musiol und H. Mühlig, „Taschenbuch der Mathematik," *Verlag Harri Deutsch*, Vol. 1, 2008.

[25] T. Lengauer und R. E. Tarjan, „A Fast Algorithm for Finding Dominators in a Flowgraph," *ACM Trans. Program. Lang. Syst.*, Vol. 1, Nr. 1, S. 121–141, Jan. 1979.

[26] G. N. Frederickson, „Optimal Algorithms for Tree Partitioning," in *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, Serie SODA '91. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1991, S. 168–177.

[27] S. Kundu und J. Misra, „A linear tree partitioning algorithm," *SIAM Journal on Computing*, Vol. 6, Nr. 1, S. 151–154, 1977.

[28] I. Griffin. (2009, Nov.) TCP state machine - Image. Nutzung gemäß der LPPL. [Online]. Verfügbar: http://www.texample.net/tikz/examples/tcp-state-machine/

[29] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher und O. Spinczyk, „CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems." in *USENIX Annual Technical Conference*, 2009.

[30] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann und O. Spinczyk, „FAIL*: Towards a Versatile Fault-Injection Experiment Framework," in *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, Serie Lecture Notes in Informatics, G. Mühl, J. Richling und A. Herkersdorf, Hrsgg., Vol. 200. Deutsche Gesellschaft für Informatik, Mar. 2012, S. 201–210.

[31] H. Schirmeier, C. Borchert und O. Spinczyk, „Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors," in *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*. IEEE Computer Society Press, Juni 2015, To appear.

[32] Intel, *Intel 64 and IA-32 Architectures. Software Developer's Manual. Volumes 2 (2A, 2B & 2C), Instruction Set Reference, A-Z*, Jan. 2015.