technische universität
dortmund

Masterarbeit

Beschleunigung von Fehlerinjektions-experimenten durch Zusammenfassung von Zustandsraum-transformationen

Merlin Stampa
28. August 2017

Betreuer:
Dr.-Ing. Horst Schirmeier
Prof. Dr.-Ing. Olaf Spinczyk

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Arbeitsgruppe Eingebettete Systemsoftware
`http://ess.cs.tu-dortmund.de`

Zusammenfassung

Fehlerinjektion ist ein experimentelles Werkzeug zur Untersuchung der Zuverlässigkeit von Hard- und Softwaresystemen. Mit dieser Methodik werden insbesondere durch Strahlungseffekte hervorgerufene Hardwarefehler imitiert, um so systematisch die Fehlertoleranz des verwendeten Computersystems – einschließlich verwendeter Software-implementierter Hardwarefehlertoleranzverfahren – zu analysieren und zu bewerten. Werkzeuge wie das am Lehrstuhl entwickelte FAIL* [Sch+15] erkunden den Fehlerraum, indem für jeden Fehlerinjektionspunkt eine Simulation durchgeführt wird. Die Laufzeit solcher Kampagnen ist aufgrund der vergleichbar langsamen Geschwindigkeit der Simulatoren und der schieren Anzahl der Experimente typischerweise enorm hoch.

Das Ziel dieser Masterarbeit besteht in der Beschleunigung der Simulationen durch Ausnutzung der Tatsache, dass diese in großen Teilen ähnlich oder identisch ablaufen. Während *Checkpoints* – die Sicherung des kompletten Systemzustandes zu festgelegten Zeitpunkten – eine relativ grobgranulare und zeitlich nach der Fehlerinjektion selten anwendbare Methode darstellen [Ber+02], soll in dieser Arbeit eine feingranulare und leichtgewichtige Variante als Erweiterung zu FAIL* entwickelt werden. Das zu untersuchende Programm wird nach der Durchführung des *Golden Run* – eines Simulationslaufs ohne Fehlerinjektion – geeignet in sogenannte *überspringbare Abschnitte* unterteilt und deren gelesener und geschriebener Maschinenzustand (die „Zustandsraumtransformation“) aufgezeichnet. Während eines Fehlerinjektionsexperiments sollen solche Abschnitte „übersprungen“ werden können, sofern der beobachtete Zustand zur Aufzeichnung passt. Dadurch soll die langsame Ausführung der darin enthaltenen Instruktionen vermieden und somit die Laufzeit verkürzt werden. Der Ansatz wurde anhand von fünf Benchmarks der MiBench-Suite [Gut+01] – aufgesetzt auf das Embedded-Betriebssystem eCos [Mas03] – mit Hilfe geeigneter Metriken evaluiert. Die Methode konnte für vier der Benchmarks Beschleunigungen von 12,5 % bis 56 % erreichen, bei einem Benchmark kam es jedoch zu einer Erhöhung der Kampagnenlaufzeit von ca. 6 %.

Inhaltsverzeichnis

Abkürzungsverzeichnis **iii**

1 Einleitung **1**
1.1 Motivation und Zielsetzung 2
1.2 Gliederung der Arbeit 3

2 Grundlagen und verwandte Arbeiten **5**
2.1 Fehler 5
2.2 Simulationsbasierte Fehlerinjektion 7
2.3 FAIL* 8
2.3.1 Ablauf des Fault Tolerance Assessment Cycle in FAIL* . . . 9
2.3.2 Architektur 10
2.4 Verwandte Arbeiten 11
2.4.1 Fehlerraumreduktion 12
2.4.2 Reduktion der Experimentlaufzeit 15
2.5 Zusammenfassung 17

3 Analyse und Entwurf **19**
3.1 Problemanalyse 19
3.2 Lösungsansatz 20
3.2.1 Aufzeichnung eines überspringbaren Abschnitts 22
3.2.2 Ablauf eines Sprungs 24
3.2.3 Bestimmung einer geeigneten Unterteilung 26
3.2.4 Einschränkungen 27
3.3 Einbettung in FAIL* 28
3.4 Zusammenfassung 29

4 Implementierung **31**
4.1 Übersicht 31
4.2 Datenstrukturen 32
4.3 Erweiterung des Tracings 32
4.3.1 Unterteilung eines Trace in überspringbare Abschnitte . . . 34
4.3.2 Anfertigung von Aufzeichnungen 35

4.4 Datenbank-Import und Filterung 36
4.5 Plugin zum Durchführen der Sprünge 37
4.5.1 Einlesen der Daten 38
4.5.2 Blockierung durch Experiment-Listener 39
4.5.3 Zustandsraumtransformation 42
4.5.4 Sammeln statistischer Informationen 43
4.6 Einschränkungen 44
4.7 Zusammenfassung 44

5 Evaluation **45**
5.1 Kriterien 45
5.2 Verwendete Benchmarks 46
5.3 Beschleunigung eines Golden Run 47
5.4 Empirische Untersuchung von Effektivität und Ergebnistreue 47
5.4.1 Aufbau 48
5.4.2 Ergebnisse 50
5.5 Diskussion der Ergebnisse 55
5.6 Zusammenfassung 56

6 Zusammenfassung **59**
6.1 Fazit 59
6.2 Ausblick 60

Literaturverzeichnis **61**

Abbildungsverzeichnis **65**

Tabellenverzeichnis **67**

Listingverzeichnis **69**

Abkürzungsverzeichnis

AES	*Advanced Encryption Standard*. 46
ALU	*Arithmetic Logic Unit*. 6
CPU	*Central Processing Unit*. 1, 2, 6, 8, 11, 13, 15, 16, 22, 23, 26, 36, 42, 48
EDM	*Error-Detection Mechanism*. 1, 7
EEA	*Execution-Environment Abstraction*. 10
ERM	*Error-Recovery Mechanism*. 1, 16, 20
FI	Fehlerinjektion. 2, 5–7, 9, 11, 14–17, 20–22, 25, 35, 53–57, 59
FPGA	*Field Programmable Gate Array*. 15
FPU	*Floating Point Unit*. 44
IP	*Instruction Pointer*. 23, 42
ISA	*Instruction-Set Architecture*. 6, 13
MMIO	*Memory Mapped I/O*. 27, 46, 55, 59, 60
PC	*Program Counter*. 23
RAMS	*Reliability, Availability, Maintainability, Safety*. 5
SDC	*Silent Data Corruption*. 7, 20, 52
SEU	*Single Event Upset*. 1
SIHFT	*Software-Implemented Hardware Tolerance*. 1, 6, 7, 13
SWIFI	*Software-Implemented Fault Injection*. 7
VM	Virtuelle Maschine. 48

1 Einleitung

Im Bereich der eingebetteten Systeme ist ein klarer Trend zu höheren Transistordichten und niedrigeren Versorgungsspannungen zu beobachten [IRC13]. Dies erhöht die Wahrscheinlichkeit für Hardwarefehler, die u. a. durch Strahlungseffekte, Variationen im Herstellungsprozess, Alterung oder Hitze [DW11; NX06] verursacht werden können. Hardwarefehler können permanent (bleiben für unbekannte Zeit bestehen), intermittierend (erscheinen nach dem ersten Auftreten erneut) oder transient (auftretend und verschwindend) sein [Muk11]. Diese Arbeit fokussiert sich insbesondere auf transiente Hardwarefehler (engl.: *Single Event Upset*s (SEUs) oder *Soft Errors*). Transiente Hardwarefehler treten vorrangig nach einer Ionisierung der Halbleiter durch einschlagende Partikel – hauptsächlich Alphateilchen aus der direkten Umgebung des Mikroprozessors und Neutronen aus der Atmosphäre – auf und manifestieren sich beispielsweise durch einzelne oder mehrere gekippte Bits (engl.: *Bit Flips*) in Speicherzellen oder CPU-Registern.

Während solche Hardwarefehler in vielen Anwendungen lediglich monetären Schaden verursachen, können sie in sicherheitsrelevanten Bereichen zu einer direkten Gefahr für menschliches Leben werden [Yos13]. Entwickler müssen deswegen in der Lage sein, ihre Systeme fehlertolerant auszulegen. In einigen Anwendungsfeldern wird dies durch die Verwendung redundanter Hardware erreicht: So sind bspw. kritische Bauteile in Flugzeugen oft redundant verbaut, sodass ein teilweiser Ausfall kompensiert werden kann [BT93]. Im Kontext eingebetteter Systeme bringt der Einsatz redundanter Hardware jedoch oft eine nicht vertretbare Steigerung der Herstellungskosten und des Energieverbrauchs mit sich. Hier ist deswegen auf eine andere Lösung zu setzen: Der Redundanz auf Softwareebene. So können Berechnungen mehrfach ausgeführt oder Daten redundant gespeichert werden, wobei die Software in die Lage versetzt wird, Diskrepanzen zu erkennen und idealerweise auch zu korrigieren – engl.: *Error-Detection Mechanism* (EDM) und *Error-Recovery Mechanism* (ERM). Hierbei ist *Software-Implemented Hardware Tolerance* (SIHFT) der Oberbegriff für diese Mechanismen [Gol+06]. Ein Nachteil von SIHFT ist, dass die Mechanismen zur Fehlerdetektion und -korrektur selbst CPU-Zeit und Speicher beanspruchen, womit sie sowohl den Ressourcenbedarf der Anwendung erhöhen, als auch mehr potentielle „Angriffsfläche“ für transiente Fehler bieten, was wiederum die Robustheit des Systems verschlechtern könnte [BSS13; Sch16]. Der Einsatz von SIHFT sollte deswegen immer zielgerichtet und anwendungsspezifisch erfolgen. An-

statt also grundlegend jede Komponente der Software zu schützen, sollten folgende Schritte durchlaufen werden:

1. Auffinden und Analysieren kritischer Programmteile (Funktionen, Datenstrukturen u. ä.).

2. Auswahl, Implementierung und funktionaler Test geeigneter Schutzmechanismen für diese Teile.

3. Messung und Vergleich der Fehlertoleranz des gesamten Programms in geschützten und ungeschützten Varianten.

Im Rahmen der Systementwicklung erfolgen diese Maßnahmen idealerweise iterativ im sogenannten *Fault Tolerance Assessment Cycle* [Sch+15], sodass die Änderung der Fehlertoleranz über verschiedene Softwareversionen hinweg beobachtet werden kann. Um Systementwickler bei dieser Aufgabe zu unterstützen, wurden Werkzeuge entwickelt, die automatisiert die Robustheit eines Programms gegenüber Hardwarefehlern analysieren können. Ein dafür häufig verwendeter Ansatz ist die Fehlerinjektion (FI) [ZA04; KD14], d. h. die gezielte Einbringung eines künstlichen Fehlers in den Programmablauf mit anschließender Beobachtung des Systemverhaltens.

1.1 Motivation und Zielsetzung

Eine *Fehlerinjektionskampagne* besteht aus einer Vielzahl von Experimenten, die das Zielprogramm untersuchen. Jedes Experiment injiziert üblicherweise einen einzelnen Fehler an einem bestimmten Zeitpunkt – gemessen in CPU-Zyklen – und einem bestimmten Ort – bspw. einem Bit im Hauptspeicher. Die Ergebnisse der Experimente geben Aufschluss über die Fehlertoleranz des Zielprogramms. Die Menge aller möglichen Injektionskoordinaten eines Zielprogramms wird als *Fehlerraum* bezeichnet. Ist eine detaillierte, feingranulare Analyse der Fehlertoleranz des Zielprogramms gewünscht, muss der Fehlerraum vollständig erkundet werden [Sch16]. Dies ist allerdings, durch die schiere Größe des Fehlerraums, selbst bei vergleichsweise einfachen Zielprogrammen mit einem enormen Rechenaufwand verbunden, was eine praktikable und wirtschaftliche Durchführung des Fault Tolerance Assessment Cycle erschwert. Diese Problematik wird in Abschnitt 2.4 vertieft.

Die vorliegende Arbeit präsentiert den Ansatz der *überspringbaren Abschnitte* zur Reduktion der durchschnittlichen Experimentlaufzeit, was ebenso wie eine Reduktion des Fehlerraums zu einer Verkürzung der Kampagnenlaufzeit beiträgt [Gol+06]. Die Methode umgeht die verhältnismäßig langsame Ausführung von Instruktionen im Simulator (oder einem anderen Backend) so oft wie möglich durch

die Zusammenfassung und Anwendung sogenannter *Zustandsraumtransformationen*. Dafür wird die Tatsache ausgenutzt, dass sich die Mehrheit der Experimente untereinander sehr stark ähnelt: Durch die genaue Beobachtung und anschließende Zusammenfassung des Verhaltens (der Zustandsraumtransformationen) bestimmter Experimente – insbesondere des *Golden Run*, welcher ein Experiment ohne Fehlerinjektion darstellt – können spätere, ähnlich laufende Experimente beschleunigt werden. Die Anwendung zusammengefasster Zustandstransformationen kommt einem Vorwärtssprung des Zielprogramms gleich (Abb. 1.1). Die simulierte Maschine wird in möglichst kurzer Zeit in einen Zustand versetzt, den sie in längerer Zeit durch die Durchführung einer Vielzahl von dynamischen Instruktionen erreicht hätte.

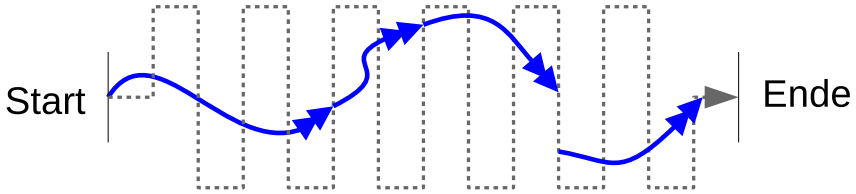


Abb. 1.1: Idee der überspringbaren Abschnitte: Statt alle Instruktionen des Zielprogramms (gestrichelte Linie) zeitaufwendig einzeln zu simulieren, sollen so viele Teile des Programms wie möglich in kürzerer Zeit übersprungen werden (blaue Pfeile). Je nach Experiment können trotzdem Programmteile existieren, die nicht übersprungen werden können (Lücke zwischen den blauen Pfeilen rechts).

Soweit dem Autor bekannt, wurde bis zur Abgabe dieser Arbeit keine vergleichbare Methode publiziert. Der Ansatz soll als Erweiterung zu FAIL* (*FAult Injection Leveraged*) [Sch+15] implementiert werden, einem Framework für Fehlerinjektionskampagnen, welches im Rahmen der Dissertation von Horst Schirmeier [Sch16] entstand.

1.2 Gliederung der Arbeit

Die Arbeit beginnt in Kapitel 2 mit einigen Grundlagen wie Fehlermodellen und der simulationsbasierten Fehlerinjektion, stellt FAIL* vor und untersucht verwandte Arbeiten zur Beschleunigung von Fehlerinjektionskampagnen. Nachfolgend werden in Kapitel 3 eine detaillierte Problemanalyse und der entwickelte Lösungsansatz präsentiert. Wie dieser Ansatz als Erweiterung zu FAIL* implementiert wurde, ist Kapitel 4 zu entnehmen. Kapitel 5 untersucht die Leistungsfähigkeit und Grenzen der Methode mittels unterschiedlicher Benchmarks und selbst gewählter

Evaluationsmetriken. Die Arbeit schließt in Kapitel 6 mit einer Zusammenfassung der Ergebnisse und einem Ausblick auf mögliche Verbesserungen.

2 Grundlagen und verwandte Arbeiten

Dieses Kapitel präsentiert eine Einführung in die simulationsbasierte Fehlerinjektion und das dafür entwickelte Framework FAIL*. Anschließend folgt eine Übersicht verwandter Arbeiten.

2.1 Fehler

An technische Systeme werden unterschiedliche Qualitätsanforderungen gestellt. So können beispielsweise eine hohe Performanz oder ein großer Funktionsumfang gefordert sein. In sicherheitskritischen Anwendungen – bspw. in Fahrzeugen, medizinischen Geräten oder Kraftwerken – sind jedoch besonders die sogenannten RAMS-Eigenschaften [DIN00] entscheidend. Die Abkürzung steht im *Reliability, Availability, Maintainability, Safety*, auf Deutsch *Zuverlässigkeit, Verfügbarkeit, Instandhaltbarkeit und Sicherheit*. RAMS definiert die Zuverlässigkeit als die Fähigkeit eines Systems, seine Funktion über eine bestimmte Betriebszeit in korrekter Weise zu erfüllen. Die Verfügbarkeit ist die Wahrscheinlichkeit, ein System zu einem beliebigen Zeitpunkt fehlerfrei anzutreffen. Die Instandhaltbarkeit repräsentiert den Aufwand, der betrieben werden muss, um ein System nach einem Ausfall zu reparieren. Das Merkmal Sicherheit gibt die Gefährdungsfreiheit des Systems an, d. h. wie gut das System die Gefährdung seiner Umwelt vermeidet. Avižienis et al. [Avi+04] definieren zusätzlich die Eigenschaft *Integrity* (deutsch: *Integrität*) als die „Abwesenheit von unzulässigen Zustandsänderungen“.

Um ein technisches System hinsichtlich dieser Kriterien zu bewerten, sind nicht nur funktionale Analysen durchzuführen, sondern auch die Auswirkungen von unerwarteten Fehlern zu untersuchen, die die Integrität des Systems verletzen. Im Kontext der in dieser Arbeit betrachteten elektronischen Systeme ist der Begriff Fehler genauer zu differenzieren:

- Als *Fehlerursache* (engl.: *Fault*) wird der physikalische Defekt bzw. die Störstelle in einer Hard- oder Softwarekomponente bezeichnet. Defekte können permanenter[1], wiederkehrender oder transienter Natur sein.

- Der *Fehlzustand* (engl.: *Error*) ist die Manifestation des Defekts, bspw. in Form abweichender interner Daten.

- Ein *Fehlverhalten* (engl.: *Failure*) ist schließlich zu beobachten, wenn ein Fehlzustand von außen sichtbar wird. Er kann sich zu anderen Komponenten ausbreiten und zu der Entstehung eines Defekts auf höherer Abstraktionsschicht führen.

Als Fehlerkette bezeichnet man den Prozess des Übergangs von Defekt zu Fehlverhalten [Hof16]. Die vorliegende Arbeit beschränkt sich auf Fehler in Hardwarekomponenten zur Analyse SIHFT-gestärkter Systeme, reine Softwarefehler durch falsche Programmierung u. ä. werden nicht betrachtet.

Ein Beispiel, adaptiert von [Sch16, 3.1.3]: In einem x86-Mikroprozessor kann ein Hardware-Defekt auftreten, wenn ein Transistor unerwartet umschaltet – bspw. aufgrund von elektromagnetischer Strahlung. Ist der Transistor Teil der Eingabe eines logischen Gatters, produziert dieses Gatter nun möglicherweise eine logische 1 statt einer 0. Ist das Gatter wiederum Teil einer *Arithmetic Logic Unit* (ALU), die zwei Zahlen addiert, stimmt auch die berechnete Summe nicht mehr. Dieses falsche Ergebnis wird nun in der höher gelegenen Schicht der *Instruction-Set Architecture* (ISA) im `EAX`-Register der *Central Processing Unit* (CPU) gespeichert, womit ein Fehlzustand vorliegt. Im Vergleich zum korrekten Ergebnis wurde ein Bit von 0 auf 1 gekippt. Bei anschließender Rückgabe dieses falschen Ergebnisses an den Benutzer wird von einem Fehlverhalten gesprochen.

Hardwarefehler auf Transistorebene zu modellieren stellt im Rahmen der simulationsbasierten FI (s. u.) einen unverhältnismäßig hohen Aufwand dar; einerseits aufgrund der Komplexität, andererseits auch, weil für viele Mikroprozessoren und Speichertypen keine ausreichend genauen Modelle zur Verfügung stehen. Werkzeuge zur SIHFT-Analyse modellieren Fehler deswegen erst ab einer höheren Abstraktionsebene, üblicherweise der ISA-Schicht. Dies hat den u. a. den Grund, dass SIHFT-Mechanismen erst auf dieser Schicht Fehler erkennen und korrigieren können.

[1] Ein permanenter Speicherfehler z. B. liegt dann vor, wenn ein Bit nicht mehr beschreibbar ist, sondern konstant den selben Wert behält.

2.2 Simulationsbasierte Fehlerinjektion

Wie Kapitel 1 darlegt, dient der sogenannte *Fault Tolerance Assessment Cycle* [Sch+15] zur iterativen Schwachstellenanalyse und Verbesserung eines Systems in Bezug auf Hardwarefehler. Um kritische Datenstrukturen, Funktionen und andere Programmteile zu identifizieren, hat sich die Fehlerinjektion (FI) als Methode etabliert [ZA04]. Dabei wird gezielt ein Defekt in den Programmablauf eingebracht und so die Entstehung einer Fehlerkette angeregt. Durch Beobachtung (engl.: *Monitoring*) des weiteren Verhaltens sind Rückschlüsse auf die Robustheit des Systems und die Effektivität eingesetzter SIHFT-Schutzmechanismen möglich. Nach Abschluss eines solchen Experiments lässt sich das Ergebnis in eine von mehreren Kategorien einordnen. Ergebniskategorien können bspw. die folgenden sein:

- *Kein Effekt*: Das Programm verhielt sich trotz des Fehlers korrekt.
- *Detektion*: Das Programm konnte den Fehler mittels eines *Error-Detection Mechanism* (EDM) erkennen und darauf reagieren, z. B. eine Warnung ausgeben.
- *Silent Data Corruption* (SDC): Das Programm konnte den Fehler nicht detektieren und verhält sich augenscheinlich normal, allerdings werden aufgrund veränderter Daten inkorrekte Ausgaben erzeugt.
- *Timeout*: Das Programm konnte nicht innerhalb einer vorgegebenen Zeitschranke terminieren, z. B. weil der Fehler eine Endlosschleife erzeugte.
- *Trap*: Das Programm versuchte eine unerlaubte Operation auszuführen (bspw. Division durch Null) und musste vorzeitig abgebrochen werden.

Werden Fehler mittels physikalischer Quellen direkt in die für das Zielprogramm dedizierte Hardware injiziert, spricht man von *Hardware Fault Injection* [Son+11]. Werden die Fehler mittels Software in die Hardware injiziert, wird dies als *Software-Implemented Fault Injection* (SWIFI) bezeichnet Die vorliegende Masterarbeit ist dagegen im Bereich der *simulationsbasierten Fehlerinjektion* angesiedelt. Hierbei wird ein geeigneter Simulator eingesetzt, um sowohl das Zielprogramm, als auch die Fehlerinjektion auszuführen. Zwischen Simulator und realer Hardware können Unterschiede bestehen, die das Programmverhalten beeinflussen und die Übertragbarkeit der im Simulator gewonnenen Ergebnisse einschränkt. Zudem ist eine Simulation für gewöhnlich deutlich langsamer als eine native Ausführung auf realer Hardware. Die simulationsbasierte Fehlerinjektion bietet im Vergleich zu anderen Methoden aber auch signifikante Vorteile [Sch16]:

- Eine Aufbereitung (d. h. Modifikation) des Zielprogramms, welche eine Messverfälschung zur Folge haben könnte, ist nicht nötig.
- Fehler können sehr präzise injiziert werden, sowohl in Bezug auf den Zeitpunkt als auch den Ort (die Hardwarekomponente).
- Es besteht keine Gefahr, dass Entwicklungshardware beschädigt wird.
- Jedes Experiment lässt sich absolut deterministisch wiederholen.
- Experimente sind parallelisierbar.

Für die Durchführung simulationsbasierter Fehlerinjektion existieren Tools wie *GangES* [Har+14], *SmartInjector* [LT13] und das in dieser Arbeit erweiterte FAIL*. Kooli und Di Natale präsentieren in [KD14] eine Übersicht weiterer Tools.

2.3 Fail*

FAIL*[2] ist ein Framework für Fehlerinjektionskampagnen, welches im Rahmen der Dissertation [Sch16] von Horst Schirmeier entstand. Ein Einstieg findet sich in [Sch+15]. Zum vollständigen Verständnis der vorliegenden Arbeit (insbesondere der Implementierung, beschrieben in Kapitel 4) sind Kenntnisse über die Funktionsweise und den Aufbau dieses Tools notwendig. In diesem Textabschnitt werden deswegen die nötigen Teile aus [Sch16] wiedergegeben und zusammengefasst.

FAIL* basiert vorrangig auf dem Modell transienter, gleichverteilter *Single Bit Flips* in CPU-Registern und Speicher. Zusätzlich kann es Multi-Bit-Fehler [Li+10] durch das Kippen eines ganzen zusammenhängenden Bytes approximieren, und auch permanente Speicherfehler sind simulierbar. Gegenüber ähnlichen Tools zeichnet sich FAIL* insbesondere durch folgende Vorteile aus:

- FAIL* ist in der Lage, den gesamten Fehlerraum zu prüfen, und nicht nur eine heuristisch gewählte Probe daraus (vgl. 2.4.1).
- Es werden unterschiedliche *Backends* unterstützt: Bochs [Law96], QEMU [Bel05], gem5 [Bin+11] und Hardware[3] via OpenOCD [Rat05].

[2] FAIL* steht für *FAult Injection Leveraged*, das Asterisk symbolisiert die unterschiedlichen Backends. Es kann von `https://github.com/danceos/fail` bezogen werden.

[3] Diese Masterarbeit fokussiert sich auf simulationsbasierte Fehlerinjektion, da dies die Implementierung und Evaluation des Ansatzes vereinfacht. Prinzipiell sind aber auch Experimente mit Hardware-Backend beschleunigbar.

- Experimentbeschreibungen können leicht für verschiedene Zielprogramme wiederverwendet werden.
- Durch eine Reihe spezialisierter Tools wird eine detaillierte, feingranulare Analyse der Ergebnisse ermöglicht.

2.3.1 Ablauf des Fault Tolerance Assessment Cycle in Fail*

Der grundlegende Ablauf eines *Fault Tolerance Assessment Cycle* im Kontext von Fail* (veranschaulicht durch Abb. 2.1) ist wie folgt [Sch16, 4.5.1]:

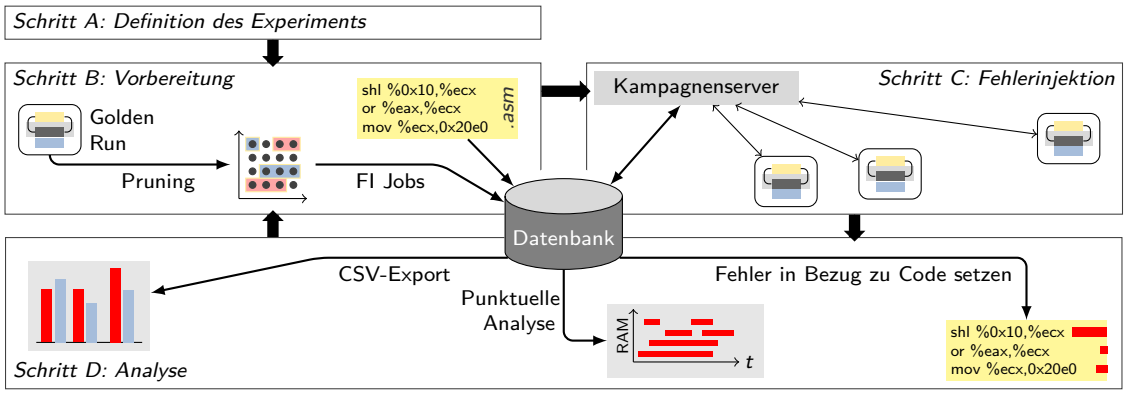


Abb. 2.1: Fault Tolerance Assessment Cycle im Kontext von Fail*. *Adaptiert von [Sch16, Abb. 4.10].*

Schritt A: Der Anwender definiert ein Experiment in C++. Viele Anwendungsfälle werden allerdings schon durch das mitgelieferte `generic-experiment` abgedeckt.

Schritt B: Der sogenannte *Golden Run* (ein Experiment ohne Fehlerinjektion) wird ausgeführt, wobei alle ausgeführten Instruktionen und Speicherzugriffe im sogenannten *Trace* (deutsch: Ablaufverfolgung) aufgezeichnet werden. Aus dem Trace lässt sich der Fehlerraum ableiten und anschließend reduzieren (Abschnitt 2.4.1). Die durchzuführenden Experimente (und ggf. weitere Informationen) werden in eine Datenbank importiert.

Schritt C: Die Fehlerinjektionskampagne wird ausgeführt. Dafür generiert der Kampagnenserver aus der Datenbank *Jobs* und verteilt sie per TCP an aktive Clients. Die Clients melden ihre Ergebnisse an den Kampagnenserver zurück, welcher Sie wiederum in die Datenbank kopiert.

Schritt D: Die Ergebnisse, die nun als Tabelle in der Datenbank vorliegen, können vom Benutzer analysiert werden. Die verschiedenen Resultate lassen sich bspw. nach Quelltextabschnitten oder Datenstrukturen aufschlüsseln, außerdem existieren Möglichkeiten zur Visualisierung.

2.3.2 Architektur

FAIL* lässt sich grob in zwei Schichten unterteilen: Der *Plumbing Layer* (deutsch: Klempnereischicht) und der *Assessment Layer* (deutsch: Begutachungsschicht), welches auf dem Plumbing Layer aufsetzt und davon abstrahiert [Sch16, 4.2]. Der Plumbing Layer (Abb. 2.2) stellt grundlegende Funktionalitäten bereit, welche die Parallelisierung von Experimenten erlaubt, d. h. insbesondere die Server/Client-Infrastruktur. Weiterhin kapselt es die konkrete Ausführungsumgebung (das Backend) durch das *Execution-Environment Abstraction* (EEA) *Layer*.

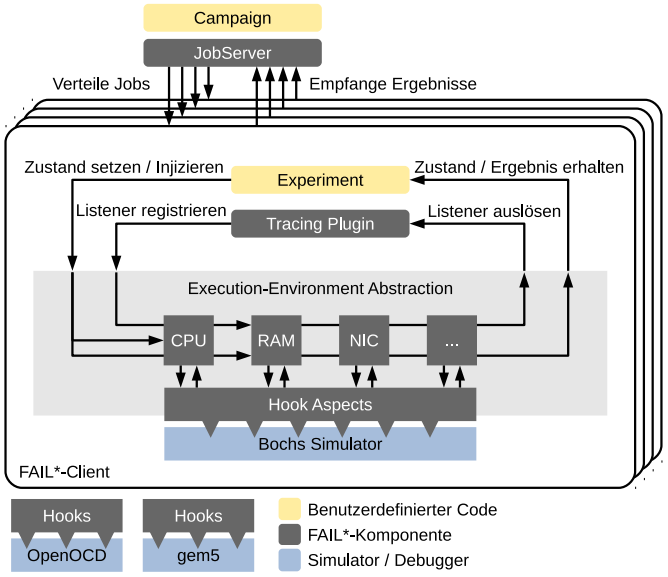


Abb. 2.2: Übersicht der Architektur des Plumbing Layer von FAIL*. *Adaptiert von [Sch16, Abb. 4.1].*

Ein FAIL*-Server kontrolliert eine Kampagne. Er generiert sogenannte *Jobs*, d. h. Parametersätze, die jeweils ein durchzuführendes Experiment repräsentieren. Diese werden zur Bearbeitung an die Clients verteilt, die im Anschluss ihre Ergebnisse zurückmelden. Ein FAIL*-Client dient zur Durchführung von Experimenten. Er besteht im Wesentlichen aus dem Backend und FAIL*-Komponenten, die dieses Backend kontrollieren. Dazu wird mit Koroutinen gearbeitet. Das Backend, das Experiment und geladene *Plugins*[4] stellen jeweils eigene Kontrollflüsse dar, zwischen denen FAIL* umschalten kann. Die Umschaltung erfolgt im Normalfall beim Auslösen eines *Listeners*. Listener (deutsch: Zuhörer) können vom Experiment und den Plugins beim `ListenerManager` registriert werden und warten auf

[4]Zusatzprogramme, welche über eine vordefinierte Schnittstelle eingebunden werden und den Funktionsumfang des Experiments erweitern.

das Eintreten eines bestimmten Ereignisses im Zielprogramm (engl.: *Trigger*). Die Basisklasse `BaseListener` verfügt zudem über einen setzbaren Zähler. Er wird jedes Mal dekrementiert, wenn der Listener aktiviert wird (bspw. wenn eine bestimmte Instruktion erreicht wurde). Der Listener löst erst dann vollständig aus, wenn dieser Zähler den Wert 0 erreicht. Für die verschiedenen Ereignistypen existieren verschiedene Subklassen von `BaseListener`, u. a. die folgenden:

- `BPListener` sind an *Breakpoints* angelehnt, sie warten auf das Erreichen bestimmter Instruktionen. Die Subklasse `BPSingleListener` lässt sich auf eine einzelne statische Instruktion oder jede Instruktion (Parameter `ANY_ADDR`) einstellen. Die Klasse `BPRangeListener` dagegen erlaubt das Warten auf das Erreichen einer Instruktion innerhalb eines definierten Bereiches. Viele Experimente verwenden `BPSingleListener`, die auf `ANY_ADDR` und einen bestimmten Zählerwert eingestellt sind, um einen bestimmten CPU-Zyklus im Backend zu erreichen und dort die Fehlerinjektion durchzuführen.

- `MemAccessListener` warten darauf, dass das Zielprogramm auf einen bestimmten Speicherbereich zugreift. Sie können so eingestellt werden, dass sie bei Lesezugriffen, Schreibzugriffen oder beidem aktiviert werden.

- `TimerListener` warten darauf, dass ein backend-spezifischer *Timer* auslöst, d. h. die interne Uhr des Simulators einen bestimmten Wert erreicht.

2.4 Verwandte Arbeiten

Fehlerinjektionskampagnen sind ein geeignetes Mittel zur Analyse der Fehlertoleranz eines elektronischen Systems (2.2). Ein zentraler Nachteil dieser Methode ist der benötigte zeitliche Aufwand [Sch16, 1.2.3].

Sei Δt die Anzahl der vom Zielprogramm benötigten CPU-Zyklen, und Δm die Anzahl aller vom Programm verwendeten Speicherbits. Betrachtet man Experimente, bei denen das Kippen eines einzelnen Bits simuliert wird (engl.: *Single Bit Flip*), ergibt sich für jede mögliche Kombination aus CPU-Zyklus und Speicherbit je ein Fehlerinjektionsexperiment. Die Anzahl aller möglichen Experimente (der Fehlerraum, illustriert in Abb. 2.3) errechnet sich ergo aus $w = \Delta t \cdot \Delta m$. Dieses Produkt kann selbst bei vergleichsweise simplen Zielprogrammen schnell astronomische Größen erreichen. Schirmeier et al. verwenden für dieses Problem den Begriff *Fault Space Explosion* (deutsch: *Fehlerraumexplosion*) [SBS15; Hof+16].

Wenn $T_{Experiment}$ die durchschnittliche Laufzeit eines Experiments bezeichnet, errechnet sich die Laufzeit der Kampagne insgesamt zu $T_{Kampagne} = w \cdot T_{Experiment}$. Selbst unter Zuhilfenahme massiver Parallelisierung wären die Kampagnen, die

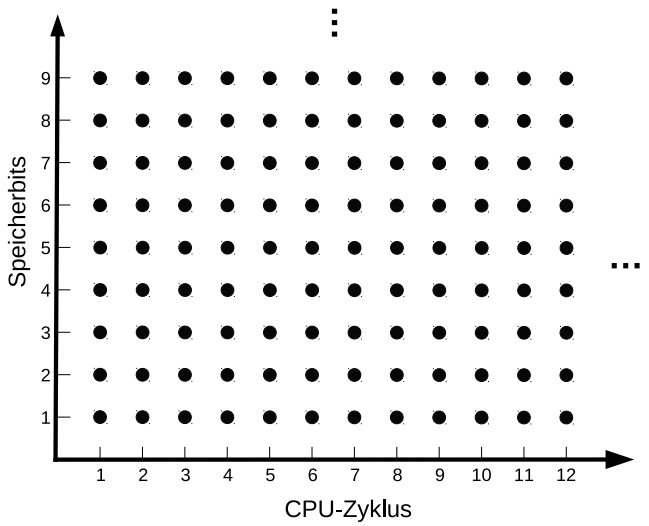


Abb. 2.3: Fehlerraum für Single-Bit-Flip-Experimente. Jeder schwarze Punkt steht für ein mögliches Experiment. *Adaptiert von [Sch16, Abb. 1.1].*

sich mit dieser Methode ergäben, schlicht zu zeitaufwendig, um durchführbar zu sein. Eine Reduktion der Laufzeit ist also unabdingbar. Um die Laufzeit einer Kampagne zu reduzieren, muss entweder die Anzahl der Experimente N, oder die durchschnittliche Experimentlaufzeit $T_{Experiment}$ verkleinert werden [Gol+06].

In der Literatur finden sich Methoden (2.4.1) um die Anzahl der Experimente N um mehrere Größenordnungen reduzieren – bspw. die statistische Stichprobenauswahl oder *Def/use Pruning* –, sodass $N \ll w$, ohne die Genauigkeit der Ergebnisse zu stark zu gefährden. Kampagnen für typische Zielprogramme können nach dieser Reduktion auf entsprechend leistungsfähigen Computerclustern innerhalb mehrerer Stunden bis Tage (statt Jahre) durchgeführt werden. Allerdings ist eine weitere Reduktion von $T_{Kampagne}$ wünschenswert, um die Durchführung von Kampagnen praktikabler und wirtschaftlicher zu machen. Dies gilt insbesondere vor dem Hintergrund, dass der *Fault Tolerance Assessment Cycle* – und damit auch Messungen in Form von Kampagnen – in der Systementwicklung idealerweise iterativ durchlaufen werden soll. Abschnitt 2.4.2 erläutert dazu Methoden, die – wie auch die vorliegende Arbeit – versuchen, eine Reduktion von $T_{Experiment}$ zu erreichen.

2.4.1 Fehlerraumreduktion

Wie soeben erläutert, wächst die Größe w des *Fehlerraums*, welcher die Koordinaten aller möglichen Fehlerinjektionspunkte repräsentiert, quadratisch mit der von

dem Zielprogramm verwendeten Menge des verwendeten Speichers und der benötigten CPU-Zyklen. Versucht man, ein Experiment für jeden Fehlerinjektionspunkt durchzuführen, setzt also die Anzahl durchzuführender Experimente $N = w$, wäre die Durchführung einer Kampagne selbst bei sehr kurzen Experimentlaufzeiten und dem Ausnutzen von Parallelisierungsmöglichkeiten nicht in einem akzeptablen Zeitrahmen möglich. Eine Reduktion von N ist also unabdingbar. FAIL* implementiert die Fehlerraumreduktion im Tool `prune-trace` .

2.4.1.1 Statistische Stichprobenauswahl

Eine simple heuristische Methode um eine hinreichend kleine Anzahl von Experimenten zu erreichen, ist das statistische Auswählen von Stichproben (engl.: *statistical sampling*) [Lev+09]. Dabei werden so lange zufällig Experimente aus dem Fehlerraum ausgewählt, bis die Wahrscheinlichkeit, dass die spätere Ergebnisverteilung der des kompletten Fehlerraums entspricht, innerhalb eines vorgegebenen Konfidenzintervalls liegt.

Je mehr Experimente durchgeführt werden, desto genauer sind die Ergebnisse. Im Umkehrschluss gilt dagegen: Je weniger Experimente durchgeführt werden, desto höher ist die Wahrscheinlichkeit, dass Programmteile in ihrer Relevanz für die Robustheit unter- oder überschätzt werden und kritische Stellen möglicherweise gar nicht erst identifiziert werden können. Einige der Analysetools von FAIL* setzen eine vollständige Abdeckung des Fehlerraums voraus, um die Experimentergebnisse in Bezug zu einzelnen Zeilen der Quellcodeebene oder Datenstrukturen setzen zu können [Sch16, 4.5.5]. Dies ist wünschenswert, um Aussagen über die Robustheit des Zielprogramms auf möglichst feingranularer Ebene treffen zu können und – auf Basis dieses Wissens – SIHFT-Mechanismen so gezielt wie möglich einzusetzen.

2.4.1.2 Def/use Pruning

Die Methode des *Def/use Pruning* erlaubt – im Gegensatz zur statistischen Stichprobenauswahl – eine Reduktion des Fehlerraums ohne Verlust der Genauigkeit. Zu den ersten, die diese Technik beschrieben und veröffentlichten, gehörten Smith et al. [Smi+95] sowie Günthoff und Sieh [GS95]. Das Def/use Pruning basiert auf zwei grundlegenden Beobachtungen (illustriert durch Abb. 2.4):

1. Ein auf der ISA-Schicht injizierter Defekt in ein Speicherbit[5] wird erst dann aktiviert, wenn dieses Bit durch eine nachfolgende Instruktion gelesen wird (*use*, in der Abbildung „R“ für engl. *read*).

[5] Def/use Pruning lässt sich problemlos für alle Ebenen der Speicherhierarchie verwenden, d. h. nicht nur für Hauptspeicher, sondern auch für Cache und Register.

2. Ist der nächste Speicherzugriff auf ein Bit nach der FI schreibend (*def*, im Bild „W“ für engl. *write*), eliminiert dies den Defekt, da das fehlerhafte Bit überschrieben wird (sofern es sich um einen transienten, und keinen permanenten Fehler handelt).

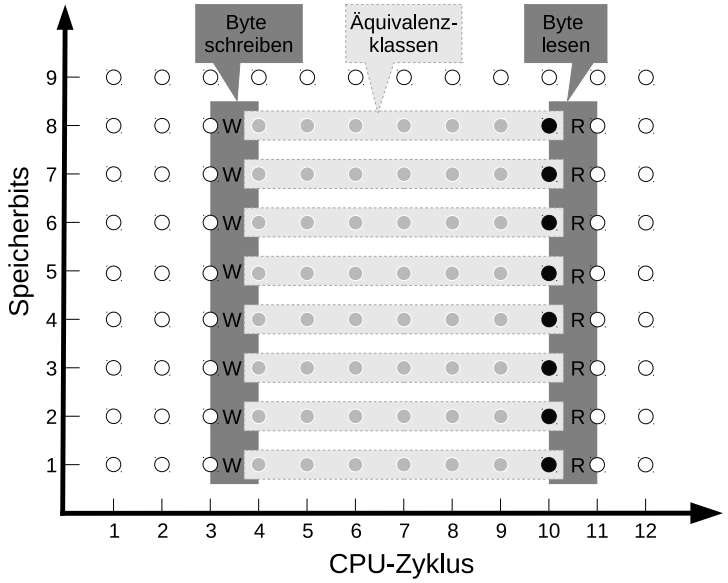


Abb. 2.4: Def/use Pruning für Single-Bit-Flip-Experimente. Experimente zwischen dem Schreiben und Lesen eines Speicherbits (graue Punkte) gehören Äquivalenzklassen an (hellgraue Rechtecke), von denen jeweils nur ein Experiment (schwarzer Punkt) durchgeführt werden muss. Die weißen Kreise repräsentieren Experimente, die aussortiert werden können, da dort injizierte Fehler niemals gelesen (aktiviert) werden. *Adaptiert von [Sch16, Abb. 3.3].*

Die erste Beobachtung impliziert, dass hinsichtlich der Fehlerinjektion alle Zeitpunkte zwischen einem Schreib- und Lesezugriff auf dasselbe Bit zu einer sogenannten *Äquivalenzklasse* (engl. *Equivalence Class*) zusammengefasst werden können, da derselbe Fehlerzustand erzeugt wird. Von jeder Äquivalenzklasse muss lediglich ein einzelnes Experiment durchgeführt werden, um ein Ergebnis zu erhalten, welches sich auf die anderen Experimente übertragen lässt. Weiterhin können – gemäß der zweiten Beobachtung – alle Fehlerinjektionen nach einem Lesezugriff bis zum nächsten Schreibzugriff eines Speicherbits aussortiert werden, da das Überschreiben den Defekt eliminiert und das Experiment somit dem Golden Run entsprechen wird (Ergebnis „Kein Effekt“). Ein Nebeneffekt der Methode ist, dass die Experimente auf diejenigen Speicherbits eingeschränkt werden, die vom Zielprogramm auch tatsächlich benutzt werden.

Die Zusammenfassung von Fehlerinjektionsexperimenten zu Äquivalenzklassen – auf Basis der zuvor aufgezeichneten Speicherzugriffe – kann die Anzahl der durchzuführenden Experimente N um mehrere Größenordnungen $N \ll w$ reduzieren[6] Falls die verbleibende Anzahl der Experimente nach wie vor einen zu hohen Aufwand darstellt, können Heuristiken herangezogen werden, um eine weitere Reduktion zu erreichen. Hari et al. fassen in ihrem *Relyzer* genannten Tool [Har+13] Äquivalenzklassen zu größeren Gruppen zusammen, bei denen jeweils ein durchzuführendes Experiment (der sogenannte *Pilot*) ausgewählt wird, dessen Ergebnis nicht nur die seiner Äquivalenzklasse, sondern die der gesamten Gruppe repräsentieren soll. Allerdings bietet die Methode keine Möglichkeit, die Genauigkeit über eine Erhöhung von N einstellen zu können, und setzt zudem einige Annahmen über das Fehlermodell und mögliche Ergebnisarten voraus. Ähnliche Probleme existieren beim *SmartInjector* Tool von Li und Tan [LT13]. Horst Schirmeier beschreibt in Kapitel 5 seiner Dissertation [Sch16] die Methode des *Fault Similarity Pruning*, mit welcher die Möglichkeit eröffnet wird, den *Trade-Off* zwischen der Anzahl der Experimente und der Genauigkeit frei zu wählen. Dabei wird keine Einschränkung der möglichen Ergebnisarten vorgenommen und eine Abdeckung des kompletten Fehlerraums gewährleistet.

2.4.2 Reduktion der Experimentlaufzeit

Die Laufzeit eines Experiments ist primär durch die Anzahl der dynamischen Instruktionen und der Geschwindigkeit der Ausführungsumgebung bestimmt. Die Verwendung einer möglichst schnellen Ausführungsumgebung – bspw. ein Prototyp der für die Anwendung vorgesehenen Hardware – stellt also eine naheliegende Maßnahme zur Reduktion der Laufzeit dar. *Field Programmable Gate Array*s (FPGAs) können als vergleichsweise schnelles Backend verwendet werden, wenn kein Hardware-Prototyp zur Verfügung steht [Ber+02; Pel+08]. Dennoch kann der Einsatz simulationsbasierter FI erforderlich oder wünschenswert sein. Dies ist bspw. dann der Fall, wenn eine hohe Zahl von CPUs zur Verfügung steht, sodass die Kampagne von der Parallelisierung der Experimente profitieren kann, oder falls nur ein Simulator die Anforderungen des gewählten Fehlermodells erfüllen kann.

Simulatoren weisen üblicherweise sehr langsame Ausführungsgeschwindigkeiten auf (Abschnitt 2.2), der hohe Grad der Kontrolle über die simulierte Maschine eröffnet jedoch andere Optimierungsmöglichkeiten. Es kann versucht werden, die Anzahl der tatsächlich simulierten dynamischen Instruktionen zu verringern, und zwar einerseits in der Phase vor der Injektion und andererseits danach.

[6] Barbosa et al. [Bar+05] berichten von einer Reduktion von $5 \cdot 10^8$ auf $7,7 \cdot 10^6$ Experimente für einem Benchmark mit Fehlerinjektionen in Register, sowie einer Reduktion von $1,9 \cdot 10^{11}$ auf $3,3 \cdot 10^6$ Experimente für einem anderen mit Fehlerinjektionen in den Hauptspeicher.

2.4.2.1 Vor der Fehlerinjektion

In der *Präinjektionsphase* (auch *fast-forwarding* genannt) werden die dynamischen Instruktionen einzeln nacheinander ausgeführt (engl.: *Single Stepping*) bis der CPU-Zyklus, bei dem die FI erfolgen soll, erreicht wurde. Um diese Phase abzukürzen, kann das Konzept der *Checkpoints*[7] (deutsch: Prüfpunkt) [Par+00; Ber+02; Har+14] verwendet werden.

Checkpoints sind Abbilder von Maschinenzuständen, die zu a priori festgelegten (üblicherweise äquidistanten) Zeitpunkten angefertigt werden. Der Maschinenzustand kann später aus einem solchen Abbild wiederhergestellt werden. Ist ein solcher Checkpoint zu einem Zeitpunkt vor der Fehlerinjektion verfügbar, kann durch die Wiederherstellung des Maschinenzustandes die Ausführung der Instruktionen vor dem Checkpoint umgangen werden, was einen Laufzeitvorteil bringt, sofern der Wiederherstellungsvorgang schneller abläuft als Single Stepping.

Die Methode hat zwei wesentliche Nachteile: Einerseits ist die zu sichernde Datenmenge – und damit auch der Aufwand für die Wiederherstellung – nicht unbeträchtlich; sie umfasst u. a. die Inhalte aller Register und jeder Speicherzelle. Andererseits können Checkpoints üblicherweise nur vor der Fehlerinjektion angewandt werden. Der Grund dafür ist, dass eine Fehlerinjektion den Maschinenzustand verändert, was dazu führt, dass die a priori angefertigten Abbilder mit einer hohen Wahrscheinlichkeit nicht mehr zu 100 % mit den beobachteten Maschinenzuständen übereinstimmen. Dies zu prüfen ist aufgrund der hohen Datenmenge relativ zeitaufwendig. Selbst die Frage, wann solche Prüfungen durchzuführen sind, ist nach der Fehlerinjektion nicht trivial zu beantworten. So könnte ein Fehlerkorrekturmechanismus (ERM) den injizierten Fehler zwar korrigieren und damit den gewünschten Maschinenzustand wiederherstellen. Die Korrektur hätte allerdings im Vergleich zum Golden Run eine zeitliche Verzögerung des Programmablaufs verursacht.

Der in dieser Arbeit entwickelte Ansatz der *überspringbaren Abschnitte* baut auf dem Checkpoint-Konzept auf und versucht, die genannten Nachteile zu minimieren. Kapitel 3 beschreibt den Entwurf im Detail.

2.4.2.2 Nach der Fehlerinjektion

Um die Postinjektionsphase (das *Monitoring*) zu beschleunigen, kann versucht werden die Simulation vorzeitig abzubrechen, wenn sich das Ergebnis vorhersehen

[7] Vergleichbare Konzepte finden sich in vielen Bereichen der Informatik, teilweise ist statt „Checkpoint“ auch die Bezeichnung „Snapshot“ (deutsch: Schnappschuss) üblich [CL85; AL80].

lässt. Dies kann auf Grundlage des beobachteten Maschinenzustands und zuvor abgeschlossener Fehlerinjektionsexperimente geschehen.

So schlagen Berrojo et al. [Ber+02] das *Dynamic Fault Collapsing* vor, bei dem der Maschinenzustand in quadratisch wachsenden Abständen zum Fehlerinjektionspunkt mit dem Golden Run verglichen wird. Weicht der Maschinenzustand in einem einzelnen Bit ab, kann versucht werden, ein äquivalentes Experiment zu finden, welches den gleichen Fehler aufwies und dessen Ergebnis bekannt ist. Das laufende Experiment kann in diesem Fall frühzeitig beendet werden, da das Ergebnis des äquivalenten Experimentes übernommen werden kann. Li und Tan verwenden im Tool *SmartInjector* [LT13] einen ähnlichen Ansatz, welcher jedoch nicht den vollständigen Maschinenzustand mit dem Golden Run vergleicht, sondern das Verhalten einer heuristisch getroffenen Auswahl von Ausgabe-, Maskierungs- und Sprunginstruktionen. Im *GangES* Tool von Hari et al. [Har+14] wird die Idee weiterentwickelt: Es gruppiert FI-Experimente in sogenannte *Gangs*, die zu dem gleichen zwischenzeitlichen Maschinenzustand führen, sodass nur eines der Experimente vollständig durchgeführt und beendet werden muss. Zu welchen Zeitpunkten die Simulationen verglichen werden und welche Teile des Maschinenzustands geprüft werden, entscheidet GangES laut Aussage der Autoren heuristisch anhand der Programmstruktur.

2.5 Zusammenfassung

Hardwarefehler stellen für jedes elektronische System ein schwer kalkulierbares Risiko dar. Wie Abschnitt 2.1 zeigte, können sich Defekte schnell von der Transistorebene zu höheren Abstraktionsschichten ausbreiten und dort zu ungewolltem oder sogar gefährlichem Verhalten führen. Zur Beurteilung der Fehlertoleranz eines Systems hat sich die die in Abschnitt 2.2 vorgestellte Methode der Fehlerinjektion etabliert. Während die simulationsbasierte FI, d. h. die Durchführung von FI-Experimenten in einem Simulator statt auf realer Hardware den Nachteil von langsamen Ausführungsgeschwindigkeiten aufweist, bietet sie auch erhebliche Vorteile in Bezug auf Reproduzierbarkeit und Parallelisierung. Das in Abschnitt 2.3 vorgestellte Framework Fail* dient zur Durchführung von Fehlerinjektionskampagnen. Es wurden die grundlegende Funktionsweise und die für das Verständnis dieser Arbeit relevanten Teile der Architektur erörtert. Abschnitt 2.4 legte dar, warum die Durchführung von Fehlerinjektionskampagnen für gewöhnlich mit einem enormen Rechenaufwand verbunden ist. Es folgte eine Übersicht aus der Literatur bekannter Methoden zur Reduktion der Kampagnenlaufzeit über eine Reduktion der Anzahl durchzuführender Experimente und der durchschnittlichen Experimentlaufzeit.

3 Analyse und Entwurf

In diesem Kapitel soll das Konzept der *überspringbaren Abschnitte* zur Laufzeitreduktion von Fehlerinjektionsexperimenten entworfen werden. Es stellt den Kern der vorliegenden Arbeit dar und ist die Grundlage der in Kapitel 4 präsentierten Implementierung.

3.1 Problemanalyse

Wie Abschnitt 2.4 beschrieb, weisen Fehlerinjektionskampagnen für gebräuchliche Fehlermodelle einen beachtlichen Zeit- und Ressourcenbedarf auf. Auch nach der Reduktion des Fehlerraums durch Def/use Pruning (Abschnitt 2.4.1.2) kann die Anzahl der durchzuführenden Experimente mehrere Millionen erreichen [Bar+05]. Um die Laufzeit der Kampagnen über eine Beschleunigung der einzelnen Experimente zu reduzieren, können die in Abschnitt 2.4.2 vorgestellten Methoden in Betracht gezogen werden. *Checkpoints* – Sicherungen des gesamten Maschinenzustands zu festgelegten Zeitpunkten – haben den Nachteil, dass sie als vergleichsweise grobgranulare Methode selten nach der Fehlerinjektion anwendbar sind [Ber+02] und relativ viele Daten gespeichert werden müssen. Das *Dynamic Fault Collapsing* [Ber+02] und verwandte Ansätze, wie die von GangES [Har+14] bekannten *Gangs*, versuchen das Experiment vorzeitig abzubrechen, falls das Ergebnis vorhersehbar ist. Diese Methoden sind entweder mit einem hohen Rechenaufwand verbunden oder greifen auf Heuristiken zurück, die zu einer Verfälschung der Ergebnisse führen können.

Die vorliegende Arbeit möchte einen neuen Ansatz entwickeln, um die Laufzeit von Fehlerinjektionsexperimenten – und darüber auch die Laufzeiten ganzer Kampagnen – zu reduzieren, welcher die Nachteile der genannten Methoden minimiert. Konkret sind die folgenden Anforderungen zu erfüllen:

Effektivität: Die Laufzeit von Kampagnen soll möglichst stark verkürzt werden. Erreicht werden soll dies u. a. dadurch, dass eine Anwendung prinzipiell sowohl vor als auch nach der Fehlerinjektion möglich sein soll.

Ergebnistreue: Die Methode soll die Ergebnisse der Kampagne nicht verfäl-

schen. Bei der Anwendung in einem Experiment soll zwar die Laufzeit verkürzt werden, der finale Maschinenzustand aber unverändert sein.

Generalität: Die neue Methode soll für eine größtmögliche Menge unterschiedlicher Zielprogramme und unterschiedlicher Backends geeignet sein.

Nutzerfreundlichkeit: Für die Anwender soll ein möglichst geringer Mehraufwand entstehen. Dies beinhaltet sowohl den Änderungsbedarf für Quellcodes bestehender Experimente, als auch den zeitlichen Aufwand in der Tracingphase. Die Methode sollte sich bspw. mit Kommandozeilenschaltern auf einfache Weise an- oder abschalten lassen.

3.2 Lösungsansatz

Ein Simulationslauf ohne Fehlerinjektion wird als *Golden Run* bezeichnet, er repräsentiert das Sollverhalten des Zielprogramms. Bis zum Zeitpunkt der FI läuft ein Experiment absolut identisch zum Golden Run ab. Danach sind hinsichtlich des Programmverhaltens die folgenden Fälle unterscheidbar:

- Die Injektion hatte keinen Effekt. Die im Experiment ausgeführten Instruktionen und verarbeiteten Daten sind weiterhin die gleichen wie im Golden Run.
- Der Fehler führt zu falschen Ergebnissen (SDC) oder lässt das Zielprogramm gänzlich andere Ausführungspfade durchlaufen, bspw. Endlosschleifen.
- Das System detektiert und korrigiert den Fehler. Nach der Korrektur ist das Programmverhalten wieder identisch zum Golden Run. Allerdings benötigt das Programm bis zur Terminierung im Vergleich zum Golden Run etwas mehr Zeit, da der *Error-Recovery Mechanism* (ERM) eine Verzögerung verursacht hat.
- Der Fehler konnte detektiert, aber nicht korrigiert werden. Das Programm kann den Fehler ignorieren, eine Warnung ausgeben oder sogar eine vorzeitige Terminierung einleiten.

Beobachtung: Die Experimente, von denen es in einer typischen FI-Kampagne enorm viele gibt, ähneln sich untereinander sehr stark (illustriert durch Abb. 3.1). Bis zum Zeitpunkt der FI sind sie stets identisch zum Golden Run. Auch nach der FI können Ähnlichkeiten zum Golden Run auftreten, wenn der injizierte Fehler keinen Effekt hatte oder korrigiert werden konnte.

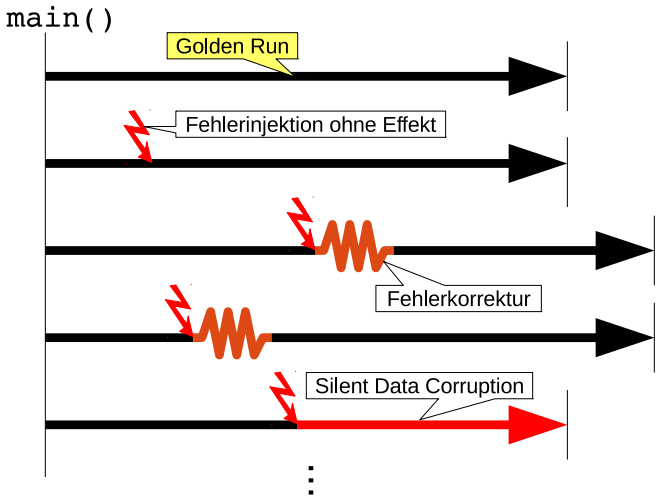
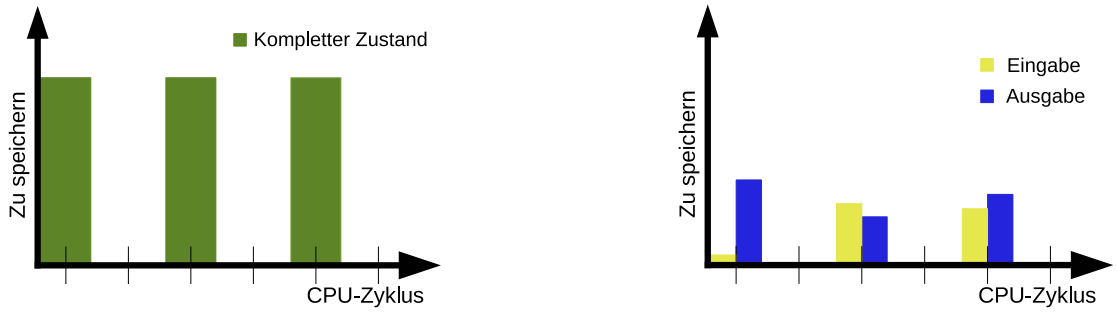


Abb. 3.1: Illustration der Ähnlichkeit verschiedener FI-Experimente. Der vertikale Balken links repräsentiert den Start der Experimente (hier: die `main()`-Funktion), die Balken rechts zeigen jeweils die Terminierung an. Schwarz: Sollverhalten (wie im Golden Run, oberstes Experiment). Rot: Fehlerinjektion oder verändertes Programmverhalten. Orange: Fehlerkorrekturmechanismus.

Die Ähnlichkeit der Experimente kann ausgenutzt werden, um die langsame Ausführung von Instruktionen im Simulator zu umgehen. Dazu werden Mengen von Instruktionen zu sogenannten *überspringbaren Abschnitten* (engl. *skippable sections*) zusammengefasst. Anstatt zeitaufwendig alle Instruktionsauführungen zu simulieren, sollen so oft wie möglich überspringbare Abschnitte angewandt werden, um die gleichen Änderungen des Maschinenzustands in signifikant kürzerer Zeit zu bewirken. Änderungen des Maschinenzustands werden im Kontext dieser Arbeit als *Zustandsraumtransformation* bezeichnet. Sie umfassen insbesondere Schreibzugriffe auf Register und Speicher, aber auch das Voranschreiten der internen Uhr des Simulators.

Der Ansatz ist als leichtgewichtigere Variante von Checkpoints zu verstehen. Im Gegensatz zu Checkpoints werden für überspringbare Abschnitte aber nur diejenigen Teile des Maschinenzustandes (bzw. dessen Änderungen) gespeichert, die für den jeweiligen Abschnitt relevant sind (illustriert in Abb. 3.2). Ein überspringbarer Abschnitt ist ergo auch dann anwendbar, wenn ein fehlerhafter Teil des Zustands von den in dem Abschnitt enthaltenen Instruktionen nicht gelesen wird. Auch zeitliche Verzögerungen, die durch etwaige Korrekturmechanismen verursacht werden können, spielen bei der Anwendung keine Rolle, wenn die Startpunkte der Ab-

schnitte nicht an bestimmte CPU-Zyklen, sondern statische Instruktionen geknüpft werden. Die Sprünge sollen das Experimentergebnis – soweit wie möglich – nicht verfälschen. Deswegen darf die Prüfung, ob ein Abschnitt übersprungen werden darf, nicht heuristisch erfolgen.



(a) Checkpoints **(b)** Überspringbare Abschnitte

Abb. 3.2: Vergleichende Illustration der zu speichernden Daten für drei Checkpoints (a) und drei überspringbare Abschnitte (b).

Da alle Experimente – zumindest vor der Fehlerinjektion, und teilweise auch danach – dem Golden Run ähneln, ist es naheliegend, diesen zur Unterteilung von überspringbaren Abschnitten heranzuziehen. Es ist aber auch denkbar, während einer Kampagne neue Abschnitte zu „lernen“, falls bestimmte fehlerbehaftete Zustände gehäuft in mehreren Experimenten auftreten. Nachfolgende Experimente, die den selben Fehlzustand aufweisen, können gleichermaßen von der Überspringung eines solchen Abschnitts profitieren, als würde dieser aus dem Golden Run stammen.

3.2.1 Aufzeichnung eines überspringbaren Abschnitts

Für jeden Abschnitt wird mindestens eine sogenannte *Aufzeichnung* (engl.: *Record*) angefertigt. Die Aufzeichnung teilt sich vorrangig in *Eingabe* und *Ausgabe* auf (vgl. Abb. 3.2b). Die Eingabe umfasst die Teile des Maschinenzustandes, die von den Instruktionen des Abschnitts gelesen werden. Genauer gesagt sind dies die Werte aller gelesenen Register und Speicherzellen zu Beginn des Abschnitts. Falls ein Wert während des Abschnitts verändert und später erneut gelesen wurde, ist dies aus Sicht des Abschnitts nicht relevant. Nur der jeweils erste gelesene Wert wird aufgezeichnet. Die Ausgabe dagegen fasst die im Abschnitt getätigten Zustandsraumtransformationen zusammen. Für jedes Register und jede Speicherzelle, die von den Instruktionen des Abschnitts geschrieben werden, muss der Wert

aufgezeichnet werden, der am Ende des Abschnitts vorliegt. Zusätzlich muss erfasst werden, wie viele CPU-Zyklen der Abschnitt überspringt, um die interne Uhr des Backends korrekt vorspulen zu können. Der Befehlszähler – *Instruction Pointer* (IP) oder *Program Counter* (PC) – ist immer Teil der Ausgabe: Durch den dafür gespeicherten Wert wird bestimmt, welche Instruktion nach dem Sprung als nächstes ausgeführt werden soll.

Listing 3.1 zeigt ein kurzes Beispiel. Es enthält die Disassemblierung[1] einer einfachen Funktion namens `square()`, welche das Quadrat einer als Parameter übergebenen ganzen Zahl berechnet. Die statische Instruktion `10001a`, die den Beginn der Funktion darstellt, soll auch der Startpunkt eines überspringbaren Abschnitts sein, der die Funktion umfasst. Wie sich aus den Maschinenbefehlen ableiten lässt, werden die Register `EBP`, `ESP` sowie der Stack gelesen. Diese beiden Register und der Teil des Stack, der den Aufrufparameter enthält, sowie die zu Beginn des Abschnitts enthaltenen Werte, werden als Eingabe aufgezeichnet. Weiterhin ist zu sehen, dass die Funktion die Register `EBP` und `ESP` auch wieder beschreibt. Auch das Register `EAX` wird beschrieben, es enthält am Ende der Funktion ihren Rückgabewert. Ebenso sind bestimmte Speicheradressen Teil der Ausgabe, da der `push`-Befehl an der Adresse `10001a` den Wert des `EBP`-Registers auf den Stack kopiert. Der `ret`-Befehl bei Adresse `100025` verlässt die Funktion, was das Ende des Abschnitts markiert. Der Ausgabewert für den Befehlszähler ergibt sich in diesem Beispiel aus dem Stack, welcher die vom Aufrufer der `square()`-Funktion hinterlegte Rücksprungadresse enthält, die ergo auch der Eingabe hinzuzufügen ist. Die Aufzeichnung wird in Abb. 3.3 illustriert.

```
1 0010001a < _Z6squarei >:
2   10001a: 55              push %ebp
3   10001b: 89 e5           mov %esp,%ebp
4   10001d: 8b 45 08        mov 0x8(%ebp),%eax
5   100020: 0f af 45 08     imul 0x8%ebp),%eax
6   100024: 5d              pop %ebp
7   100025: c3              ret
```

Lst. 3.1: Disassemblierte `square()`-Funktion.

Da der Startpunkt eines Abschnittes durch eine statische Instruktionsadresse anstatt einem CPU-Zyklus definiert wird, kann ein solcher Startpunkt auch im Golden Run mehrfach erreicht werden. Dabei muss der vorgefundene Maschinenzustand nicht unbedingt der gleiche sein. Im Beispiel von Listing 3.1 kann dies bspw.

[1] Die Dissassemblierung wurde mit dem GNU-Tool *objdump* durchgeführt.

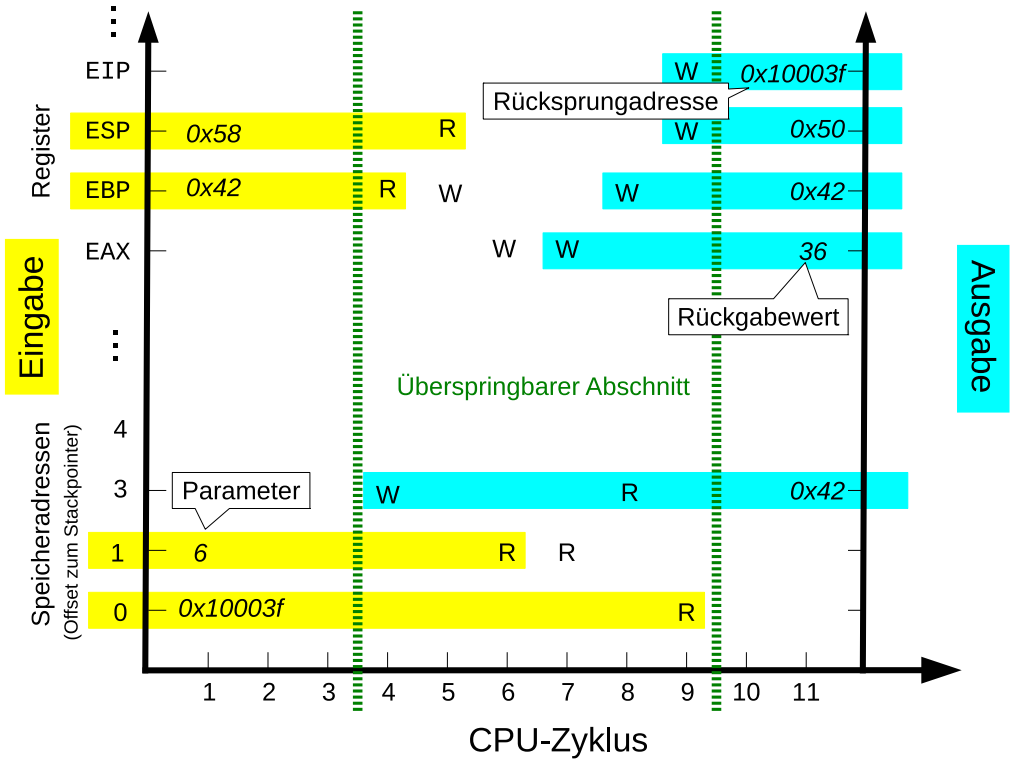


Abb. 3.3: Illustration einer Aufzeichnung für das den beispielhaften überspringbaren Abschnitt (grün) aus Listing 3.1. Im Eingabeteil werden die gelb markierten Register, Speicheradressen und ihre Werte gespeichert. Im Ausgabeteil die türkis markierten.

eintreten, wenn die `square()` -Funktion mehrfach aufgerufen wird. Selbst falls der übergebene Parameter identisch ist, können die Werte für die Rücksprungadresse und den Stackpointer (und damit auch die aufzuzeichnenden Speicheradressen) unterschiedlich sein. Einem überspringbaren Abschnitt müssen deswegen mehr als eine Aufzeichnung zugeordnet werden können. Dies impliziert auch, dass die Anzahl der übersprungenen Instruktionen variieren kann, da unterschiedliche Eingabewerte über bedingte Sprünge (z. B. `jne`) zu unterschiedlichen Ausführungspfaden führen können.

3.2.2 Ablauf eines Sprungs

Der Start eines überspringbaren Abschnitts wird durch eine statische Instruktionsadresse angezeigt. Erreicht die Simulation diese Instruktionsadresse, kann geprüft werden, ob eine zugeordnete Aufzeichnung anwendbar ist. Dies ist genau dann der

Fall, wenn innerhalb des Abschnitts kein Listener des Experiments auslösen soll und der aktuelle Maschinenzustand zu dem Eingabeteil passt (*Match*).

FAIL*-Experimente registrieren üblicherweise *Listener* (Abschnitt 2.3.2), um den Ablauf der Simulation zu kontrollieren und verschiedene Ergebnisarten zu erfassen. So ist für gewöhnlich ein `BPSingleListener` auf die dynamische Instruktion registriert, bei der die Fehlerinjektion erfolgen soll. Ein weiterer löst beim Erreichen des Endes des Programms aus, und ein `TimerListener` registriert das Überschreiten eines a priori festgelegten Zeitlimits. Das Überspringen eines Abschnitts darf niemals das Auslösen eines solchen Listeners verhindern, da sonst die Ergebnisse des Experiments verfälscht werden könnten. Es muss ergo ein Mechanismus existieren, mit dem solche Listener die Anwendung bestimmter Aufzeichnungen blockieren können.

Liegt keine Blockade vor, wird die Aufzeichnung in der *Matchingphase* mit dem aktuellen Maschinenzustand verglichen. Die Einträge der Aufzeichnung entscheiden darüber, welche Teile des Maschinenzustandes betrachtet werden müssen (für das Beispiel aus Listing 3.1: Register `EBP`, `ESP`, bestimmte Adressen des Stacks). Sind die in allen betrachteten Registern und Speicherzellen enthaltenen Werte identisch mit denen, die die Aufzeichnung erwartet, liegt ein *Match* vor. In diesem Fall darf die Aufzeichnung angewandt werden.

Ist eine Aufzeichnung anwendbar, wird ein Sprung durchgeführt, andernfalls muss die Simulation normal fortgesetzt werden. Unter einem Sprung ist die Übernahme des Ausgabeteils in den Maschinenzustand zu verstehen: Speicher und Register werden mit den Werten beschrieben, die auch die enthaltenen Instruktionen hineingeschrieben hätten, und die interne Uhr des Simulators wird vorgestellt. Damit wurden die dynamischen Instruktionen übersprungen. Im Anschluss kann geprüft werden, ob ein nächster überspringbarer Abschnitt direkt an das Ende des letzten Sprungs anschließt, um auch für diesen eine Prüfung der Anwendbarkeit der zugeordneten Aufzeichnungen einzuleiten. Ist dies nicht der Fall, wird die Simulation fortgesetzt. Abb. 3.4 illustriert diese Vorgehensweise.

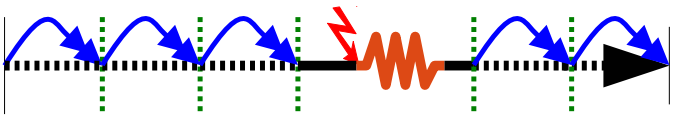


Abb. 3.4: Illustration einer Simulation mit fünf Sprüngen (blau). Die vertikalen Linien repräsentieren die Start- und Endpunkte der Sprünge. Lediglich der Abschnitt, der die Fehlerinjektion enthält (roter Blitz), wird nicht übersprungen, sondern normal simuliert (durchgängige schwarze Linie). Da der Fehler korrigiert werden konnte (orange Linie), sind auch nach der Fehlerinjektion Sprünge möglich.

3.2.3 Bestimmung einer geeigneten Unterteilung

Bei der Durchführung des Golden Run zeichnet FAIL* den sogenannten *Trace* auf, in welchem chronologisch alle dynamischen Instruktionen und Speicherzugriffe festgehalten werden. Mit Hilfe dieser Datei lassen sich die dynamischen Instruktionen des Golden Run in überspringbare Abschnitte einteilen. In diesem Schritt ist nicht vorhersagbar, welche Unterteilung die größte Laufzeitverbesserung für die Kampagne erreicht. Es sind allerdings einige Einflussfaktoren bestimmbar:

- *Länge:* Da die zu implementierenden Algorithmen für Matching, Sprünge u. ä. selbst CPU-Zeit beanspruchen (*Overhead*), existiert eine Mindestanzahl von Instruktionen, ab der ein Sprung einen Laufzeitvorteil bringt. Diese ist abhängig von der Geschwindigkeit des Backends in Relation zur Geschwindigkeit der FAIL*-Komponenten auf der verwendeten Hardware. Je mehr Instruktionen übersprungen werden, desto mehr Laufzeit kann potentiell eingespart werden. Es ist jedoch zu erwarten, dass längere Abschnitte seltener anwendbar sind, da sich mit jeder hinzugenommen Instruktion die Wahrscheinlichkeit erhöht, dass der beim Startpunkt vorgefundene Maschinenzustand zu keiner Aufzeichnung passt oder der Sprung durch einen Listener des Experiments blockiert wird.

- *Aufzeichnungsgröße:* Um zu prüfen ob ein Abschnitt anwendbar ist, müssen nacheinander alle aufgezeichneten Register- und Speicherwerte des Eingabeteils einer Aufzeichnung mit dem aktuellen Maschinenzustand verglichen werden. Dies hat eine Komplexität von $\mathcal{O}(n)$, d. h. die Anzahl der Einträge sollte für eine möglichst kurze Prüfung möglichst klein gehalten werden. Eine geschickte Sortierung der Einträge könnte vorteilhaft sein, um so früh wie möglich Diskrepanzen zu erkennen und das Matching abzubrechen. Analog müssen bei erfolgreicher Prüfung – ebenfalls in $\mathcal{O}(n)$ – die Einträge des Ausgabeteils in den Maschinenzustand übernommen werden.

- *Statische Adresse des Startpunktes:* Wie in Abschnitt 3.2.2 beschrieben, soll FAIL* jedes Mal versuchen, eine passende Aufzeichnung zu finden, wenn während der Simulation eine Instruktion erreicht wird, deren statische Adresse den Start eines überspringbaren Abschnitts markiert. Es ist denkbar, dass dies während des Experiments – insbesondere während nicht-übersprungener Abschnitte – sehr häufig der Fall ist, wobei aber nur für wenige Vorkommen Aufzeichnungen angefertigt wurden. Die Startpunkte sollten also idealerweise selten im Trace enthalten sein, um die Anzahl der Prüfungen mit negativem Ergebnis zu minimieren.

Auf Basis dieser Einflussfaktoren kann ein heuristisches Verfahren entworfen werden, welches zum Ziel hat, eine möglichst günstige Unterteilung für eine möglichst große Menge von Zielprogrammen zu produzieren. Sei l die a priori festgelegte Mindestlänge eines überspringbaren Abschnitts. Das Verfahren soll über alle Instruktionen iterieren und dabei die folgende Schritte durchführen:

1. Wenn kein Startpunkt gesetzt wurde, setze ihn auf die statische Adresse der aktuellen Instruktion.

2. Wurde das Ende des Trace erreicht, verwerfe den aktuellen überspringbaren Abschnitt.

3. Wurde die Mindestlänge l erreicht, beginne die Suche nach einem geeigneten Endpunkt innerhalb einer festgelegten Distanz ($l' < l$) zur aktuellen Instruktion. Der Endpunkt (d. h. die nächste in der Simulation auszuführende Instruktion) ist entweder Ende des Trace oder die Instruktion mit den geringsten Kosten. Die Kosten eines Endpunktkandidaten errechnen sich aus der Häufigkeit der zugrunde liegenden statischen Adresse im gesamten Trace und einer Schätzung der Speichernutzung.

4. Wurde das Ende des Trace erreicht, beende die Unterteilung. Andernfalls gehe zurück zu Schritt 1.

3.2.4 Einschränkungen

In der x86-Architektur erfolgen Ein- und Ausgaben von der bzw. zur seriellen Schnittstelle durch bestimmte Instruktionen wie `in` oder `outs`. Wenn solche Instruktionen Teil eines überspringbaren Abschnitts wären, würden in der Aufzeichnung nur die jeweils ersten Eingaben oder Ausgaben erfasst werden. Da aber insbesondere die Ausgaben über die serielle Schnittstelle in vielen Experimenten relevant für die Unterscheidung zwischen korrektem und fehlerhaftem Programmverhalten sind, sollten die genannten Instruktionen nicht übersprungen werden, d. h. niemals Teil von überspringbaren Abschnitten sein. Dies ist bei der Unterteilung des Trace in überspringbare Abschnitte zu berücksichtigen.

Ein etwas komplexeres Problem ergibt sich, wenn Peripheriezugriffe über *Memory Mapped I/O* (MMIO) erfolgen. Auch bei MMIO entscheidet i. A. nicht nur der letzte geschriebene Wert über den Zustand eines Peripheriegeräts, sondern die gesamte Folge der Schreibzugriffe. Dies müsste bei der Aufzeichnung und Anwendung von überspringbaren Abschnitten berücksichtigt werden, wenn Peripherie für das Programmverhalten relevant ist. Automatisiert zu erkennen welche Bereiche des Speichers nicht nur für reine Daten, sondern MMIO vorgesehen sind, ist allerdings

nicht ganz trivial. Im Rahmen dieser Arbeit wurde dieses Problem deswegen ausgeklammert, da zunächst nur die prinzipielle Nützlichkeit (*Proof-Of-Concept*) des Verfahrens nachgewiesen werden sollte und genügend viele Zielprogramme existieren, die keine Peripheriegeräte benötigen.

Weiterhin sind Timer auf besondere Weise zu berücksichtigen. Timer sind Peripheriegeräte, deren Zustand explizit durch das Zielprogramm verändert werden kann (bspw. indem es einen Timer aktiviert oder seinen Auslösungszeitpunkt setzt), aber auch implizit durch das Voranschreiten der Simulatorzeit. So könnte ein Betriebssystem, auf welchem das eigentliche Zielprogramm aufsetzt, periodische Timerinterrupts zu Schedulingzwecken benutzen. In der Aufzeichnung eines überspringbaren Abschnitts können die durch durch einen Timer erzeugten Zustandsraumtransformationen enthalten sein, jedoch – gemäß des derzeitigen Entwurfs (vgl. Abschnitt 3.2.1) – nicht die Zustandsänderungen des Timers an sich. Wird diese Aufzeichnung dann für einen Sprung benutzt, werden die Zustandsraumtransformationen aus dem Ausgabeteil der Aufzeichnung in den Maschinenzustand übernommen. Das anschließende Vorspulen der Simulatorzeit kann jedoch den Timer erneut auslösen, da dessen Zustandsänderungen beim Sprung nicht berücksichtigt wurden. Aus Sicht der Maschine wurde der Interrupt damit zweimal ausgelöst und behandelt, was eine unerlaubte Verfälschung des Maschinenzustandes zur Folge hat. Auch dieses Problem wurde in der vorliegenden Arbeit ausgeklammert. Ergo können derzeit nur Zielprogramme beschleunigt werden, die keine Timerinterrupts verwenden.

3.3 Einbettung in Fail*

Der beschriebene Ansatz lässt sich in die Architektur von Fail* (2.3) einbetten und erfordert nur wenige Änderungen an bereits bestehenden Komponenten.

Das Experiment `generic-tracing` – welches der Durchführung und Aufzeichnung des Golden Run dient – kann erweitert werden, um die Unterteilung und Aufzeichnung von überspringbaren Abschnitten anzuregen. Anwender möchten jedoch möglicherweise eigene Experimente für das Tracing benutzen, weshalb diese Funktionalitäten gekapselt werden sollten. Für den Unterteilungsvorgang (Abschnitt 3.2.3) ist ein Modul zu implementieren, welches den Trace als Eingabe erhält und eine Liste von Abschnitten generiert. Der Aufruf dieses Moduls soll durch `generic-tracing` nach der Durchführung des Golden Run erfolgen. Für die anschließende Aufzeichnung der Abschnitte (Abschnitt 3.2.1) bietet sich die Entwicklung eines neuen Plugins an. `generic-tracing` soll um die Möglichkeit erweitert werden, einen zweiten Golden Run auszuführen und dieses Plugin parallel dazu zu aktiveren.

Analog zu dem FAIL*-Tool `import-trace` , welches den Trace in die Datenbank importiert, soll ein Tool geschaffen werden, welches die überspringbaren Abschnitte und ihre Aufzeichnungen importiert. Dies erleichtert den späteren Zugriff im Rahmen von Kampagnen. Das Datenbankschema muss dementsprechend um einige Tabellen erweitert werden, die bestehenden Tabellen [Sch16, 4.5.3.2] können dagegen unverändert bleiben. Der Import bietet zudem die Möglichkeit, die erhaltenen Daten noch einmal zu begutachten und ggf. zu filtern.

Für die Durchführung der Sprünge ist ein weiteres Plugin zu implementieren. Es soll jedes Mal, wenn ein bekannter Startpunkt erreicht wurde, die in Abschnitt 3.2.2 beschriebenen Schritte zum Finden einer anwendbaren Aufzeichnung durchlaufen und im Erfolgsfall einen Sprung durchführen. Ein Experiment, welches von überspringbaren Abschnitten profitieren möchte, muss dieses Plugin vor der Simulation des Zielprogramms aktivieren und die für das Experiment relevanten Listener bekannt geben. Während des Experiments sollte keine Interaktion zwischen dem Experiment und dem Plugin nötig sein. Dieses Vorgehen soll exemplarisch in das `generic-experiment` integriert werden, welches typische FAIL*-Anwendungsfälle abdeckt.

Alle genannten Komponenten sollten weitestgehend unabhängig vom Backend gestaltet werden (Anforderung der Generalität, Abschnitt 3.1). Es ist jedoch zu erwarten, dass manche Teile (wie bspw. das Setzen des Befehlszählers) Anpassungen des Backends erfordern. Die Menge dieser Anpassungen sind gering zu halten und idealerweise zu kapseln, um eine spätere Verbesserung der Implementierung zur Unterstützung weiterer Backends zu vereinfachen.

3.4 Zusammenfassung

Nach einer Problemanalyse und der Aufstellung eines Anforderungskatalogs in Abschnitt 3.1 wurde in Abschnitt 3.2 das Konzept der *überspringbaren Abschnitte* entworfen. Sie umgehen die langsame Ausführung von dynamischen Instruktionen im Simulator durch zusammengefasste Zustandsraumtransformationen. Es wurde dargelegt, welche Daten zu speichern sind, wie ein Sprung abläuft und wie eine geeignete Unterteilung des Golden Run in überspringbare Abschnitte zu bestimmen ist. Weiterhin wurden die Einschränkungen des aktuellen Entwurfs diskutiert. In Abschnitt 3.3 wurde erörtert, wie sich der Ansatz in die Infrastruktur von FAIL* einfügen lässt.

4 Implementierung

Dieses Kapitel beschreibt, wie Fail* verändert und erweitert wurde, um das in Kapitel 3 vorgestellte Konzept der *überspringbaren Abschnitte* umzusetzen.

4.1 Übersicht

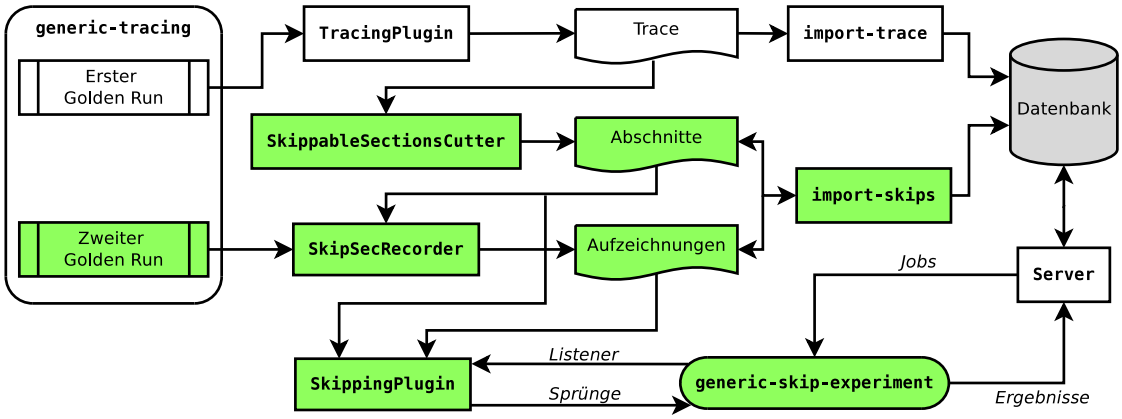


Abb. 4.1: Übersicht der implementierten Komponenten und ihrer Datenflüsse. Grün: Neue Komponenten. Weiß: Bestehende Fail*-Komponenten (vgl. Abb. 2.1).

Im Wesentlichen wurde Fail* um vier neue Module erweitert (Abb. 4.1), die in insgesamt rund 1400 C++-Quellcodezeilen implementiert wurden[1]. Zusätzlich wurden kleinere Modifikationen und Ergänzungen der zugrundeliegenden Fail*-Codebasis vorgenommen.

Das Modul `SkippableSectionsCutter` (Abschnitt 4.3.1) dient zur Unterteilung des aus dem Golden Run gewonnenen *Trace* in überspringbare Abschnitte. Das Plugin `SkipSecRecorder` fertigt für die überspringbaren Abschnitte während eines zweiten Golden Runs Aufzeichnungen an. Überspringbare Abschnitte und Aufzeichnungen werden in separaten Dateien gespeichert, die durch das Tool `import-skips`

[1] Effektive Quellcodezeilen (ohne Leerzeilen und Kommentare), ermittelt mit *cloc*: `http://cloc.sourceforge.net`.

(Abschnitt 4.3.2) gefiltert und in die Datenbank importiert werden. Dieses überschreibt die Dateien mit den gefilterten Daten, damit sie vom `SkippinPlugin` (Abschnitt 4.5) benutzt werden können. Dieses kann einem Fehlerinjektionsexperiment zugeschaltet werden um Sprünge durchzuführen. Zur Evaluation dieses Plugins wurde das `generic-skip-experiment` als Derivat des `generic-experiment` von FAIL* eingerichtet. Neben den überspringbaren Abschnitten benötigt das Plugin Informationen über die vom Experiment benutzten Listener.

4.2 Datenstrukturen

Die Datei `src/util/SkipSec.h` (dargestellt in Listing 4.1) definiert C++ Structs (Datenverbände) für die interne Speicherung von überspringbaren Abschnitten und ihren Aufzeichnungen. Für einen überspringbaren Abschnitt `SkipSec` werden in erster Linie eine Identifikationsnummer, der Startpunkt (eine statische Instruktionsadresse) und ein Vektor von Aufzeichnungen gespeichert. Eine Aufzeichnung `SkipSecRec` wiederum besteht aus einer Nummer (eindeutig für die jeweilige `SkipSec`), zwei Vektoren von Aufzeichnungseinträgen (je einen für den Ein- und Ausgabeteil), einem Vektor der enthaltenen (statischen Instruktionen), die nächste auszuführende Instruktion, sowie die Anzahl der zu überspringenden Instruktionen und Ticks. Ein Aufzeichnungseintrag `SkipSecRecEntry` kann über die boolesche Variable `is_reg` als Speicher oder Register markiert sein und speichert dementsprechend die FAIL*-interne ID eines Registers, bzw. die Adresse der Speicherzelle, sowie einen Wert. `SkipSec` und `SkipSecRec` und beinhalten darüber hinaus weitere Datenfelder, die vom `SkippingPlugin` (Abschnitt 4.5) benutzt werden.

Für überspringbare Abschnitte und ihre Aufzeichnungen wurden außerdem Google Protocol-Buffer [Goo08] Nachrichten definiert. Mit diesen Nachrichten können die Daten serialisiert und zwischen den implementierten Softwarekomponenten ausgetauscht werden.

4.3 Erweiterung des Tracings

Das von FAIL* bereitgestellte Experiment `generic-tracing` dient zur Durchführung des Golden Run und der Aufzeichnung des Trace. Es kann darüber hinaus den Maschinenzustand beim Erreichen eines definierten Startpunkts (z. B. der `main()` -Funktion) als Checkpoint sichern. Fehlerinjektionsexperimente können durch die Wiederherstellung dieses Zustands bspw. den zeitaufwendigen Boot-Vorgang umgehen. Auch können über das zuschaltbare Plugin `SerialOutput` die Ausgaben der seriellen Schnittstelle erfasst und in einer Datei gespeichert werden.

```
1  struct SkipSecRecEntry {
2    bool is_reg;      //! If true: This entry concerns a register
3    uint64_t id_or_addr;  //! If is_reg: Register ID, otherwise memory address
4    uint64_t value;      //! Value that was read or written
5  };
6
7  struct SkipSecRec {
8    uint32_t rec_num;        //! Starts at 0 for each SkipSec
9    fail::address_t next_instr;    //! Instruction that would follow the skip
10   uint64_t num_skipped_instr;    //! Number of skipped instructions
11   uint64_t num_skipped_ticks;    //! Number of skipped ticks (>=
       ↪ num_skipped_instr)
12   std::vector<SkipSecRecEntry> in;  //! All entries of the input set
13   std::vector<SkipSecRecEntry> out;  //! All entries of the output set
14   std::vector<fail::address_t> skip_instr;  //! Static addresses of
       ↪ instructions that will be skipped
15
16   // Used by SkippingPlugin
17   std::vector<fail::MemAccessListener*> blocking_mal;   //! List of
       ↪ MemAccessListeners blocking this record
18   uint32_t occurences;                //! How often this record appeared in
       ↪ the golden run
19   uint32_t use_counter;                //! If 0, the record is "used up"
20 };
21
22 struct SkipSec {
23   uint64_t id;            //! Unique identifier
24   fail::address_t start;        //! Starting instruction.
25   uint64_t gr_cut_length;        //! Cutting length in golden run (record for
       ↪ this many instructions)
26   std::vector<SkipSecRec> records;  //! All records for this skippable section
27
28   // Used by SkippingPlugin
29   uint32_t last_matching_rec;      //! Index of the record that matched last,
       ↪ the successor will be tested first
30   fail::BPSingleListener* l;      //! Listener to the start of this skipsec
31   uint32_t recs_to_use_left;      //! How many of this skipsec's records are
       ↪ not used up yet. Deactivate skipsec if this reaches 0.
32   uint32_t use_counter;        //! Deactivate skipsec if this reaches 0. Can
       ↪ help reduce runtime of experiments stuck in endless loops.
33 };
```

Lst. 4.1: `SkipSec.h`

Das Experiment wurde erweitert, um – nach einer Aktivierung durch Kommandozeilenschalter – eine Unterteilung des Trace in überspringbare Abschnitte anzuregen und für diese während eines zweiten Golden Runs Aufzeichnungen anzufertigen.

4.3.1 Unterteilung eines Trace in überspringbare Abschnitte

Die Klasse `SkippableSectionsCutter` dient zur Unterteilung der im Golden Run ausgeführten Instruktionen in überspringbare Abschnitte. Den Kern bildet die Funktion `cut()`. Bevor diese Funktion benutzt werden kann, ist die Übergabe von zwei Zeigern auf Protobuf-Dateiströmen erforderlich: Einen zum Einlesen des Trace und einen für die Serialisierung der erzeugten überspringbaren Abschnitte. Außerdem ist das Zielprogramm in Form einer ELF-Datei zu übergeben.

Die Funktion `cut()` beginnt mit einer Disassemblierung der ELF-Datei, damit später Ein- und Ausgabeinstruktionen (vgl. Abschnitt 3.2.4) von den überspringbaren Abschnitten exkludiert werden können. Anschließend wird der Trace eingelesen, wobei die Vorkommen von statischen Instruktionen zur Anfertigung eines *Instruktionshistogramms* (Abb. 4.2) mitgezählt werden. Dieses wird bei der Aufstellung der *Kostenfunktion* benötigt, welche beim Finden eines günstigen Endpunktes für einen überspringbaren Abschnitt eingesetzt wird. Soll nämlich der Endpunkt eines überspringbaren Abschnittes der Startpunkt eines nachfolgenden überspringbaren sein, um den zwischenzeitlichen Kontrollflusswechsel zum Simulator zu vermeiden, so sollte diese statische Instruktion im Zielprogramm möglichst selten vorkommen. Wird diese Instruktion während eines Fehlerinjektionsexperiments erreicht, löst das die in Abschnitt 3.2.2 erörterten Schritte zum Finden einer passenden Aufzeichnung aus, welche ungewollten Overhead verursachen wenn keine solche Aufzeichnung gefunden werden kann.

Ein anderer Bestandteil der Kostenfunktion ist der *Speicherzugriffsstrace*[2]. Dies ist eine Funktion über die dynamischen Instruktionen, welche approximiert, wie viel Speicher (in Byte) das Zielprogramm zu jedem Zeitpunkt nutzt. Schreiboperationen auf zuvor ungenutzten Speicherzellen erhöhen den Wert dieser Funktion, Leseoperationen verringern ihn. Die Start- und Endpunkte von überspringbaren Abschnitten sollten nahe der Minima dieser Funktion liegen, damit die Aufzeichnungen geringe Größen aufweisen, was sowohl für das *Matching* als auch die Durchführung des Sprungs günstig ist (Abschnitt 3.2.3).

[2] Instruktionshistogramm und Speichernutzungstrace können zu Visualisierungszwecken von der Klasse in CSV-Dateien exportiert werden.

Die Unterteilung des Trace in überspringbare Abschnitte erfolgt schließlich gemäß des in 3.2.3 entworfenen Algorithmus, wobei zusätzlich sichergestellt wird, dass die Abschnitte keine Ein- oder Ausgabeinstruktionen enthalten (Abschnitt 3.2.4).

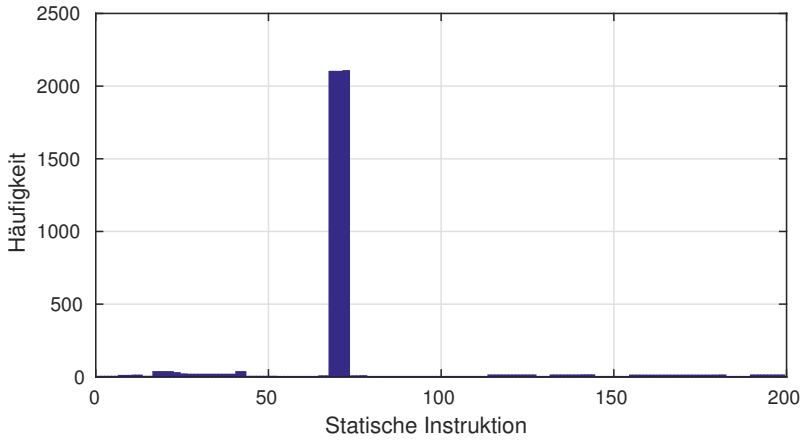


Abb. 4.2: Instruktionshistogramm eines einfachen Zielprogramms. Die statischen Instruktionen um 68 bis 73 liegen in einer Schleife und sind ungünstige Startpunkte für überspringbare Abschnitte, da sie überproportional häufig auftreten und im FI-Experiment viele *Mismatches* zur Folge haben können.

Die Mindestlänge für die überspringbaren Abschnitte kann über die Funktion `setMinCutLength()` gesetzt werden. Anwender sind angehalten zu vermessen, wie viele Instruktionen das Backend in einer gegebenen Zeitspanne durchführen kann und wie lange ein Sprung benötigt, um abzuschätzen, was ein geeigneter Wert für die Mindestlänge sein könnte. Dabei sind auch die Länge des Zielprogramms, die Häufigkeit von Ein- und Ausgabeinstruktionen sowie die Speichernutzung zu berücksichtigen.

Zum Abschluss werden die generierten überspringbaren Abschnitte – d. h. ihre Startpunkte und die Anzahl der dynamischen Instruktionen bis zum jeweiligen Endpunkt – in die Protobuf-Datei geschrieben. Zur Ausgabe und Kontrolle des Inhalts kann das in `fail/tools/dump-skips` implementierte Python-Skript verwendet werden.

4.3.2 Anfertigung von Aufzeichnungen

Nach der Unterteilung kann `generic-tracing` einen zweiten Golden Run durchführen. Für diesen wird das neue Plugin `SkipSecRecorder` aktiviert, um für jeden überspringbaren Abschnitt eine Aufzeichnung anzufertigen. Das Plugin registriert

zunächst Listener für jede Instruktion und jeden Speicherzugriff, bevor die Simulation gestartet wird. Bei jeder Instruktion wird geprüft, ob der Startpunkt eines überspringbaren Abschnittes erreicht wurde. In diesem Fall wird die Funktion `openRecord()` aufgerufen, um eine Protobuf-Nachricht vorzubereiten und die interne Uhrzeit des Simulators sowie alle Registerinhalte mittels `dumpRegs()` zu sichern. Nun wird wie folgt verfahren:

1. Jede dynamische Instruktion erhöht den Zähler `num_skipped_instr` der Aufzeichnung. Die statische Instruktionsadresse wird, falls noch nicht erfasst, der Menge `skip_instr` hinzugefügt.

2. Jeder lesende Speicherzugriff mit zugehörigem Wert wird gespeichert, sofern es sich um den ersten Lesezugriff innerhalb der aktuell betrachteten Aufzeichnung auf diese Speicheradresse handelt. Analog werden auch die Schreibzugriffe aufgezeichnet, die Werte ersetzen dabei ggf. ihre Vorgänger.

3. Ist die Anzahl der zu überspringenden Instruktionen erreicht, schließt die Funktion `closeRecord()` die Aufzeichnung ab. Alle Speicherzugriffe werden in die Protobuf-Nachricht kopiert, ebenso erneut alle derzeitigen Registerinhalte sowie die Anzahl der übersprungenen Instruktionen `num_skipped_instr` und die Zeitdifferenz seit Beginn der Aufzeichnung[3] `num_skipped_ticks`. Nun kann eine neue Aufzeichnung beginnen, sofern der Golden Run noch nicht am Ende angelangt ist.

Die Aufzeichnungen werden alle in einer zu den Abschnitten separaten Protobuf-Datei gespeichert. Das o. g. Tool `dump-skips` kann auch diese Datei auslesen, um die Aufzeichnungen darzustellen.

4.4 Datenbank-Import und Filterung

Nachdem die überspringbaren Abschnitte definiert und aufgezeichnet wurden, sind sie in die Datenbank zu importieren. Gleichzeitig findet hierbei eine Filterung statt. So werden lediglich diejenigen Register als Aufzeichnungseinträge importiert, die für den Abschnitt auch relevant sind, während die ursprüngliche Aufzeichnung noch alle Register und ihre Werte enthielt. Auch ist es möglich, dass zwei Aufzeichnungen für den selben überspringbaren Abschnitt identisch sind, sodass eine Kopie entfernt werden kann. Die Filterung erst zu diesem Zeitpunkt und nicht

[3] Die meisten Instruktionen benötigen einen einzelnen CPU-Takt (Tick), es gibt aber auch Instruktionen (wie bspw. `hlt`) die mehrere Ticks in Anspruch nehmen. Diese Unterscheidung ist u. a. dann wichtig, wenn `TimerListener` (4.5.2.3) berücksichtigt werden sollen.

schon während der Tracingphase vorzunehmen, vereinfacht die Implementierung und Lesbarkeit des Codes. Zur Aufnahme der Daten wurde das Datenbankschema von FAIL* um mehrere Tabellen erweitert. Abbildung 4.3 zeigt die Attribute und Verbindungen dieser Tabellen.

SKIPSEC: Speichert die überspringbaren Abschnitte, d. h. insbesondere ihre Startpunkte.

SKIPSECREC: Beinhaltet die Basisinformationen von Aufzeichnungen, wie bspw. die Anzahl der zu überspringenden Instruktionen und die Anzahl der Vorkommen dieser Aufzeichnung im Golden Run.

SKIPSECREC_INSTR: Ordnet Aufzeichnungen die übersprungenen *statischen* Instruktionen zu. Sie werden benötigt, um Sprünge zu blockieren, die mit einem `BPListener` in Konflikt stehen (Abschnitt 4.5.2.2).

SKIPSECREC_IN: Repräsentiert die Eingabeteile der Aufzeichnungen. Speichert für jede Aufzeichnung, welche Teile des Maschinenzustandes (Register, Speicher) mit welchem Wert gelesen wurden.

SKIPSECREC_OUT: Analog zu SKIPSECREC_IN wird hier der Ausgabeteil jeder Aufzeichnung gespeichert, d. h. welche Werte durch den Abschnitt in Register und Speicher geschrieben werden.

Diese Tabellen werden durch das Tool `import-skips` gefüllt. Es liest dazu die zwei Protobuf-Dateien aus, die zuvor in der Tracingphase angelegt wurden. Nach dem Import werden die gefilterten und sortierten Daten zurück in die Protobuf-Dateien exportiert. Sie dienen dann als Datenquelle für die Clients (Abschnitt 4.5.1).

4.5 Plugin zum Durchführen der Sprünge

Das Herzstück der gesamten Implementierung ist das `SkippingPlugin`. Es kann parallel zum eigentlichen Fehlerinjektionsexperiment laufen und versucht die Ausführungszeit – wann immer möglich – mittels Sprüngen zu verkürzen. Die Vorgehensweise entspricht dem in Abschnitt 3.2.2 vorgestellten Konzept.

Nach dem Einlesen der Daten (Abschnitt 4.5.1) wird für jeden überspringbaren Abschnitt ein `BPSingleListener` auf den Startpunkt registriert. Löst ein solcher Listener aus, wird geprüft ob ein anwendbarer Abschnitt vorliegt. Ein Abschnitt ist

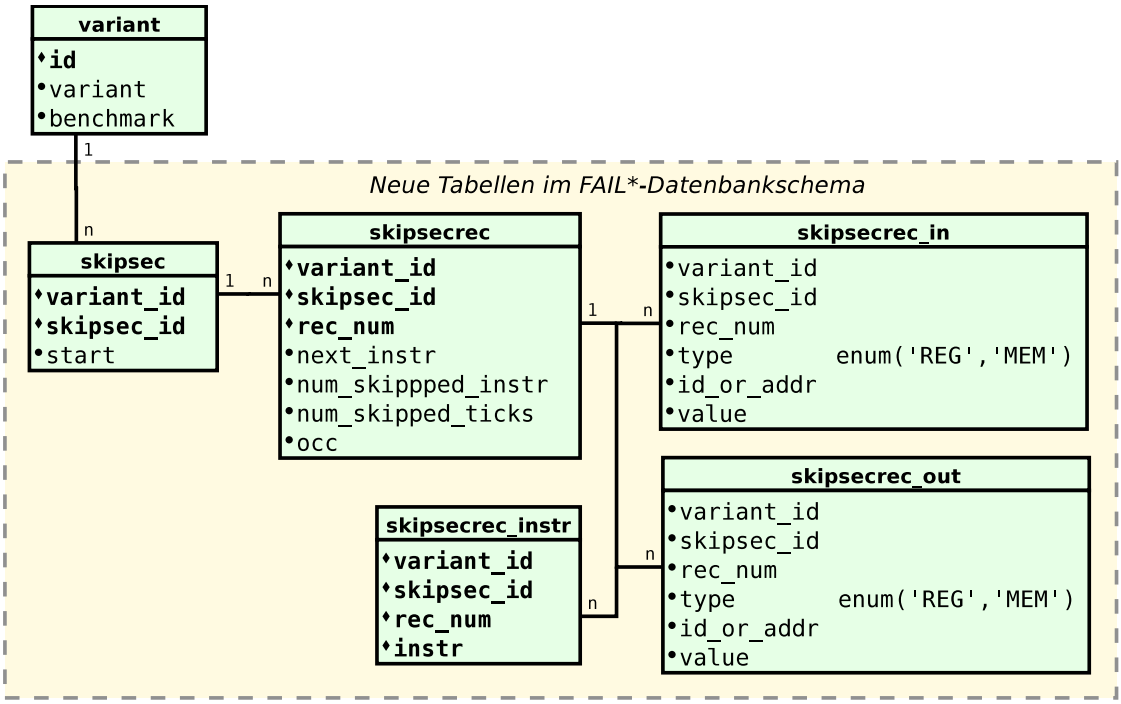


Abb. 4.3: Erweiterung des Datenbankschemas von Fail*.
Legende:
♦ = Primärschlüsselspalte,
• = Wert verpflichtend (nicht null).

dann anwendbar, wenn kein anderer Listener ihn blockiert (Abschnitt 4.5.2) und der Maschinenzustand zum Eingabeteil der Aufzeichnung passt (Match). Konnte eine passende Aufzeichnung gefunden werden, wird die Zustandsraumtransformation durchgeführt (Abschnitt 4.5.3). Andernfalls wird die Simulation normal fortgesetzt.

4.5.1 Einlesen der Daten

Fail* sieht vor, dass Clients via Protobuf-Nachrichten über TCP mit dem Server kommunizieren, um *Jobs* (Parameter auszuführender Experimente) zu erhalten. Langfristig könnte die Übertragung überspringbarer Abschnitte und ihrer Aufzeichnungen ebenfalls in diesen Kanal integriert werden. In der vorliegenden Implementierung werden hierfür jedoch die beiden Protobuf-Dateien verwendet, deren Daten vom `import-skips` Tool gelesen, gefiltert und zurück in die Dateien geschrieben werden (Abschnitt 4.4). Das Einlesen dieser Dateien erfolgt über die Funktion `readSkipSecData()`, welche Zeiger zu den beiden Dateien übergeben bekommt. Das Einlesen erfolgt – sofern die Dateinamen als Kommandozeilenparameter über-

geben wurden – einmalig beim Start des Clients. Beim `generic-skip-experiment` wurde der Aufruf in die Funktion `cb_start_experiment()` integriert (Listing 4.2).

```
1  if (cmd[SKIPSEC_FILE]) {
2    if (cmd[SKIPSECREC_FILE]) {
3      enabled_skipping = true;
4      skipper = new SkippingPlugin();
5      m_log << "Enabled skipping code sections of the target program to reduce
         ↪  runtime." << endl;
6
7      skipsec_file = std::string(cmd[SKIPSEC_FILE].first()->arg);
8      skipsecrec_file = std::string(cmd[SKIPSECREC_FILE].first()->arg);
9      igzstream if_ss(skipsec_file.c_str());
10     igzstream if_ssr(skipsecrec_file.c_str());
11     skipper->readSkipSecData(&if_ss, &if_ssr);
12   } else {
13     m_log << "If you want to use skippable sections, set --skipsecrec-file
         ↪  <file> as well." << endl;
14   }
15 }
```

Lst. 4.2: Initialisierung des `SkippingPlugin` im `generic-skip-experiment`.

4.5.2 Blockierung durch Experiment-Listener

Ein Experiment kann unterschiedliche Listener registrieren, um die Simulation zu steuern und unterschiedliche Ergebnisse zu erfassen. Da kein Sprung das Auslösen eines solchen Listeners verhindern darf (Abschnitt 3.2.2), wurde im Plugin die Schnittstelle `blockSkippingForListener()` geschaffen. Diese akzeptiert einen Zeiger zu Listenern unterschiedlicher Typen (für Speicherzugriffe, Erreichen bestimmter Instruktionen, Timer, etc.). Wie genau Sprünge für den jeweiligen Typus blockiert werden, ist in den nachfolgenden Textabschnitten beschrieben. Bei Bedarf kann eine Blockierung durch die komplementäre Schnittstelle `removeListenerBlockage()` wieder aufgehoben werden.

4.5.2.1 MemAccessListener

Ob eine Aufzeichnung durch einen Listener auf einen Speicherzugriff blockiert ist, lässt sich durch einen Abgleich des beobachteten Speicherbereichs mit den Aufzeichnungseinträgen feststellen (Abb. 4.4). Soll der Listener beim Lesen eines be-

stimmten Speicherbereichs auslösen, muss getestet werden, ob dieser Speicherbereich im Eingabeteil der Aufzeichnung vorkommt. Analog muss bei Listenern auf Schreibvorgänge geprüft werden, ob ein entsprechender Eintrag im Ausgabeteile der Aufzeichnung existiert.

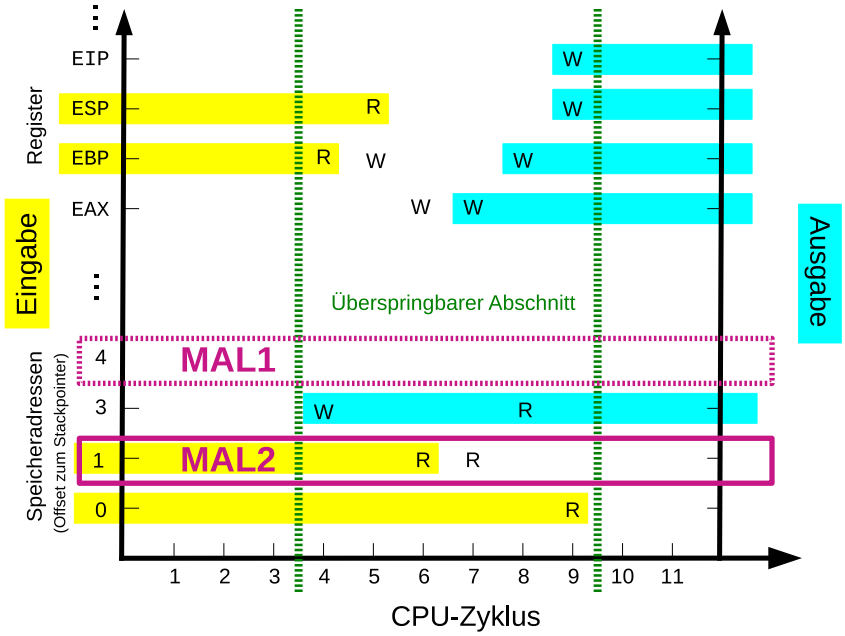


Abb. 4.4: Illustration einer Blockierung durch einen `MemAccessListener` . Die Aufzeichnung (grün) wird durch den Listener „MAL2“ blockiert, da die beobachtete Speicheraddresse (1) im Eingabeteil der Aufzeichnung (gelb) enthalten ist (vgl. Abb. 3.3). MAL1 dagegen blockiert den Sprung nicht.

Findet sich eine solche Übereinstimmung, markiert das `SkippingPlugin` die Aufzeichnung als blockiert, indem es die Adresse des Listeners in den dafür vorgesehenen Vektor `blocking_mal` einfügt. Aufzeichnungen dürfen nur dann angewandt werden, wenn diese Liste vollständig leer ist.

4.5.2.2 BPListener

Breakpoint-Listener horchen auf das Erreichen statischer Instruktionen. Im Normalfall wird ein `BPSingleListener` auf eine bestimmte statische Instruktion eingerichtet. Ist diese Instruktion in dem Vektor `skip_instr` enthalten, welcher die Menge der statischen Adressen der zu überspringenden Instruktionen enthält, darf die Aufzeichnung nicht angewandt werden (Abb. 4.5). Analog kann mit einem

`BPRangeListener` verfahren werden, der auf das Erreichen einer Instruktion innerhalb eines festgelegten Bereichs horcht.

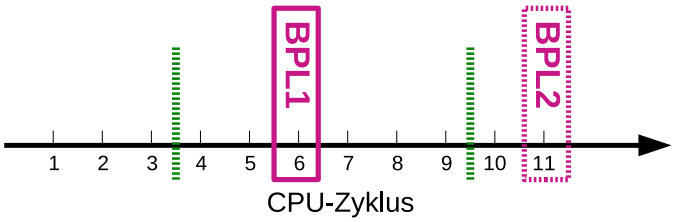


Abb. 4.5: Illustration einer Blockierung durch einen `BPSingleListener` . Die Aufzeichnung (grün) wird durch den Listener „BPL1“ blockiert, da der Breakpoint (Instruktion 6) zu den zu überspringenden Instruktionen gehört. BPL2 dagegen blockiert den Sprung nicht.

In vielen Experimenten wird zur Fehlerinjektion jedoch ein `BPSingleListener` auf beliebige Instruktionen (`ANY_ADDR`) eingerichtet. Dieser dekrementiert bei jeder Änderung des Befehlszählers einen internen Zähler und löst erst dann aus, wenn dieser Zähler den Wert 0 erreicht. Vor der Anwendung einer Aufzeichnung muss also für diese Listener die Anzahl der zu überspringenden Instruktionen `num_skipped_instr` mit der Größe dieses internen Zählers (die Anzahl verbleibender Ticks bis zur Auslösung) verglichen werden. Ist der Zähler kleiner als die Anzahl der zu überspringenden Instruktionen, darf die Aufzeichnung nicht benutzt werden.

4.5.2.3 TimerListener

`TimerListener` lösen dann aus, wenn ein zugeordneter *Timer* auslöst. Sie werden von Fehlerinjektionsexperimenten bspw. dazu benutzt, das Ergebnis *Timeout* (Ausführung des Programms überschreitet ein festgelegtes Zeitlimit) zu erkennen. Um zu prüfen ob ein Sprung durch einen `TimerListener` blockiert wird, muss der Auslösungszeitpunkt des Listeners mit der Simulatorzeit nach dem potentiellen Sprung (Summe aus aktueller Simulatorzeit und der Anzahl zu überspringender Ticks `num_skipped_ticks`) verglichen werden. Liegt der Auslösungszeitpunkt davor, ist die Aufzeichnung blockiert.

Um den Auslösungszeitpunkt zu erhalten, wurde die Bochs-Variante der FAIL*-Klasse `TimerListener` um die Funktion `getTicksToFire()` erweitert. Diese wiederum verwendet eine gleichnamige Funktion, die Bochs im Modul `pc_system` hinzugefügt wurde.

4.5.3 Zustandsraumtransformation

Wurde eine Aufzeichnung als anwendbar erkannt, transferiert das `SkippingPlugin` den Ausgabeteil der Aufzeichnung in den Maschinenzustand. Im Anschluss werden die Zählerwerte aller `BPSingleListener` , die auf beliebige Adressen (`ANY_ADDR`) registriert wurden, um `num_skipped_instr` reduziert. Abschließend wird die Simulatorzeit um `num_skipped_ticks` vorgespult. Der Aufruf verwendet dazu die Funktion `tickn()` des Bochs-Moduls `pc_system` . Sie dekrementiert auch den aktuellen *Countdown*, was ein Auslösen der anhänglichen Timer zur Folge haben kann (vgl. Abschnitt 3.2.4). Über das Setzen der Flag `skip_request` kann dem Backend signalisiert werden, dass ein Sprung ausgeführt wurde.

Nach dem Sprung kehrt das `SkippingPlugin` zu dem Beginn einer Schleife zurück, wo geprüft wird, ob die neue Instruktion den Startpunkt eines weiteren überspringbaren Abschnitts markiert. Ist dies der Fall, kann erneut nach einer passenden Aufzeichnung gesucht und idealerweise ein weiterer Sprung durchgeführt werden. Ist die neue Instruktion dagegen kein bekannter Startpunkt, oder konnte keine anwendbare Aufzeichnung gefunden werden, muss die Simulation normal fortgesetzt (d. h. die Kontrolle zurück an den Simulator übergeben) werden. Unmittelbar bevor die Simulation fortgesetzt werden soll, kann es erforderlich sein, dass nach einem Sprung die nächste auszuführende Instruktion (neuer Inhalt des Registers `IP` bzw. `EIP`) im Speicher gefunden und dekodiert wird. Dieser Vorgang ist für Bochs in der Funktion `redecodeCurrentInstruction()` implementiert worden (Listing 4.3). Wechselt die Kontrolle zu Bochs, wird – falls `skip_request` gesetzt wurde – vor dem Durchlaufen des nächsten CPU-Zyklus die Länge der auszuführenden Instruktion neu bestimmt. Darüber hinaus soll nach dem CPU-Zyklus ein neuer Eintrag aus dem – durch den Sprung invalidierten – internen Instruktionspuffer[4] geladen werden.

Als Optimierungsmaßnahme kann das `SkippingPlugin` überspringbare Abschnitte und Aufzeichnungen vorzeitig deaktivieren. Aufzeichnungen werden deaktiviert, wenn sie im Experiment so oft angewandt wurden, wie sie im Golden Run während der Aufzeichnung auftraten (Spalte `occ` der Datenbanktabelle SKIPSECREC). Aufgrund des injizierten Fehlers können weitere Anwendungen zwar möglich sein, dies ist jedoch eher unwahrscheinlich. Ein Verzicht auf weitere Prüfungen reduziert den Overhead. Analog werden überspringbare Abschnitte dann deaktiviert, wenn alle zugeordneten Aufzeichnungen deaktiviert wurden oder zu oft keine anwendbare Aufzeichnung gefunden wurden, obwohl noch aktive existieren. Dies vermeidet den

[4]Die Verwendung des Instruktionspuffers ist optional und kann beim Kompilieren der Clients durch die Bochs-Option `--disable-trace-cache` deaktiviert werden.

```
1  inline void SkippingPlugin::redecodeCurrentInstruction(fail::address_t&
   ↪  next_instr) {
2    BX_CPU_C *cpu_context = fail::simulator.getCPUContext();
3    cpu_context->invalidate_prefetch_q();
4    cpu_context->iCache.flushICacheEntries();
5
6    bxInstruction_c *currInstr = fail::simulator.getCurrentInstruction();
7
8    Bit32u eipBiased = next_instr + cpu_context->eipPageBias;
9    Bit8u  instr_plain[32];
10   mm->getBytes(next_instr, 32, instr_plain);   // mm: MemoryManager
11
12   unsigned remainingInPage = cpu_context->eipPageWindowSize - eipBiased;
13   int ret;
14   #if BX_SUPPORT_X86_64
15   if (cpu_context->cpu_mode == BX_MODE_LONG_64)
16     ret = cpu_context->fetchDecode64(instr_plain, currInstr, remainingInPage);
17   else
18   #endif
19     ret = cpu_context->fetchDecode32(instr_plain, currInstr, remainingInPage);
20   if (ret < 0) {
21     // handle instrumentation callback inside boundaryFetch
22     cpu_context->boundaryFetch(instr_plain, remainingInPage, currInstr);
23   }
24 }
```

Lst. 4.3: Redekodierung der Instruktion nach Setzen des Befehlszählers.

Overhead insbesondere in Szenarien, wo das Zielprogramm in einer Endlosschleife gefangen wird, welche einen oder mehrere Startpunkte enthält.

4.5.4 Sammeln statistischer Informationen

Während der Ausführung des Plugins werden verschiedene Zähler verwendet um zu erfassen, wie viele Sprünge durchgeführt werden konnten, wie oft eine Prüfung fehlschlug, wie oft keine passende Aufzeichnung gefunden werden konnte und wie oft Blockierungen durch Listener vorlagen. Diese Informationen werden im Datenverbund `stats` gespeichert und nach Ende einer Simulation in die Ergebnistabelle in der Datenbank übertragen. Sie dienen zur Evaluation der Effizienz des Plugins und können beispielsweise Anhaltspunkte darüber liefern, ob die Abschnittsunterteilung (Abschnitt 4.3.1) geeignet vorgenommen wurde.

4.6 Einschränkungen

Fail* unterstützt unterschiedliche Backends (2.3). Der Großteil der Implementierung ist unabhängig vom verwendeten Backend. Wenige Teile mussten jedoch Backend-spezifisch implementiert werden (siehe Abschnitte 4.5.2.3 und 4.5.3). Die Wahl des zu verwendenden Backends fiel hierbei auf Bochs [Law96]. Prinzipiell könnten die entsprechenden Codeabschnitte auch für die anderen von Fail* unterstützten Backends portiert werden. Da der Fokus dieser Masterarbeit jedoch auf einem grundsätzlichen Nachweis der Nützlichkeit des vorgestellten Verfahrens lag (*Proof of Concept*) wurde auf diese Anpassungen zu Gunsten einer Verfeinerung des Ansatzes verzichtet.

Im aktuellen Stand ist die Implementierung ungeeignet für Zielprogramme, die Gleitkommaarithmetik benutzen. Gleitkommazahlen werden üblicherweise mit 32 (*float*) oder 64 Bit (*double*) repräsentiert. Die acht als Stack organisierten Register `ST0` bis `ST7` der *Floating Point Unit* (FPU) weisen jedoch eine Breite von 80 Bit auf. Die zusätzlichen Bits dienen der Minimierung von Rundungsfehlern. Die von der Implementierung verwendeten Fail*-Funktionen zum Lesen und Setzen von Registerinhalten sind auf ein Maximum von 64 Bit ausgelegt. Die dafür nötigen Änderungen können in einer weiterführenden Arbeit vorgenommen werden.

4.7 Zusammenfassung

Die als Erweiterung zu Fail* implementierten Komponenten wurden übersichtlich in Abschnitt 4.1 vorgestellt. Die grundlegenden Datenstrukturen zum Speichern und Übertragen der überspringbaren Abschnitte und ihrer Aufzeichnungen wurden in Abschnitt 4.2 erläutert. Abschnitt 4.3 die Unterteilung eines Trace in überspringbare Abschnitte durch die Klasse `SkippableSectionsCutter` sowie `SkipSecRecorder`, ein Plugin zur Aufzeichnung dieser Abschnitte. Wie das Datenbankschema von Fail* erweitert wurde und wie die in der Tracingphase gewonnenen Daten in die Datenbank importiert werden, erläuterte Abschnitt 4.4. Das `SkippingPlugin` zum Durchführen der Sprünge präsentierte Abschnitt 4.5. Es wurde dargelegt, wie das `SkippingPlugin` in Fehlerinjektionsexperimente einzubinden ist, wie es funktioniert und welche Optimierungsmaßnahmen getroffen wurden. Abschnitt 4.6 legte dar, warum eine Einschränkung auf das Backend Bochs vorgenommen wurde und warum mit dem aktuellen Stand keine Zielprogramme mit Gleitkommaarithmetik beschleunigt werden können.

5 Evaluation

In diesem Kapitel soll die Nützlichkeit der in Kapitel 4 vorgestellten Implementierung untersucht werden. Nach einer Festlegung der Evaluationskriterien und der Vorgehensweise sollen Fehlerinjektionskampagnen für ausgewählte Zielprogramme mit dem Ansatz der überspringbaren Abschnitte beschleunigt und mit Kampagnen ohne Sprünge verglichen werden.

5.1 Kriterien

Das übergeordnete Ziel dieser Arbeit ist die Verkürzung der Laufzeiten von Fehlerinjektionskampagnen. In Abschnitt 3.1 wurde dieses Ziel zu vier Anforderungen konkretisiert: Effektivität, Ergebnistreue, Generalität und Nutzerfreundlichkeit.

Die ersten beiden Anforderungen, Effektivität und Ergebnistreue, können über die vergleichende Durchführung von Kampagnen evaluiert werden. Für ein gewähltes Zielprogramm dient eine Kampagne ohne Sprünge mit hinreichend vielen Experimenten als Vergleichsgrundlage. Erzeugt eine Kopie dieser Kampagne, für die Sprünge aktiviert wurden, die gleichen Ergebnisse, ist zumindest für dieses Zielprogramm die Anforderung der Ergebnistreue erfüllt. Einen weiteren Ansatz zur Untersuchung der Ergebnistreue wird 5.3 zeigen. Hat die Kampagne mit Sprüngen eine kürzere Laufzeit als ihr Gegenstücke ohne Sprünge, kann die Implementierung für das gewählte Zielprogramm als effektiv bezeichnet werden.

Die Erweiterung soll für möglichst viele unterschiedliche Zielprogramme anwendbar sein (Generalität). Es ist von daher – unter Berücksichtigung der in 3.2.4 und 4.6 aufgezählten Einschränkungen – eine Auswahl von *Benchmarks* zu treffen, d. h. Zielprogrammen, die typische Arbeitslasten von Fail* repräsentieren. Die gewählten Benchmarks werden in 5.2 vorgestellt. Sind Ergebnistreue und Effektivität für all diese Benchmarks zu beobachten, ist dies ein Indikator dafür, dass sie auch bei anderen Zielprogrammen erfüllt sind. Ein allgemeiner Nachweis ist jedoch aufgrund der Komplexität des Gesamtsystems – bestehend aus Hardware, Fail*, Backend und Zielprogramm – nicht möglich.

Die Nutzerfreundlichkeit kann qualitativ untersucht werden. Auf Grundlage von Kapitel 4 ist zu diskutieren, welche Schritte Anwender durchführen müssen, um überspringbare Abschnitte für die Beschleunigung ihrer Fehlerinjektionskampa-

gnen nutzen zu können. Von Relevanz sind dabei insbesondere die erforderlichen Anpassungen an bestehende Fehlerinjektionsexperimente.

5.2 Verwendete Benchmarks

Zur Evaluation der Implementierung wurde *MiBench* [Gut+01] von Guthaus et al. eingesetzt. Diese ist eine Sammlung von Benchmarks für eingebettete Systeme, die sich – angelehnt an die unterschiedlichen Anwendungsfelder eingebetteter Systeme – in sechs *Suiten* aufteilen: „Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications“. In dieser Masterarbeit wurde eine Auswahl von Benchmarks aus den Suiten *automotive*, *network* und *security* verwendet. Gewählt wurden diejenigen Benchmarks, die bereits in [Sch16] für die Evaluation von Fail* aufbereitet wurden (bspw. durch das Hinzufügen des Symbols `ECOS_BENCHMARK_FINISHED`, damit Fail* das Ende des Zielprogramms automatisch erkennen kann) und die beschriebenen Einschränkungen der Methode und der Implementierung (kein MMIO, keine Gleitkommaarithmetik) erfüllten. Tabelle 5.1 zeigt eine Auflistung und kurze Beschreibung der gewählten Benchmarks. Genauere Beschreibungen sind [Gut+01] zu entnehmen. In MiBench sind für jeden Benchmark zwei unterschiedliche Eingaben enthalten, je eine kleine und eine große.

Suite	Benchmark	Beschreibung
automotive	bitcount	Zählen von 1-Bits in einem Array mit verschiedenen Methoden.
	qsort	Sortierung einer Wörterliste mit *QuickSort*.
network	dijkstra	Finden von kürzesten Pfaden in einem Graphen.
security	blowfish	Ver- und Entschlüsselung mit dem Blowfish-Algorithmus.
	rijndael	Ver- und Entschlüsselung mit dem *Advanced Encryption Standard* (AES).

Tab. 5.1: Beschreibung der verwendeten Benchmarks.

Als zugrundeliegendes Betriebssystem wurde *eCos* (Abkürzung für engl. *embedded configurable operating system*) [Mas03] gewählt. Dies ist ein freies, konfigurierbares Echtzeitbetriebssystem für eingebettete Systeme. Die Benchmark nutzten

die eCos-Implementierung der C-Standardbibliothek u. a. für Speicherverwaltung und formatierte Ausgaben. Da – wie 4.6 erläuterte – Timerinterrupts die korrekte Durchführung von Sprüngen stören können, wurden zu Beginn jedes Benchmarks *alle* Interrupts durch die Zeile `__asm__("cli")` deaktiviert. Außerdem wurden Ausgaben der seriellen Schnittstelle beschleunigt, indem das Warten auf das Freiwerden des entsprechenden Puffers aus eCos entfernt wurde. Der Benchmark *dijkstra* wurde zudem so modifiziert, dass Ausgaben nicht an die serielle Schnittstelle, sondern den Port `E9` geschickt wurden, welcher von Bochs für Debugzwecke als Ausgabekanal angeboten wird. Damit konnte für diesen Benchmark eine weitere Laufzeitreduktion erreicht werden.

5.3 Beschleunigung eines Golden Run

Das Experiment `goldenrun-skip` ist, genau wie `generic-tracing`, entkoppelt von einer Fehlerinjektionskampagne zu benutzen. Im Rahmen der Entwicklung wurde es eingesetzt, um die prinzipielle Funktionalität des `SkippingPlugin` zu testen. Es führt einen gewöhnlichen Golden Run aus und misst dabei seine Laufzeit. Im Anschluss wird analog ein zweiter Golden Run ausgeführt, bei dem das `SkippingPlugin` aktiviert wurde. Am Ende jedes Golden Run werden die vom Zielprogramm erzeugten Ausgaben sowie der komplette Maschinenzustand in unterschiedlichen Verzeichnissen gespeichert. Listing 5.1 zeigt die Ausgaben eines solchen Experiments. Die Zeitmessungen geben Aufschluss darüber, wie viel Laufzeitverbesserung im Idealfall durch die Sprünge erreicht werden kann (Abb. 5.1), wenn aufgrund eines fehlerfreien Maschinenzustands und ohne Blockierungen durch Listener jede Aufzeichnung als anwendbar erkannt werden muss. Die gespeicherten Maschinenzustände und Ausgaben können mittels des Unix-Tools *diff* verglichen werden. Sind die Dateien identisch, ist dies ein Indikator dafür, dass für das gewählte Zielprogramm die Anforderung der Ergebnistreue auch im Rahmen einer Fehlerinjektionskampagne gewahrt ist. Bei allen getesteten Benchmarks war dies der Fall.

5.4 Empirische Untersuchung von Effektivität und Ergebnistreue

Gemäß der in Abschnitt 5.1 beschriebenen Vorgehensweise wurde für die Evaluation der Anforderungen Effektivität und Ergebnistreue eine Reihe von vergleichenden Kampagnen durchgeführt: Für jeden Benchmark jeweils eine Kampagne ohne und eine mit Sprüngen.

5.4.1 Aufbau

Zur Ausführung der Clients wurden 15 virtuelle Maschinen (VMs) in einem Computercluster der Universität verwendet. Acht der VMs verfügten über je 48 CPU-Kerne und 96 GB RAM, die restlichen sieben VMs verfügten über je 64 Kerne und 128 GB RAM. Die Taktrate der Kerne betrug 2,1 GHz. Pro Kern wurde ein FAIL*-Client gestartet, insgesamt also 832. Der Kampagnenserver wurde auf einem dazu separaten Computer gestartet, welcher über TCP mit einem dedizierten Datenbankserver in Verbindung stand.

Um die Dauer der Kampagnen auf praktikable Maße zu reduzieren wurden folgende Maßnahmen ergriffen:

- Die Benchmarks wurden mit kleinen Eingabegrößen ausgeführt, was Experimentlaufzeiten, Fehlerraumgrößen und die Dauer der Datenbankimporte re-

```
1  ...
2  [FAIL] Restore finished
3  [GoldenRunSkip] Resuming simulator
4  0000436505
5  [GoldenRunSkip] Stop symbol reached. Dumping state
6  ...
7  [GoldenRunSkip] Serial output (clean) saved to
   ↪  ./faildata/dev-example-grskip_results/clean/out.so
8  [GoldenRunSkip] Restoring from ./faildata/dev-example.state
9  ...
10 [FAIL] Restore finished
11 [GoldenRunSkip] Resuming simulator
12 [SkippingPlugin] SKIPPING 0x1000ee --> 0x100119 (7603 instructions) ss 0 rec 0
13 [SkippingPlugin] SKIPPING 0x100119 --> 0x1001ad (5808 instructions) ss 1 rec 0
14 0000436505
15 [GoldenRunSkip] Stop symbol reached.
16 [GoldenRunSkip] Execution time without skips: 9495 microseconds.
17 [GoldenRunSkip] Execution time with skips:    935 microseconds.
18 [GoldenRunSkip] SkippingPlugin had: 2 skips, 0 mismatches, 0 no_rec_found, 0
   ↪  listener blocks, 2 skipsec deactivations, 1 resumes.
19 [GoldenRunSkip] Stop symbol reached. Dumping state
20 ...
21 [GoldenRunSkip] Serial output (skips) saved to
   ↪  ./faildata/dev-example-grskip_results/skips/out.so
```

Lst. 5.1: Auszug der Ausgaben eines `goldenrun-skip` -Experiments.

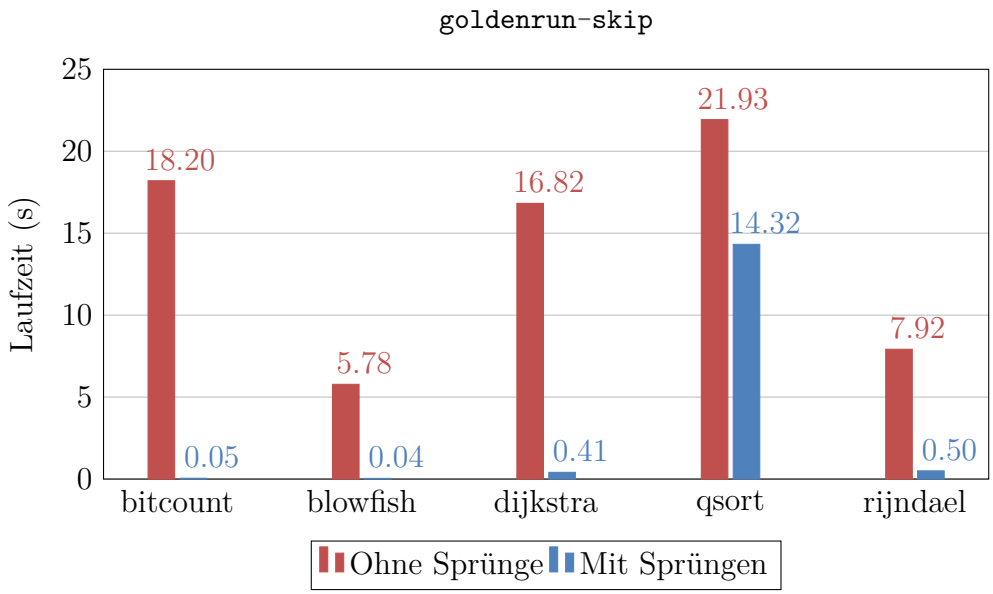


Abb. 5.1: Vergleich der Laufzeiten der Benchmarks ohne Fehlerinjektion.

duzierte. Für *blowfish*, *rijndael*und *dijkstra* wurden eigene Eingaben definiert, welche kleiner waren als die, die MiBench vorsieht[1].

- Auf eine vollständige Abdeckung des Fehlerraums wurde verzichtet. Stattdessen wurde über das Tool `prune-trace` [Sch16] mit den Optionen
  `--prune-method SamplingPruner --samplesize 1000000`
  eine repräsentative Auswahl von Experimenten getroffen[2].

- Der Server wurde mit der Option `--inject-bursts` gestartet. Dadurch werden Fehler in vollständigen Bytes statt einzelnen Bits injiziert. Dies reduziert den Fehlerraum um den Faktor 8.

Die Länge eines Experiments in Ticks, ab der das Ergebnis *Timeout* festgehalten und das Experiment abgebrochen wird, wurde auf das 1,5-fache der Dauer des Golden Run gesetzt. Die Mindestlänge der überspringbaren Abschnitte wurde

[1] Für *dijkstra*: Graph mit 30 Knoten statt 100, fünf kürzeste Wege statt zehn. Für *blowfish* und *rijndael*: Rund 50000 Bytes statt 312000.

[2] Der `SamplingPruner` untersucht Zufallskoordinaten im Fehlerraum. Wird dabei eine Def/Use-Äquivalenzklasse (vgl. Abschnitt 2.4.1.2) mehrfach angetroffen, wird nur ein Experiment der Äquivalenzklasse durchgeführt, dieses dafür aber höher gewichtet. Im Extremfall ist die Anzahl erzeugter Jobs gleich der Stichprobengröße, für gewöhnlich aber weniger, wie in Tabelle 5.3 zu sehen ist.

auf 10^6 dynamische Instruktionen festgelegt. Das Ermitteln optimaler Werte für die Mindestlänge stand nicht im Fokus dieser Evaluation. Stattdessen sollte ein grundsätzlicher Nachweis erbracht werden, dass Sprünge die Kampagnen beschleunigen können. Der – gemessen an der Simulatorgeschwindigkeit – relative hohe Wert 10^6 wurde gewählt, um eine möglichst geringe Anzahl von Listenerblockierungen und Mismatches zu erhalten, was den Overhead des `SkippingPlugin` minimal werden lässt. Als positiver Nebeneffekt wurde dadurch auch der zeitliche Aufwand für das Importieren und Filtern der Aufzeichnungen in die Datenbank reduziert.

5.4.2 Ergebnisse

Die Größe der (gefilterten) Protobuf-Dateien für die überspringbaren Abschnitte und ihrer Aufzeichnungen (Abschnitte 4.3 und 4.4), des Trace und der seriellen Ausgaben sind in Tabelle 5.2 aufgelistet. Wie zu sehen ist, waren die benötigten Datenmengen – selbst für die Benchmarks *qsort*, *blowfish* und *rijndael*, die viele Speicheroperationen durchführen – im Verhältnis zum Trace relativ gering. Die Größe des gespeicherten Maschinenzustands (Checkpoints) bei der ersten Instruktion des Zielprogramms – der `main()` -Funktion – betrug jeweils ca. 1 MB. Die Anzahl überspringbarer Abschnitte und Aufzeichnungen sowie die Länge dieser Aufzeichnungen stellt Abb. 5.2 dar.

Benchmark	Abschnitte	Aufzeichnungen	Trace	Ausgaben
bitcount	67 B	3,5 KB	3,4 MB	385 B
blowfish	55 B	1,7 MB	16,1 MB	12 B
dijkstra	43 B	1,2 MB	51,9 MB	63.5 KB
qsort	62 B	8,7 MB	71,5 MB	3,4 MB
rijndael	50 B	2,3 MB	18.6 MB	12 B

Tab. 5.2: Größen der in der Vorbereitungsphase generierten Dateien.

Der Server speichert die von den Clients empfangenen Ergebnisse in einer Tabelle der Datenbank. Nach Durchführung beider Kampagnen eines Benchmarks wurden diese Tabellen in Dateien exportiert und mittels des Unix-Tools *diff* verglichen. Dabei zeigte sich, dass die Ergebnisse beider Kampagnen identisch waren. Die gemessenen Laufzeiten werden in Abb. 5.3 und Tabelle 5.3 gezeigt. $T_{Original}$ steht für die Laufzeit der Kampagne ohne Sprünge, $T_{Sprünge}$ für die Laufzeit mit Sprüngen. Da die Nutzung der beschriebenen Hardware nicht exklusiv erfolgte,

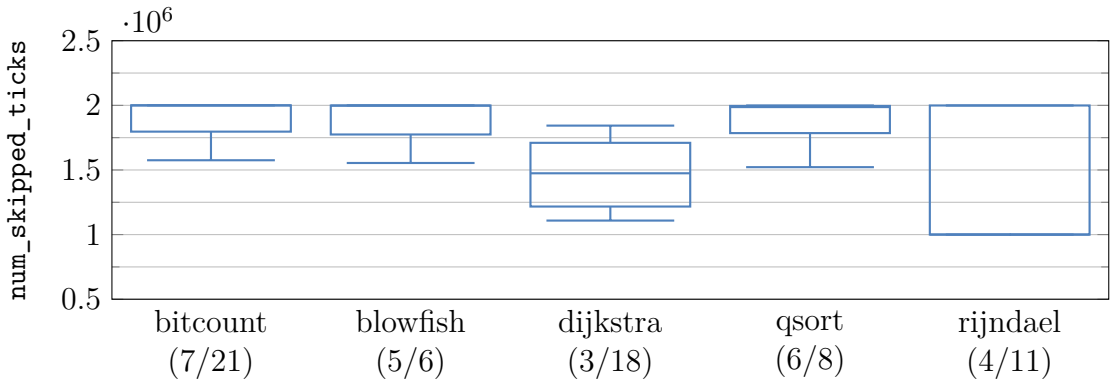


Abb. 5.2: Aufzeichnungslängen der für die Benchmarks generierten überspringbaren Abschnitte. In Klammern: Anzahl überspringbarer Abschnitte/Anzahl Aufzeichnungen.

kann eine Beeinflussung der Messergebnisse durch die Aktivitäten anderer Nutzer stattgefunden haben. Der Effekt kann jedoch als vernachlässigbar betrachtet werden, da Wiederholungen der Messungen zu späteren Zeitpunkten nahezu identische Ergebnisse produzierten. Nicht berücksichtigt wurde die Zeit, die der Import der überspringbaren Abschnitte und ihrer Aufzeichnungen in die Datenbank beanspruchte. Eine Erhöhung der Anzahl der Experimente N (größere Stichprobe oder vollständige Fehlerraumabdeckung) würde diesen Aufwand relativieren.

Benchmark	Ticks	Experimente	$T_{Original}$	$T_{Sprünge}$	Reduktion
bitcount	$4,07 \cdot 10^7$	$5,91 \cdot 10^4$	39m 12s	41m 25s	-5,65 %
blowfish	$1,13 \cdot 10^7$	$2,41 \cdot 10^5$	12m 13s	8m 26s	30,97 %
dijkstra	$2,03 \cdot 10^7$	$9,16 \cdot 10^5$	2h 36m	1h 41m	35,27 %
qsort	$4,19 \cdot 10^7$	$7,65 \cdot 10^5$	1h 52m	1h 38m	12,5 %
rijndael	$1,41 \cdot 10^7$	$3,76 \cdot 10^5$	22m 33s	9m 56s	55,95 %

Tab. 5.3: Zusammenfassung der Benchmark-Ergebnisse.

Die Tabellen 5.4 bis 5.8 präsentieren die Ergebnisse der Benchmarks im Detail. Mit Ausnahme der Anzahl der Ergebnisse sind jeweils Mittelwerte angegeben. Die Zeile „Blockierung durch Listener“ gibt an, wie oft die Situation eintrat, dass ein überspringbarer Abschnitt erreicht wurde, alle zugehörigen Aufzeichnungen jedoch durch Listener blockiert waren. Falls mindestens eine Aufzeichnung nicht blockiert

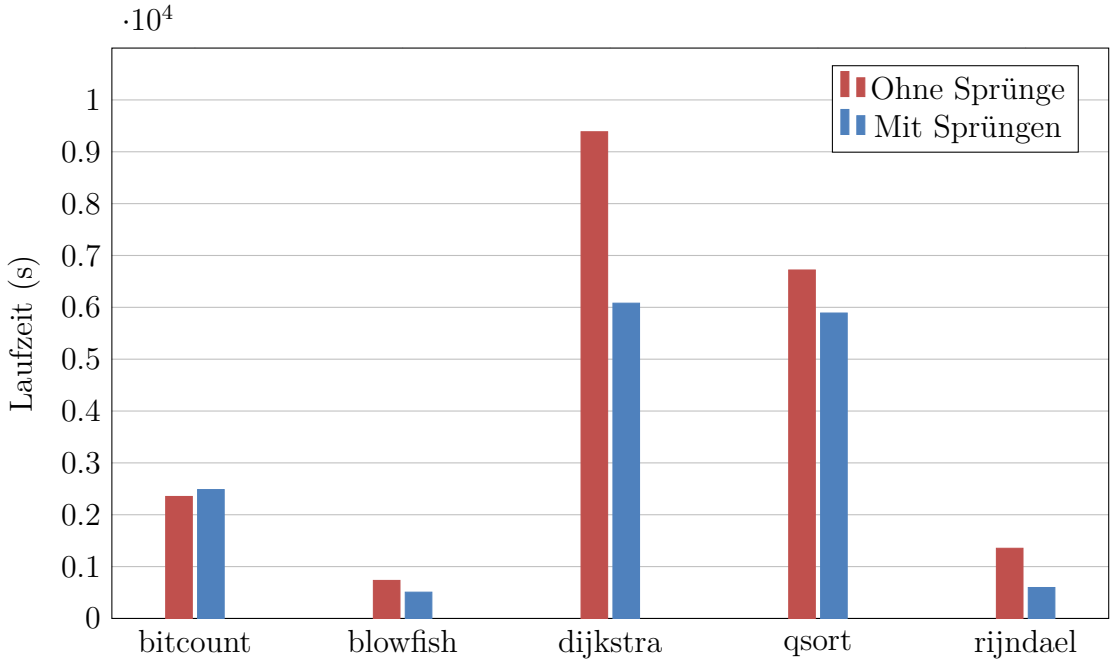


Abb. 5.3: Vergleich der Laufzeiten der Benchmarkkampagnen.

war, davon jedoch keine zum Maschinenzustand passte, erhöhte dies den Zähler „Kein Match gefunden“.

Den Ergebnissen für den *bitcount*-Benchmark (Tabelle 5.4) ist zu entnehmen, dass die durchschnittliche Laufzeit eines Experiments, welches das Ergebnis *Timeout* erreichte, sehr viel höher war als die durchschnittliche Laufzeit bei anderen Ergebnissen. Die Timeout-Experimente haben, trotz ihrer relativ geringen Anzahl, aufgrund ihrer überproportional großen Laufzeiten einen signifikanten Anteil an der Gesamtlaufzeit der Kampagne (ohne Sprünge ca. 81 %, mit 91 %). Die Sprünge beschleunigten Experimente mit den Ergebnissen OK, Trap und SDC. Für das Timeout-Ergebnis trat jedoch eine deutliche Verschlechterung ein. Die durchschnittliche Laufzeit aller Experimente wurde deswegen nur leicht reduziert (Abb. 5.4). Eine Erhöhung der Kampagnenlaufzeit (Tabelle 5.3) trat dennoch auf. Eine mögliche Erklärung hierfür ist der Overhead für das Initialisieren des `SkippingPlugin`, inklusive des Einlesens der überspringbaren Abschnitte und ihrer Aufzeichnungen aus den Protobuf-Dateien. Da das `generic-experiment` – und das für die Evaluation verwendete Derivat `generic-skip-experiment` – vorsieht, dass

Clients nach der Bearbeitung eines Satzes von Jobs terminieren und neu starten[3], erfolgt das Einlesen der Daten bei jedem Neustart des Clients. Dieser Overhead könnte durch die Übermittlung größerer Job-Sätze reduziert werden.

Messgrösse	OK	Timeout	Trap	SDC	Gesamt
Anzahl der Experimente	37793	3944	16647	758	59142
$T_{Experiment}$ ohne Sprünge	5s	270,96s	2,87s	13,98s	22,28s
$T_{Experiment}$ mit Sprüngen	2,71s	287,02s	0,14s	8,26s	21,02s
Sprünge	10	12,6	10,79	19,48	10,52
Davon nach der FI	0,9	0	0	0	0,58
Blockierung durch Listener	1,29	10,64	8,76	2,75	4,03
Kein Match gefunden	18,17	0,36	0,3	0,89	11,73
Kontrollwechsel zum Simulator	19,56	11,02	9,09	4,41	15,85

Tab. 5.4: Ergebnisse des Benchmarks *bitcount*.

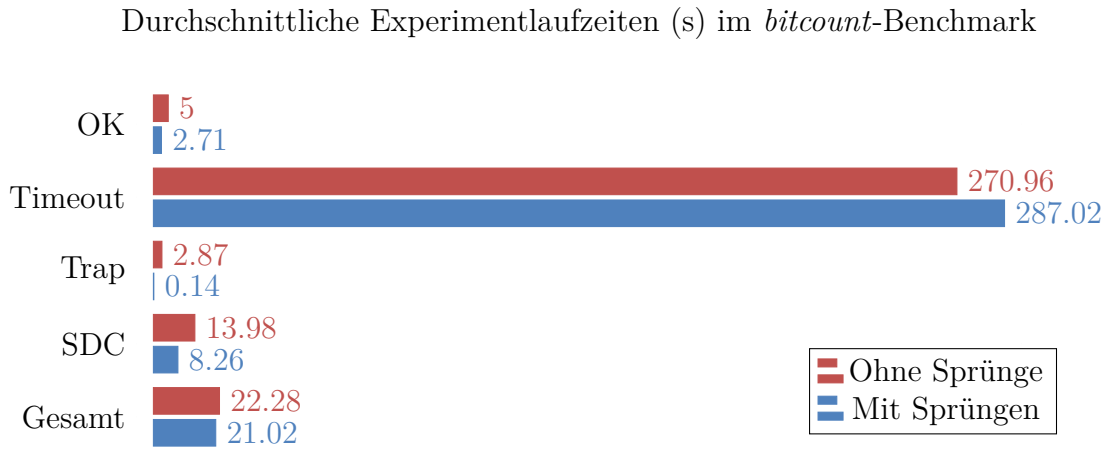


Abb. 5.4: Vergleich der durchschnittlichen Experimentlaufzeiten für den Benchmark *bitcount*, aufgeschlüsselt nach Ergebnis.

[3]Diese Vorgehensweise erleichtert die Kontrolle über die Clients während der Kampagne, bspw. wenn die Berechnungen pausiert werden sollen.

Auch bei den Benchmarks *blowfish* (Tabelle 5.5) und *qsort* (Tabelle 5.6) wurden Timeout-Experimente verlangsamt. Diese fielen jedoch aufgrund eines geringeren Anteil an der Gesamtzahl der Experimente weniger ins Gewicht, sodass trotzdem eine Verkürzung der Kampagnenlaufzeiten beobachtet wurde.

Messgrösse	OK	Timeout	Trap	SDC	Gesamt
Anzahl der Experimente	240430	170	253	1108	241961
$T_{Experiment}$ ohne Sprünge	1,67s	99,34s	7,27s	1,56s	1,77s
$T_{Experiment}$ mit Sprüngen	0,9s	107,44s	6,96s	1,85s	0,98s
Sprünge	2,96	2,7	3,62	0,26	2,94
Davon nach der FI	0,38	0,01	0,02	0	0,38
Blockierung durch Listener	3,44	3,31	2,17	1,09	3,42
Kein Match gefunden	2,39	0,61	0,81	8,5	2,42
Kontrollwechsel zum Simulator	5,83	4	3,34	9,62	5,84

Tab. 5.5: Ergebnisse des Benchmarks *blowfish*.

Messgrösse	OK	Timeout	Trap	SDC	Gesamt
Anzahl der Experimente	689835	78	373	75196	59510
$T_{Experiment}$ ohne Sprünge	6,32s	235,47s	7,34s	6,51s	6,37s
$T_{Experiment}$ mit Sprüngen	5,43s	261,87s	5,1s	4,98s	5,41s
Sprünge	3,77	6,96	7,72	6,02	4
Davon nach der FI	0	0	0	0	0
Blockierung durch Listener	5,17	1,13	0,43	2,39	4,9
Kein Match gefunden	4,51	0,19	0,02	2,03	4,26
Kontrollwechsel zum Simulator	9,68	2,06	1,35	4,99	9,22

Tab. 5.6: Ergebnisse des Benchmarks *qsort*.

Die größten Verbesserungen der Laufzeit durch Sprünge waren bei den Benchmarks *rijndael* (Tabelle 5.7) und *dijkstra* (Tabelle 5.8) zu beobachten. In diesen wurden – im Gegensatz zu den anderen Tests – (leichte) Beschleunigungen von Timeout-Experimenten erreicht.

Messgrösse	OK	Timeout	Trap	SDC	Gesamt
Anzahl der Experimente	67361	44	114	308958	376477
$T_{Experiment}$ ohne Sprünge	2,2s	127,81s	2,01s	2,31s	2,3s
$T_{Experiment}$ mit Sprüngen	1,21s	122,87s	0,12s	0,51s	0,65s
Sprünge	5,38	6,52	9,46	8,85	8,23
Davon nach der FI	1,58	0	0	0,09	0,35
Blockierung durch Listener	9,16	4,39	1,94	4,19	5,08
Kein Match gefunden	5,45	1,93	0,22	1,41	2,13
Kontrollwechsel zum Simulator	14,61	6,61	2,89	5,6	7,21

Tab. 5.7: Ergebnisse des Benchmarks *rijndael*.

Messgrösse	OK	Timeout	Trap	SDC	Gesamt
Anzahl der Experimente	896645	886	3314	15400	916245
$T_{Experiment}$ ohne Sprünge	6,72s	269,97s	9,52s	6,7s	6,98s
$T_{Experiment}$ mit Sprüngen	3,77s	269,12s	4,63s	4,75s	4,04s
Sprünge	8,5	14,3	13,15	6,19	8,48
Davon nach der FI	0,15	0,06	0,02	0,84	0,16
Blockierung durch Listener	1,38	0,53	0,69	1,39	1,38
Kein Match gefunden	15,21	5,66	0,1	19,35	15,22
Kontrollwechsel zum Simulator	16,63	6,8	1,29	20,86	16,63

Tab. 5.8: Ergebnisse des Benchmarks *dijkstra*.

5.5 Diskussion der Ergebnisse

Die Anforderung der *Generalität* wurde teilweise erfüllt. Wie Abschnitt 5.4 zeigte, war die Implementierung für unterschiedliche Zielprogramme anwendbar. Jedoch begründete Abschnitt 3.2.4, dass die vorgestellte Methode nicht angewandt werden kann, wenn das Zielprogramm Interrupts verwendet oder Peripheriegeräte via *Memory Mapped I/O* (MMIO) ansteuert. Weiterhin wurde in Abschnitt 4.6 kenntlich gemacht, dass keine Gleitkommaarithmetik benutzt werden darf. Teile der Imple-

mentierung erfolgten spezifisch für Bochs, eine zukünftige Unterstützung anderer Backends ist aber nicht grundsätzlich ausgeschlossen.

Die Implementierung kann als *nutzerfreundlich* bezeichnet werden. Das Experiment `generic-tracing` wurde derart erweitert, dass die Unterteilung und Aufzeichnung überspringbarer Abschnitte mit einfachen Kommandozeilen aktiviert werden kann (Abschnitt 4.3). Die größte Herausforderung für die Nutzer ist hierbei das Setzen einer geeigneten Mindestlänge für die überspringbaren Abschnitte, da das Finden von geeigneten Werten verhältnismäßig komplex werden kann (Abschnitt 3.2.3). Für den Import der Daten in die Datenbank wurde das Tool `import-skips` bereitgestellt, welches ähnlich leicht zu benutzen ist wie FAIL*s `import-trace`. Zur Nutzung von Sprüngen in einer Fehlerinjektionskampagne müssen einem typischen Experiment nur wenige Zeilen Code hinzugefügt werden, um das `SkippingPlugin` zu initialisieren, sowie die vom Experiment benötigten Listener bekannt zu machen (Abschnitt 4.5).

Bei allen der fünf getesteten Benchmarks wurde die Anforderung der *Ergebnistreue* vollständig erfüllt: Die Kampagnen mit Sprüngen wiesen identische Ergebnisse zu denen ohne Sprünge auf. Die Laufzeiten wurden im verwendeten Messaufbau um bis zu 56 % reduziert, was für die *Effektivität* der Methode spricht. Bei einem der getesteten Benchmarks war jedoch eine Verschlechterung der Laufzeit um ca. 6 % zu beobachten. Da das Plugin eine Erhöhung der durchschnittlichen Laufzeit von Experimenten mit dem Ergebnis Timeout verursachte, und diese einen hohen Anteil an der Gesamtlaufzeit der Kampagne haben, wurde die durchschnittliche Laufzeit aller Experimente nur gering reduziert. Diese Reduktion könnte durch den Initialisierungsprozedur des `SkippingPlugin` kompensiert worden sein, welche während der Kampagne von jedem Client aufgrund periodischer Neustarts mehrfach durchgeführt worden ist.

In einigen Kampagnen konnten die Experimente auch nach der Fehlerinjektion Sprünge durchführen, beim *bitcount*-Benchmark war dies in jedem Experiment durchschnittlich $0,58$-mal der Fall. Dies verdeutlicht den Vorteil der vorgestellten Methode gegenüber dem zugrundeliegenden Checkpoint-Konzept.

5.6 Zusammenfassung

Nach einer Definition der Evaluationskriterien und Vorgehensweise in Abschnitt 5.1 präsentierte Abschnitt 5.2 die dafür verwendeten Benchmarks der MiBench-Suite [Gut+01] Evaluation. Das Experiment `goldenrun-skip` zum Test der grundsätzlichen Funktionalität der Implementierung wurde in Abschnitt 5.3 vorgestellt. Den für die Messungen verwendeten Aufbau und die Ergebnisse präsentierte Abschnitt 5.4. Die Sprünge haben die Ergebnisse der Kampagne nicht verfälscht.

Eine Reduktion der Kampagnenlaufzeit konnte bei vier der fünf Benchmarks erreicht werden, im fünften Fall war eine leichte Verschlechterung zu beobachten. Dies kann durch den Overhead beim Einlesen überspringbarer Abschnitte und ihrer Aufzeichnungen erklärt werden. Sprünge konnten sowohl vor, als auch nach der Fehlerinjektion erfolgen. Auf Basis der gesammelten Messdaten und der vorigen Kapitel Kapitel 3 und 4 stellte Abschnitt 5.5 fest, dass die definierten Anforderungen Effektivität, Ergebnistreue, Generalität und Nutzerfreundlichkeit ganz oder teilweise erfüllt wurden.

6 Zusammenfassung

Nach Recherchen zu den Grundlagen simulationsbasierter Fehlerinjektion und einer Einarbeitung in das dafür entwickelte Tool FAIL* wurde ein Ansatz zur Reduktion der Laufzeit von Fehlerinjektionsexperimenten entworfen, der dann als Methode zur Erweiterung des FAIL*-Tools implementiert und anschließend mit entsprechenden Benchmarks evaluiert wurde.

6.1 Fazit

Der vorgestellte Ansatz der *überspringbaren Abschnitte* basiert auf der Beobachtung, dass in einer typischen *Fehlerinjektionskampagne* eine hohe Anzahl von Experimenten durchgeführt wird, die sich untereinander stark ähneln. Die Simulationsausführung kann beschleunigt werden, indem die getätigten *Zustandsraumtransformationen* zuvor aufgezeichnet und zusammengefasst werden. Die Methode ähnelt dem Konzept der Checkpoints, jedoch werden für überspringbare Abschnitte statt des gesamten Maschinenzustands nur die jeweils relevanten Teile daraus gespeichert. Dadurch können sie auch nach einer Fehlerinjektion anwendbar sein. Soweit dem Autor bekannt, wurde bis zur Abgabe dieser Arbeit keine vergleichbare Methode publiziert.

Das Generieren und Aufzeichnen überspringbarer Abschnitte kann im Experiment `generic-tracing` erfolgen. Zur Durchführung von Sprüngen mit dem entwickelten `SkippingPlugin` müssen die Experimentdefinitionen um wenige Quellcodezeilen ergänzt werden, die das Plugin einbinden und die vom Experiment verwendeten Listener bekanntgeben. Tests mit fünf Benchmarks der MiBench-Suite [Gut+01] – aufgesetzt auf das Betriebssystem eCos [Mas03] – zeigten, dass die Ergebnisse der Kampagnen durch die Sprünge nicht verändert werden. Bei vier der Benchmarks war eine Reduktion der Kampagnenlaufzeiten von 12,5 % bis 56 % zu beobachten. Bei einem Benchmark erhöhte sich die Laufzeit um ca. 6 %. Eine mögliche Ursache für diese Erhöhung ist der für das Initialisieren des Plugins verursachte Overhead, der die (geringe) Reduktion der durchschnittlichen Experimentlaufzeit kompensiert haben könnte.

Der Ansatz ist zur Zeit noch limitiert auf Zielprogramme, welche keine Interrupts oder *Memory Mapped I/O* verwenden. Als Backend wird Bochs [Law96] voraus-

gesetzt, eine zukünftige Unterstützung von anderen Backends ist jedoch möglich. Darüber hinaus ist die Menge der Zielprogramme auf solche ohne Gleitkommaarithmetik eingeschränkt, was durch eine Modifikation bestimmter FAIL*-Komponenten und des Datenbankschemas aufgehoben werden könnte.

6.2 Ausblick

In weiterführenden Arbeiten könnten sowohl der Ansatz der Ansatz der überspringbaren Abschnitte an sich, als auch die konkrete Implementierung in FAIL* verbessert werden.

Die Effektivität der Abschnitte hängt wesentlich von den gewählten Startpunkten und ihrer Länge ab (Abschnitt 3.2.3). In einer weiterführenden Arbeit könnte ein Verfahren entwickelt werden, mit dem günstige Unterteilungen – d. h. solche mit möglichst großem Laufzeitgewinn für die Fehlerinjektionskampagne – automatisiert gefunden werden können. Zusätzlich kann in diesem Kontext auch die Frage untersucht werden, ob und wie mit dem Ansatz Zielprogramme zu beschleunigen sind, die MMIO und Interrupts verwenden. Zu prüfen ist weiterhin, ob ein *Lernen* neuer überspringbarer Abschnitte während der Kampagne einen Vorteil darstellt. Es ist denkbar, dass viele Experimente ähnliche Fehlerzustände erreichen (vgl. Abschnitt 2.4.2.2), sodass dafür Abschnittsaufzeichnungen angefertigt und verteilt werden könnten. Zentral ist hierbei ist die Frage, wie identifiziert werden kann, wie viele Experimente eine solche neu gelernte, fehlerbehaftete Aufzeichnung verwenden können, so dass sich der Aufwand für das Aufzeichnen und Übertragen der Abschnitte lohnt.

Bezüglich der Verbesserungen der Implementierung ist an erster Stelle die Unterstützung weiterer Backends zu nennen, wofür Teile des `SkippingPlugin` angepasst werden müssten. Auch könnte eine Unterstützung von Zielprogramme mit Gleitkommaarithmetik implementiert werden. Die Schnittstelle zum Blockieren bestimmter Sprünge akzeptiert derzeit Listener für Breakpoints, Speicherzugriffe und Timer, eine Erweiterung auf alle von FAIL* angebotenen Listener ist wünschenswert. Die Effektivität der Sprünge könnte möglicherweise durch eine Optimierung des Quellcodes und spezialisierte Datenstrukturen gesteigert werden. Experimente mit Timeout-Ergebnis sind hierbei besonders zu berücksichtigen, da sie einen hohen Anteil an der Kampagnenlaufzeit besitzen. Die Übertragung der überspringbaren Abschnitte und ihrer Aufzeichnungen könnte zukünftig statt über Dateien – analog zu den Experimentparametern – über TCP erfolgen. Dies würde auch die Übertragung neuer, während der Kampagne gelernter Abschnitte (s. o.) vereinfachen.

Literatur

[AL80] Michel E Adiba und Bruce G Lindsay. „Database snapshots“. In: *Proceedings of the sixth international conference on Very Large Data Bases-Volume 6*. VLDB Endowment. 1980, S. 86–91.

[Avi+04] Algirdas Avižienis, J-C Laprie, Brian Randell und Carl Landwehr. „Basic concepts and taxonomy of dependable and secure computing“. In: *IEEE transactions on dependable and secure computing* 1.1 (2004), S. 11–33.

[Bar+05] Raul Barbosa, Jonny Vinter, Peter Folkesson und Johan Karlsson. „An Approach to Reducing the Cost of Fault Injection“. In: Aug. 2005, S. 129–134.

[Bel05] Fabrice Bellard. „QEMU, a fast and portable dynamic translator.“ In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, S. 41–46.

[Ber+02] Luis Berrojo, Isabel González, Fulvio Corno, Matteo Sonza Reorda, Giovanni Squillero, Luis Entrena und Celia Lopez. „New techniques for speeding-up fault-injection campaigns“. In: *Proceedings of the conference on Design, Automation and Test in Europe*. IEEE Computer Society. 2002, S. 847–852. DOI: 10.1109/DATE.2002.998398.

[Bin+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti u. a. „The gem5 simulator“. In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), S. 1–7.

[BSS13] Christoph Borchert, Horst Schirmeier und Olaf Spinczyk. „Generative software-based memory error detection and correction for operating system data structures“. In: *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE. 2013, S. 1–12.

[BT93] Dominique Brière und Pascal Traverse. „AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems“. In: *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE. Juni 1993, S. 616–623. DOI: 10.1109/FTCS.1993.627364.

[CL85] K Mani Chandy und Leslie Lamport. „Distributed snapshots: Determining global states of distributed systems“. In: *ACM Transactions on Computer Systems (TOCS)* 3.1 (1985), S. 63–75.

[DIN00] DIN. *Spezifikation und Nachweis der Zuverlässigkeit, Verfügbarkeit, Instandhaltbarkeit, Sicherheit (RAMS)*. Norm. März 2000.

[DW11] Anand Dixit und Alan Wood. „The impact of new technology on soft error rates“. In: *Reliability Physics Symposium (IRPS), 2011 IEEE International*. IEEE. 2011, 5B–4.

[Gol+06] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda und Massimo Violante. *Software-implemented hardware fault tolerance*. Springer Science & Business Media, 2006.

[Goo08] Google, Inc. *Google Protocol Buffers*. 2008. URL: https://developers.google.com/protocol-buffers/.

[GS95] Jens Guthoff und Volkmar Sieh. „Combining software-implemented and simulation-based fault injection into a single fault injection method“. In: *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*. IEEE. 1995, S. 196–206.

[Gut+01] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge und Richard B Brown. „MiBench: A free, commercially representative embedded benchmark suite“. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. IEEE. Dez. 2001, S. 3–14. DOI: 10.1109/WWC.2001.990739.

[Har+13] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi und Pradeep Ramachandran. „Relyzer: Application resiliency analyzer for transient faults“. In: *IEEE Micro* 33.3 (2013), S. 58–66.

[Har+14] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V Adve und Helia Naeimi. „GangES: Gang error simulation for hardware resiliency evaluation“. In: *ACM SIGARCH Computer Architecture News*. Bd. 42. 3. IEEE Press. 2014, S. 61–72.

[Hof+16] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, Horst Schirmeier, Daniel Lohmann und Wolfgang Schröder-Preikschat. „Experiences with software-based soft-error mitigation using AN codes“. In: *Software Quality Journal* 24.1 (2016), S. 87–113.

[Hof16] Martin Hoffmann. „Konstruktive Zuverlässigkeit: Eine Methodik für zuverlässige Systemsoftware auf unzuverlässiger Hardware“. Dissertation. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2016.

[IRC13] International Roadmap Comittee. *International technology roadmap for semiconductors, 2013 edn. (executive summary)*. 2013. URL: `http://www.itrs2.net/itrs-reports.html` (besucht am 02. 08. 2017).

[KD14] Maha Kooli und Giorgio Di Natale. „A survey on simulation-based fault injection tools for complex systems“. In: *Design & Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On*. IEEE. 2014, S. 1–6.

[Law96] Kevin P Lawton. „Bochs: A portable PC emulator for Unix/X“. In: *Linux Journal* 1996.29es (1996), S. 7.

[Lev+09] Régis Leveugle, A Calvez, Paolo Maistri und Pierre Vanhauwaert. „Statistical fault injection: Quantified error and confidence“. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design und Automation Association. 2009, S. 502–506.

[Li+10] Xin Li, Michael C Huang, Kai Shen und Lingkun Chu. „A realistic evaluation of memory hardware errors and software system susceptibility“. In: *Proc. USENIX Annual Technical Conference (ATC'10)*. 2010, S. 75–88.

[LT13] Jianli Li und Qingping Tan. „SmartInjector: Exploiting intelligent fault injection for SDC rate analysis“. In: *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. IEEE. Okt. 2013, S. 236–242. DOI: `10.1109/DFT.2013.6653612`.

[Mas03] Anthony J Massa. *Embedded software development with eCos*. Prentice Hall Professional, 2003.

[Muk11] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[NX06] Vijaykrishnan Narayanan und Yuan Xie. „Reliability concerns in embedded system designs“. In: *Computer* 39.1 (Jan. 2006), S. 118–120.

[Par+00] B Parrotta, Maurizio Rebaudengo, M Sonza Reorda und Massimo Violante. „Speeding-up fault injection campaigns in VHDL models“. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2000, S. 27–36.

[Pel+08] Andrea Pellegrini, Kypros Constantinides, Dan Zhang, Shobana Sudhakar, Valeria Bertacco und Todd Austin. „CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework“. In: *Computer Design, 2008. ICCD 2008. IEEE International Conference on*. IEEE. 2008, S. 363–370.

[Rat05] Dominic Rath. „OpenOCD: Open on-chip debugging“. Diplomarbeit. Fachhochschule Augsburg, 2005.

[SBS15] Horst Schirmeier, Christoph Borchert und Olaf Spinczyk. „Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors“. In: *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE. 2015, S. 319–330.

[Sch+15] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann und Olaf Spinczyk. „FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance“. In: *Dependable Computing Conference (EDCC), 2015 Eleventh European*. IEEE. Sep. 2015, S. 245–255.

[Sch16] Horst Schirmeier. „Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance“. Dissertation. Technische Universität Dortmund, Juli 2016. DOI: 10.17877/DE290R-17222.

[Smi+95] D Todd Smith, Barry W Johnson, Joseph A Profeta und Daniele G Bozzolo. „A method to determine equivalent fault classes for permanent and transient faults“. In: *Reliability and Maintainability Symposium, 1995. Proceedings., Annual*. IEEE. 1995, S. 418–424.

[Son+11] Ningfang Song, Jiaomei Qin, Xiong Pan und Yan Deng. „Fault injection methodology and tools“. In: *Electronics and Optoelectronics (ICEOE), 2011 International Conference on*. Bd. 1. IEEE. 2011, S. V1–47.

[Yos13] Junko Yoshida. *Toyota Case: Single Bit Flip That Killed*. 2013. URL: http://www.eetimes.com/document.asp?doc_id=1319903 (besucht am 02. 08. 2017).

[ZA04] Haissam Ziade und Raoul Ayoubi Raficand Velazco. „A survey on fault injection techniques“. In: *The International Arab Journal of Information Technology* 1.2 (Juli 2004), S. 171–186.

Abbildungsverzeichnis

1.1 Idee der überspringbaren Abschnitte 3

2.1 Fault Tolerance Assessment Cycle im Kontext von FAIL* 9
2.2 Übersicht der Architektur des Plumbing Layer von FAIL* 10
2.3 Fehlerraum für Single-Bit-Flip-Experimente 12
2.4 *Def/use pruning* 14

3.1 Illustration der Ähnlichkeit von FI-Experimenten 21
3.2 Vergleichende Illustration der zu speichernden Daten für Checkpoints und überspringbare Abschnitte. 22
3.3 Illustration einer Aufzeichnung 24
3.4 Illustration einer Simulation mit fünf Sprüngen 25

4.1 Übersicht der Implementierung. 31
4.2 Instruktionshistogramm 35
4.3 Erweiterung des Datenbankschemas 38
4.4 Blockierung einer Aufzeichnung durch einen MemAccessListener . . 40
4.5 Blockierung einer Aufzeichnung durch einen BPListener 41

5.1 Vergleich der Laufzeiten der Benchmarks ohne Fehlerinjektion. . . . 49
5.2 Längen der Aufzeichnungen der Benchmarks 51
5.3 Vergleich der Laufzeiten der Benchmarkkampagnen. 52
5.4 Vergleich der durchschnittlichen Experimentlaufzeiten für den Benchmark *bitcount*, aufgeschlüsselt nach Ergebnis. 53

Tabellenverzeichnis

5.1 Beschreibung der verwendeten Benchmarks. 46
5.2 Größen der in der Vorbereitungsphase generierten Dateien. 50
5.3 Zusammenfassung der Benchmark-Ergebnisse. 51
5.4 Ergebnisse des Benchmarks *bitcount*. 53
5.5 Ergebnisse des Benchmarks *blowfish*. 54
5.6 Ergebnisse des Benchmarks *qsort*. 54
5.7 Ergebnisse des Benchmarks *rijndael*. 55
5.8 Ergebnisse des Benchmarks *dijkstra*. 55

Listingverzeichnis

3.1 Disassemblierte `square()`-Funktion. 23

4.1 `SkipSec.h` 33
4.2 Initialisierung des `SkippingPlugin` im `generic-skip-experiment` . . 39
4.3 Redekodierung der Instruktion nach Setzen des Befehlszählers. . . . 43

5.1 Auszug der Ausgaben eines `goldenrun-skip` -Experiments. 48

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift