technische universität
dortmund

Masterarbeit

Automatisierte Analyse von Fehlerinjektions-experimenten mit Fail*

Richard Hellwig
11. April 2014

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

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

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 11. April 2014

Richard Hellwig

Zusammenfassung

Fehlerinjektion ist ein experimentelles Werkzeug zur Untersuchung der Zuverlässigkeit von unter Gesichtspunkten der Fehlertoleranz entwickelten Hard- und Softwaresystemen. Das Werkzeug Fail*, das im Rahmen des Forschungsprojekts „Dependability Aspects in Configurable Embedded Operating Systems“ *(DanceOS)* entwickelt wird, bietet die Möglichkeit zur parallelen und verteilten Durchführung von Fehlerinjektionsexperimenten. Selbst bei geringer Größe der damit untersuchten Zielsysteme fallen dabei enorme Mengen von Ergebnisrohdaten an, die ohne geeignete Weiterverarbeitung nur sehr oberflächliche Zuverlässigkeitsbewertungen zulassen. Insbesondere für gezielte Härtung der Zielsysteme benötigt ein Anwendungsentwickler jedoch detaillierte Informationen über räumliche und zeitliche Ausbreitung und Wirkung von Hardwarefehlern, um einen anwendungsspezifischen Kompromiss zwischen Fehlertoleranz und Ressourcenverbrauch treffen zu können.

Ziel dieser Masterarbeit ist es daher, geeignete Verfahren zu Weiterverarbeitung, Analyse und Darstellung von Fehlerinjektionsresultaten zu entwickeln. Durch Zusammenführung von passend gewählten Aggregationen und statischen Informationen über das Zielsystem werden aus den Rohdaten sowohl grob- als auch feingranular Informationen über die Zuverlässigkeit von *Teilen* der Software (räumlich und zeitlich) destilliert und dem Entwickler geeignet zugänglich gemacht. Als zentrale Frage wird hierbei geklärt, welche Verfahren sich hierbei besonders eignen, den Entwickler beim gezielten Einbringen von modularen Fehlertoleranzmaßnahmen zu unterstützen, oder diesen Vorgang sogar zu teilautomatisieren. Die Ergebnisse dieser Arbeit werden mit bereits vorhandenen, manuell durchgeführten Analysen verglichen. Die Ergebnisse zeigen, dass mit den in dieser Arbeit entwickelten Analyse-Verfahren eine detaillierte Aggregation auf Software-Komponenten des Hochsprachen-Codes möglich ist. Dabei können wesentlich genauere Aussagen über die Fehleranfälligkeit der Software-Komponenten getätigt werden, als es bei den bereits existierenden Analysen der Fall war.

Inhaltsverzeichnis

1. Einleitung

Durch die immer kleiner werdenden Strukturen von Hardwarekomponenten steigt die Wahrscheinlichkeit einer Störung durch sog. transiente Fehler [1]. Diese Fehler können fatale Auswirkungen innerhalb von Anwendungen oder gar im Betriebssystem zur Folge haben. Während derartige flüchtige Fehler innerhalb eines Multimediasystems vernachlässigbar sind, können die Folgen eines solchen Fehlers innerhalb eines Systems mit harten Echtzeitbedingungen, wie z.B. einer Steuerung für einen Airbag, derart fatal sein, dass sie Menschenleben gefährden.

Die Ursache von Fehlern kann im Wesentlichen in drei Kategorien eingeteilt werden [2]. Die erste Kategorie stellen Fehler dar, die durch den Entwickler verursacht werden. Dazu zählen z.B. unbeabsichtigte Zugriffe in „fremde“ Speicherbereiche. Die zweite Kategorie stellen beabsichtigte Angriffe auf ein System dar. Bei dieser Art von Fehler versucht ein Angreifer Schwachstellen innerhalb einer Software auszunutzen. Sowohl Fehler, die durch Entwickler verursacht werden, als auch solche, die durch Angriffe entstehen, sind Software-Fehler. Fehler, die durch physikalische Einflüsse entstehen, haben dagegen ihren Ursprung in der Hardware und bilden die dritte Kategorie. Wie ein solcher Hardware-Fehler zustande kommt, wird genauer in den Grundlagen betrachtet.

Software ohne weitere Untersuchungen gegen jeden möglichen Fehler zu härten ist nicht effizient. Sowohl die Laufzeit, als auch der Speicherbedarf und Energiebedarf würden um ein Vielfaches steigen. Dies ist insbesondere für eingebettete Systeme, die häufig hohen Einschränkungen bzgl. dem Ressourcenverbrauch unterliegen, nicht praktikabel. Aus diesem Grund ist es zwingend notwendig, Software individuell – und somit Ressourcensparend – gegen Fehler zu härten. Um dies zu ermöglichen muss eine genauere Analyse erfolgen.

Wie anfällig die einzelnen Komponenten eines Systems bzgl. transienter Hardware-Fehler ist, wird in dem Forschungsprojekt *DanceOS* [3] untersucht. Zu diesem Zweck werden Fehlerinjektions-Experimente durchgeführt, die durch Simulation die Auswirkungen von Hardware-Fehlern evaluieren. Mit Hilfe der Ergebnisdaten aus diesen Experimenten können Analysen durchgeführt werden, die es ermöglichen zu erkenne, welche Bestandteile der Software besonders anfällig für transiente Hardware-Fehler sind. Mit diesen Erkenntnissen können wiederum geeignete Fehlertoleranzmaßnahmen entwickelt werden, die zur Härtung der Software dienen.

1.1. Ziel der Arbeit

In Abbildung 1.1 ist der allgemeine Softwareentwicklungs-Kreislauf zur Härtung von Software zu sehen.

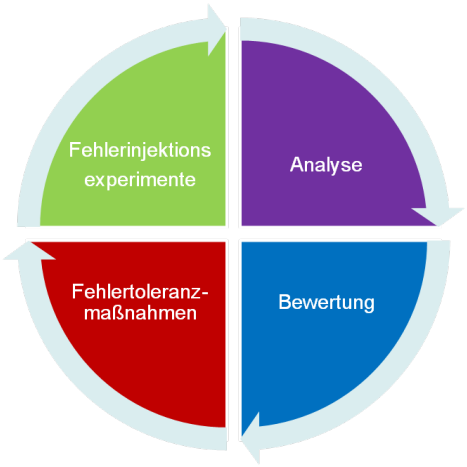

Abbildung 1.1.: Kreislauf zur Härtung von Software mit Hilfe von Fehlerinjektionsexperimenten

Im ersten Schritt werden in die zu härtende Software Fehler injiziert. In dieser Arbeit wird dazu das Framework FAIL* [4] verwendet, das im Rahmen des Forschungsprojektes *DanceOS* [3] entstanden ist. Diese Ergebnisdaten erlaubt ohne Weiterverarbeitung nur sehr begrenzte Aussagen über die Fehleranfälligkeit der unterschiedlichen Softwarekomponenten.

Um dies zu verdeutlichen ist in Abbildung 1.2 eine Analyse von Ergebnisdaten zu sehen. Für dieses Experiment wurden Fehler in Form eines Burst-Fehlers – also durch Kippen aller Bits des Byte – an der Speicheradresse vorgenommen, auf die während der Laufzeit zugegriffen wird. Dabei wird für jede Fehlerinjektion ein eigenes Experiment durchgeführt, so dass die Fehler sich nicht gegenseitig beeinflussen können. In der Abbildung ist das Ergebnis der Experimente abgebildet. Auf der X-Achse ist die Zeit (in Form von Zyklen) abgebildet. Auf der Y-Achse sind die Speicheradressen abgebildet. Die farbigen Kästchen stellen den Experiment-Ausgang dar. Sofern ein Experiment ohne ungewöhnliche Auswirkungen verlief wird es auch nicht eingefärbt. In dieser Darstellung ist zu erkennen, dass es schwer ist, präzise Aussagen über die Fehleranfälligkeit der Variablen, die auf dem Stack liegen zu, zu treffen. Gerade die Daten, die auf dem Stack liegen sind interessant, wenn man Software mit Fehlertoleranzmaßnahmen härten möchte. Für den Benchmark, der für 1.2 verwendet wurde, betreffen 42.6% der gesamten Fehler den Stack.

Dieses Beispiel zeigt, dass aussagekräftige Analysemethoden für die Daten, die aus den Experimenten resultieren, gebraucht werden. Deshalb liegt der Schwerpunkt dieser Arbeit auf der Untersuchung von Analyseverfahren. Neben den Ergebnisdaten werden zusätzlich die Debug-Informationen, die der Compiler generiert, dazu verwendet um aussagekräftigere Analysen zu ermöglichen. So kann durch Aufzeichnen eines Stacktraces eine Zuordnung der Speicheradressen - und somit der Fehler - zu lokalen Variablen und Methoden erreicht werden.

Ein weiteres Ziel der Arbeit ist es, zu untersuchen, wie die Daten, die aus der Ana-

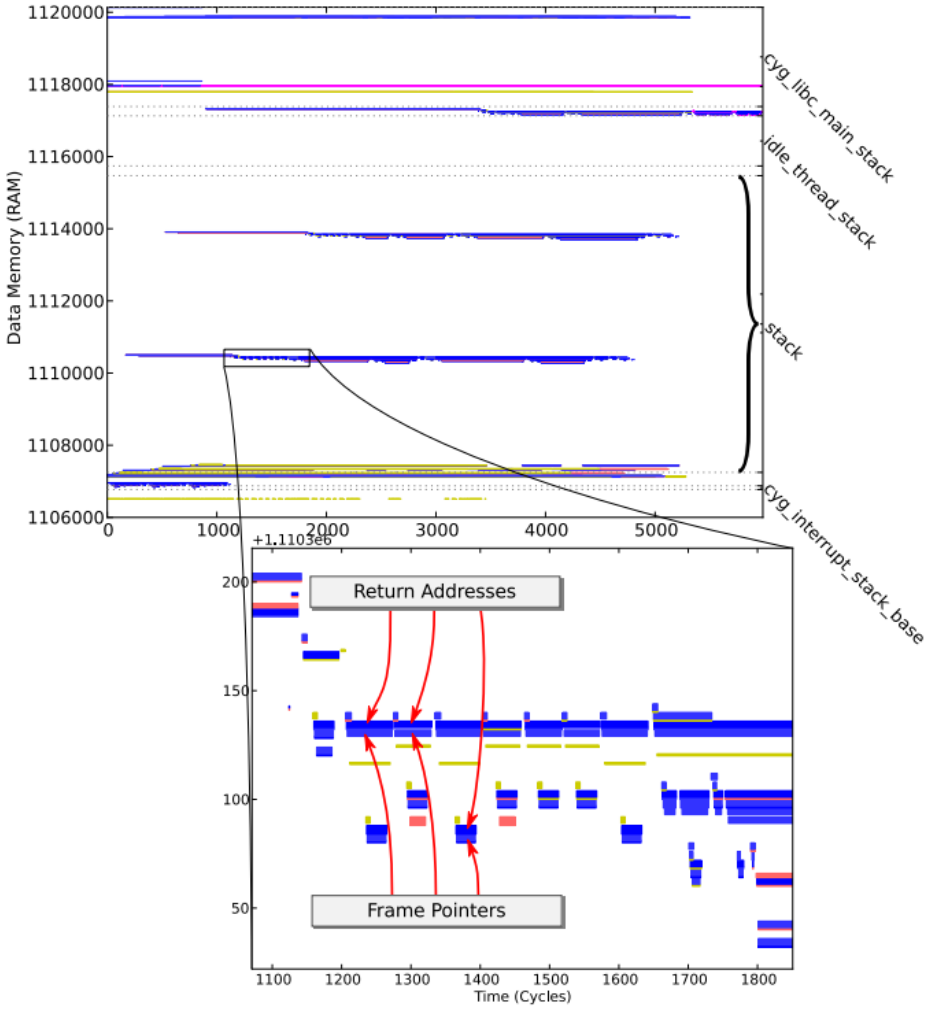

Abbildung 1.2.: Visualisierte Ergebnisdaten von Fehlerinjektions-Experimente (Quelle[5])

lyse resultieren, für eine Automatisierung der Härtung von Software gegen transiente Softwarefehler genutzt werden können. Um dies zu ermöglichen ist es notwendig Aussagen darüber zu treffen, welche Komponenten der Software besonders anfällig gegenüber Fehlerinjektionen sind. Es muss also eine Bewertung der Ergebnisse der Analyse erfolgen.

1.2. Abgrenzung

Zuvor wurden die Fehlertoleranzmaßnahmen angesprochen, die letztendlich dazu verwendet werden, um Bestandteile der Software gegen Fehler zu härten. Die Entwicklung dieser Maßnahmen ist nicht Bestandteil dieser Arbeit. Vielmehr geht es darum die dafür notwendigen Informationen bereitzustellen. Den Fehlerinjektionsexperimenten liegt ein Fehlermodell zugrunde. Das in dieser Arbeit verwendete Fehlermodell betrachtet

ausschließlich flüchtige einzelne Bitfehler im Speicher.

1.3. Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen eingeführt, die als Basis zum Verständnis der restlichen Arbeit dienen. Vorgestellt wird neben dem ELF-Binärformat und dem DWARF-Debugformat, auch wie ein transienter Fehler physikalisch entsteht und wie das Fehlerinjektionsframework FAIL* aufgebaut ist. Kapitel 3 befasst sich mit verwandten Arbeiten aus drei angrenzenden Themengebieten. Zunächst wird auf das Code-Location-Problem eingegangen, da dies in Hinsicht auf unterschiedliche Rückabbildungen für diese Arbeit relevant ist. Weiter werden Analysen aus der bisherigen Forschung bzgl. Fehlerinjektionsexperimenten betrachtet. Abschließend wird auf Literatur eingegangen, die sich mit dem Platzieren von Fehlertoleranzmechanismen befasst. Mit der Analyse und dem konkreten Entwurf befasst sich Kapitel 4. Darin wird über eine Anforderungsanalyse untersucht welche Probleme existieren und entsprechende Lösungen erarbeitet, die abschließend in einem Softwarearchitektur-Entwurf zusammenfließen. Zu dem in Kapitel 4 vorgestellte Entwurf wird in Kapitel 5 eine konkrete Implementierung präsentiert. Wie gut die erarbeiteten Analysen funktionieren und welche Einschränkungen existieren beschreibt Kapitel 6. Abgeschlossen wird diese Arbeite durch ein Fazit und einen Ausblick in Kapitel 7.

2. Grundlagen

Die in diesem Kapitel enthaltenen Information dienen als Grundlage für die folgenden Kapitel. Die Informationen über das ELF-Binärformat und die DWARF-Debuginformationen bilden die Basis für das Kapitel „Analyse und Entwurf “. Der Aufbau von FAIL* bildet das Grundgerüst für die konkrete Implementierung in Kapitel 5.

2.1. Fehler in eingebetteten Systemen

Wie in der Einleitung erläutert, beschränkt sich diese Arbeit auf Hardware-Fehler in Form von kippenden Bits im Speicher. Studien haben gezeigt, dass transiente Fehler, die dieser Art von Fehlern entsprechen für bis zu 28,8 % aller Fehler im Speicher verantwortlich sind [6]. Hardware-Fehler können in unterschiedlichen Ausprägungen auftreten. Entsprechend erfolgt nachfolgend eine genauere Definition.

2.1.1. Klassifikation von Fehlern

Fehler können in unterschiedlicher Art und Weise auftreten. Eine Klassifizierung kann wie in 2.1 abgebildet erfolgen [6, 7]. Dabei werden Fehler zunächst in „Hard faults“ und „Soft faults“unterteilt.

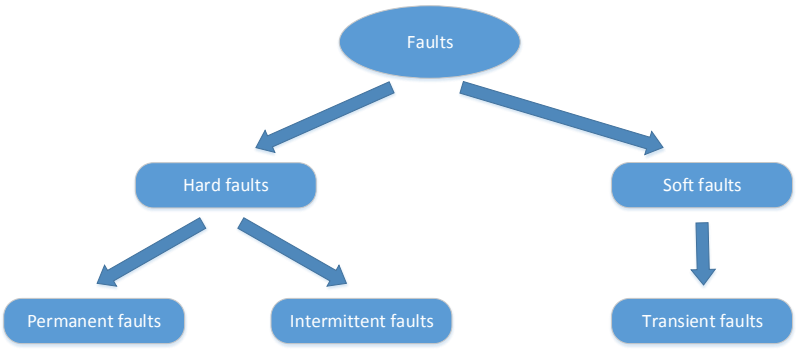

Abbildung 2.1.: Klassifikation von Fehlern im Speicher

„Hard faults“ haben ihren Ursprung im beschädigter Hardware, die einen Austausch erfordert. Darunter kann unterschieden werden, zwischen permanenten und periodischen Fehlern (Intermittent faults). Während die Permanenten immer auftreten, können die periodischen Fehler beispielsweise nur dann auftreten, wenn Temperaturschwankungen

auf den Speicher wirken. Sie hängen also von äußeren Einflüssen ab. Zu den „Soft faults“ gehören die hier als Fehlermodell betrachteten transienten Fehler. Diese transienten Fehler sind flüchtig, d.h. sofern ein Bit im Speicher gekippt ist, ist dieser Fault nach dem Überschreiben der Daten behoben. In den restlichen Kapiteln dieser Arbeit wird lediglich von Fehlern gesprochen. Dabei sind stets transiente Hardware-Fehler gemeint.

2.1.2. Physikalische Sicht

Wenn ein Bit im Speicher kippt, liegt der Grund dafür darin, dass ein Transistor seinen logischen Zustand geändert hat. Diese Änderung kann ungewollt durch hochenergetische Strahlung erfolgen [7]. Es werden drei Arten dieser Strahlung unterschieden [8]:

- **α-Teilchen**

  α-Teilchen entstehen durch den Zerfall von Uran-238 und Thorium-232, die z.B. in geringen Mengen in den Füllstoffen von Quarzen vorkommen. Die Anzahl der Fehler, die durch α-Teilchen entstehen, steigen exponentiell zu den immer kleiner werdenden Strukturgrößen der Chips. Abbildung 2.2 zeigt, wie ein α-Teilchen in einen Transistor einschlägt. Dabei ändert sich die Ladung des Substrats des Transistors. Als Resultat erfolgt eine Änderung des logischen Zustandes des Transistors.

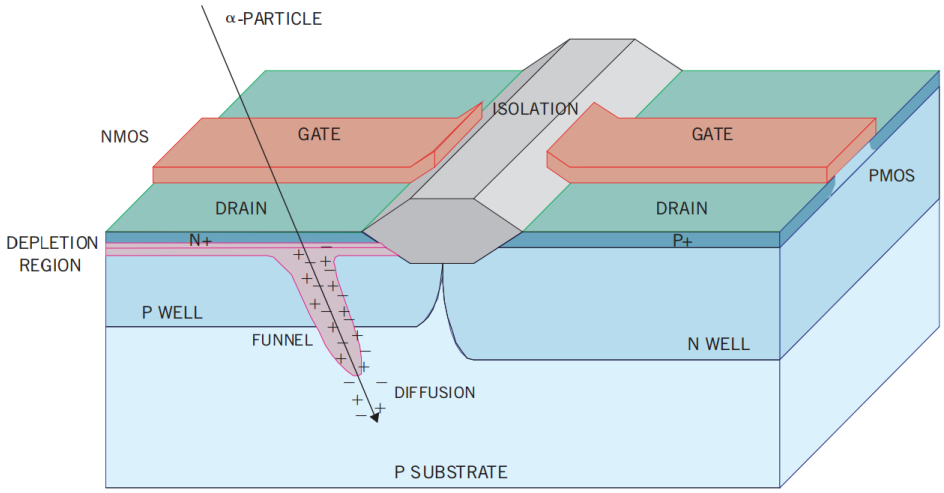

Abbildung 2.2.: Entstehung eines transienten Fehlers innerhalb eines Transistors durch ein α-Teilchen (Quelle: [8])

- **Kosmische Strahlung**

  Es gibt im Weltall unterschiedliche Strahlungen, die durch unsere Atmosphäre größtenteils gefiltert werden. Jedoch entstehen durch Kollisionen von kosmischen Partikeln und der Atmosphäre unserer Erde Neutronen und Protonen mit hoher Energie. Treffen diese auf das Silizium eines Chips kann dies z.B. zu einem Fehler in Form eines Bitflips im Speicher führen. Wie hoch die Gefahr ist, die durch

kosmische Strahlung ausgeht, hängt unter anderem davon ab wo sich das System befindet. So ist die Gefahr eines Fehlers aufgrund kosmischer Strahlung bis zu 800 mal höher, wenn sich das System in einem fliegenden Flugzeug befindet.

- **Thermische Neutronen**

  Thermische Neutronen sind der Ursprung eines Großteils der Fehler in einem System, sofern BPSG (boron-phosphor-silicate-glas) innerhalb des Chips verwendet wurde. Der Grund dafür liegt darin, dass das Boron-10 im BPSG leicht thermische Neutronen aufnimmt und so zu Fehlern führt [8].

2.1.3. Terminologie

Fehler, die in Form eines kippenden Bits im Speicher auftreten, können unterschiedliche Auswirkungen haben. Prinzipiell kann ein Fehler sowohl räumlich, als auch zeitlich propagieren. Die räumliche Dimension kann dabei beispielsweise derart beeinflusst werden, dass sich der Kontrollfluss des Programms ändert und „falsche “ Instruktionen abgearbeitet werden. Zeitlich können Fehler darin resultieren, dass z.B. Geräte des Systems nicht rechtzeitig freigegeben werden und somit Deadlines verpasst werden.

Grundlegend kann man die Auswirkungen eines Fehler, wie in 2.3 dargestellt, in bis zu drei Stadien unterteilen [6]: [1]

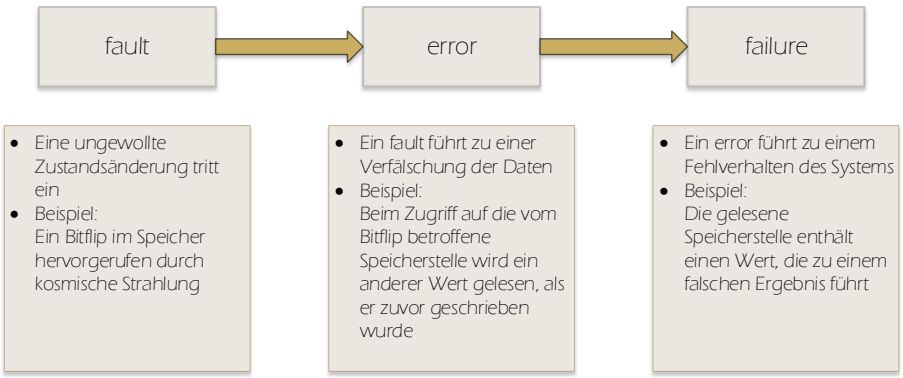

Abbildung 2.3.: Auswirkungen eines Fehlers auf ein System

- **Fault**

  Sobald ein Bit im Speicher kippt liegt ein *Fault* vor. Dieser *Fault* verursacht lediglich eine Zustandsänderung der Daten an der entsprechenden Speicherstelle. Werden die Daten an dieser Speicherstelle überschrieben oder nicht genutzt hat dieser *Fault* keine weiteren Auswirkungen.

[1] Hier wurde zur besseren Unterscheidung absichtlich die englische Bezeichnung gewählt. In der deutschen Sprache heißen alle drei Begriffe „Fehler“.

- **Error**

  Wird die betroffene Speicherstelle gelesen, werden durch den Speicherzugriff fehlerhafte Daten zur Verfügung gestellt. In diesem Fall spricht man von einem *Error*.

- **Failure**

  Verändern die verfälschten Daten, die aus dem *Error* resultieren, das Ergebnis, spricht man von einem *Failure*.

Ein *Fault* oder *Error* muss somit nicht immer zu einem *failure* werden. Für die Untersuchungen im Rahmen dieser Arbeit sind lediglich *failures* relevant, da dadurch der Kontrollfluss oder Ergebnis beeinflusst werden.

Ein Fehlermodell definiert eine Verteilung und die Art und den Ort des Auftretens von Fehlern im Fehlerraum. Beispielsweise kann als Art des Auftretens eines Fehlers das Kippen eines oder mehrerer Bits angenommen werden, wenn man die Auswirkungen kosmischer Strahlung auf den Speicher untersuchen möchte.

Der Fehlerraum spannt einen Raum aus Ort und Zeit auf. Die Zeit wird hier durch die dynamischen Instruktionen dargestellt. Eine dynamische Instruktion kann auf exakt eine statische Instruktion abgebildet werden. Der Ort wird durch die Komponente bestimmt, in die die Fehler injiziert werden. Für Register wären entsprechend genau diese die Ortsangabe im Fehlerraum. Für die in dieser Arbeit betrachteten Fehlerinjektionen in den Speicher stellen die Speicheradressen die Ortsangabe dar.

Um diesen Fehlerraum aufspannen zu können, wird Wissen darüber benötigt, welche dynamischen Instruktionen durchlaufen werden und auf welchen Speicherbereich diese zugreifen. Die Aufzeichnung dieser Informationen, wird als sogenannter *Trace* bezeichnet.

2.2. Fehlerinjektion

Um Fehler innerhalb eines Systems zu untersuchen gibt es zwei Möglichkeiten. Die erste Möglichkeit stellt das *Live-Testing* dar[9]. Beobachtungen von Fehlern und deren Auswirkungen mit Hilfe von *Live-Testing* erfolgen durch Beobachtung des Systems innerhalb eines realen Szenarios. Es erfolgt somit keine direkte Fehlerinjektion. Die größte Schwäche dieses Verfahrens liegt entsprechend darin, dass ein hoher Zeitaufwand investiert werden muss, damit Aussagen über die Anfälligkeit der Software getroffen werden können. Der wesentliche Vorteil dagegen ergibt sich dadurch, dass ein exaktes Fehlermodell entsteht, da lediglich reale und natürlich auftretende Fehler betrachtet werden.

Um dies zu erreichen wird ein sogenannter *Trace* aufgezeichnet, der genau diese Daten enthält. Der *Trace* enthält schließlich Informationen darüber welche Instruktionen ausgeführt wurden und – im Kontext dieser Arbeit – welche Speicherzugriffe stattgefunden haben. Der *Trace* wird meist lediglich über ein bestimmtes Intervall von dynamischen Instruktionen aufgezeichnet, der den zu untersuchenden Programmcode betrifft. Für Endlos-Programme ist dies in jedem Fall notwendig.

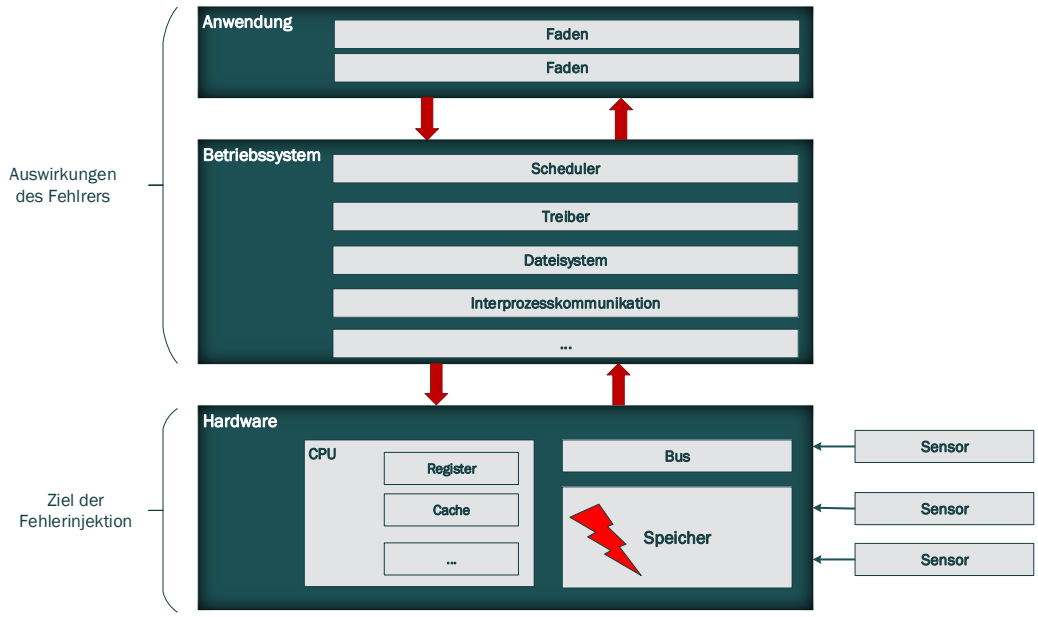

Abbildung 2.4.: Propagation eines Fehlers über die Systemschichten. Die Hardware-Ebene stellt dabei das Ziel der Fehlerinjektion dar. Die Schichten darüber werden dagegen auf Auswirkungen des injizierten Fehlers untersucht.

Die zweite Möglichkeit zur Untersuchung der Auswirkungen von Fehlern besteht darin Fehler vorsätzlich zu injizieren. Dabei kann man im Wesentlichen drei Verfahren zur Injektion unterscheiden[10, 11]:

- **Hardware-basiert**

  Für die Hardware-basierte Fehlerinjektion wird stets zusätzliche Hardware benötigt. Es gibt zwei grundsätzliche Wege Hardware zu stören. Zum einen gibt es die Möglichkeit über Störströme den logischen Pegel am Pin eines Bauteils zu verändern - es wird also ein direkter Kontakt mit den Bauteilen hergestellt. Die zweite Möglichkeit besteht darin z.B. den Speicher störender Strahlung auszusetzen und so Fehler innerhalb der Bauteile zu provozieren. Insbesondere für die zweite Möglichkeit lassen sich keine deterministischen Ergebnisse erzielen.

- **Software-basiert**

  Für die Injektion von Fehlern über Software können zwei Methoden unterschieden werden. Einerseits können Fehler innerhalb des Programmcodes direkt mit einkompiliert werden. Andererseits können Fehler zur Laufzeit eingestreut werden. Dies kann entweder durch Ausnahmebehandlungen erfolgen oder aber durch einen eigenen Faden, der lediglich für die Einstreuung des fehlerhaften Codes zuständig ist. Dabei werden die Fehler durch Manipulation der Datenstrukturen des anderen Fadens injiziert. Beispielsweise ließe sich ein kippendes Bit im Speicher zur

Laufzeit simulieren. Dazu könnte ein Timer verwendet werden, der nach Ablauf einer bestimmten Zeit eine Unterbrechung auslöst, die wiederum den Inhalt einer bestimmten Variable verfälscht.

- **Simulator-basiert**

  Um Fehler innerhalb der simulierten Hardware injizieren zu können muss der Programmcode des Simulators angepasst werden. Dagegen bleibt der Programmcode des Gastsystems - dem Programm, dass der Simulator gerade simuliert - unberührt. Der Simulator-basierte Ansatz bietet gegenüber den anderen beiden Ansätzen mehrere Vorteile. Die Injektion kann zeitlich exakt gesteuert werden. Eine Fehlerinjektion mit dem Simulator-basierten Ansatz ist deterministisch. Ebenso kann der Ort der Injektion beliebig gewählt werden. Voraussetzung dafür ist, dass der Simulator ausreichend detailliert simuliert. Darin liegt auch zugleich einer der Nachteile. Wenn die Simulation die echte Hardware nicht genau genug simuliert kann es zu Abweichungen kommen. Ergebnisse, die aufgrund stark abweichender Simulation zustande gekommen sind, sind entsprechend unbrauchbar für die Wahl der Fehlertoleranzmechanismen zur Härtung der Software.

Die Injektion eines Fehlers erfolgt dabei sowohl für die Hardware-basierten, als auch für die Simulator-basierten Verfahren, grundsätzlich auf der Hardware-Ebene (siehe Abbildung 2.4). Für den Software-basierten Ansatz wird die Injektion in die Hardware auf Ebene der Software simuliert. Für diese Arbeit wird ausschließlich das Simulator-basierte Verfahren betrachtet.

Hierzu wird das Fehlerinjektions-Framework Fail*, dass in Kapitel 2.3 vorgestellt wird, verwendet.

Gegenüber dem Hardware-basierten Ansatz kann beim Simulator-basierten Ansatz absoluter Determinismus angenommen werden.

Wie in **??** dargestellt ist, beziehen eingebettete Systeme häufig zusätzliche Informationen über die Umgebung über Sensoren, die dazu dienen die physikalische Welt abzubilden. Um dennoch vollständigen Determinismus zu erreichen ist es notwendig diese Sensoren ebenfalls zu simulieren.

Um festzustellen, ob eine Abweichung, die durch den injizierten Fehler bedingt ist, eingetreten ist, muss das Experiment den Zustand des Zielsystems mit dem sogenannten *Golden-Run* abgleichen [12]. Der *Golden-Run* ist eine Aufzeichnung des Zustandes des Zielsystems ohne, dass ein Fehler injiziert wurde. Mit Hilfe dieser Aufzeichnung kann eine Abweichung nach einer Fehlerinjektion eindeutig identifiziert werden.

2.2.1. Ablauf

Der grundlegende Ablauf von Fehlerinjektionen ist im Wesentlichen bei allen Verfahren ähnlich [11]. Abbildung 2.5 stellt einen solchen Ablauf dar. Die einzelnen konkreten Fehler, die injiziert werden sollen, werden von einer zentralen Steuer-Einheit - nachfolgend Kampagne genannt [13, 14] - bereitgestellt. Diese leitet den konkreten Fehler an ein Experiment weiter. Dieses Experiment ist in der Lage das Zielsystem zu steuern

und dessen Zustand zu überwachen. Das Experiment kann so beispielsweise ein kippendes Bit im Speicher des Zielsystems verursachen und anschließend die resultierende Zustandsänderung mit Hilfe des *Golden-Run* überwachen.

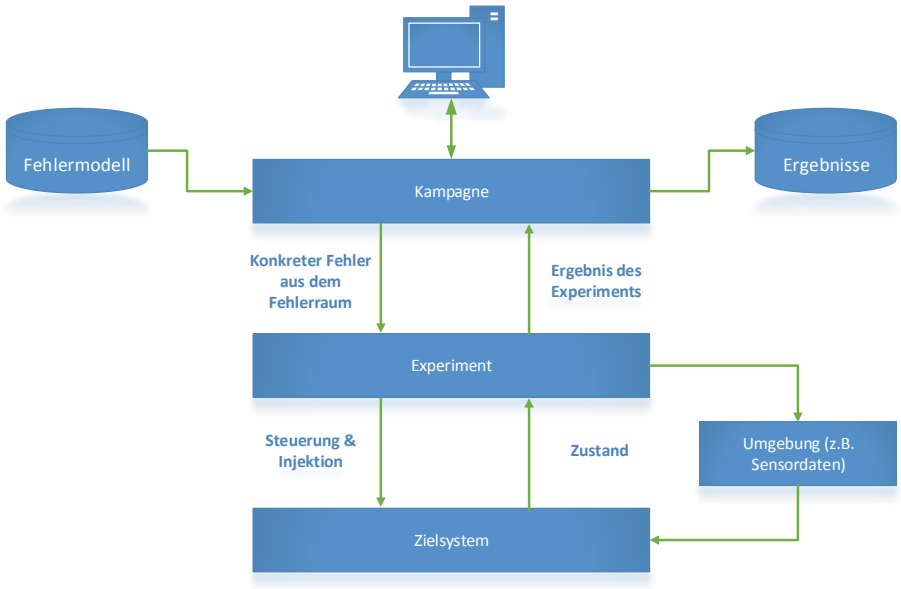

Abbildung 2.5.: Allgemeiner Ablauf einer Fehlerinjektion. (angelehnt an [11])

Zusätzlich hat das Experiment die Kontrolle über die Umgebung (Sensoren) und kann diese entsprechend dem Fehler anpassen. Wie sich der Zustand des Simulators geändert hat, wird letztlich wieder an die zentrale Steuerungs-Einheit übergeben. Diese speichert die Daten und reicht den nächsten zu injizierenden Fehler an das Experiment weiter. Um eine identische Ausgangssituation für alle Experimente zu schaffen, muss das Zielsystem zurückgesetzt werden. Dies kann beispielsweise durch einen Neustart erfolgen.

2.2.2. Optimierung

Der Fehlerraum wächst exponentiell mit der Anzahl der dynamischen Instruktionen und der Speicheradresse, die er umfasst. Somit steigt auch der Aufwand in Form von Experimenten exponentiell. Um diesen entgegen zu wirken, werden Optimierung benötigt welche die Anzahl der notwendigen Experimente verringert. Dabei sollte das Ergebnis der Kampagne möglichst genau bleiben. Um dies zu erreichen, können zwei Ansätze unterschieden werden:

- **Pruning-Verfahren**

  Das Ziel von Pruning-Verfahren ist es die Anzahl der Experimente zu verringern, ohne dabei das Ergebnis der Kampagne zu verändern.

Eine Variante ist das *def/use-Pruning*, bei dem die Speicherzugriffe untersucht werden [10]. In Abbildung 2.6 ist dargestellt, wie das *def/use-Pruning* von Speicherzugriffen funktioniert. Für die nachfolgende Erläuterung wird vereinfacht angenommen, dass Fehler betrachtet werden, für die lediglich ein Bit gekippt wird. Der Fehlerraum wird in zwei Gruppen eingeteilt.

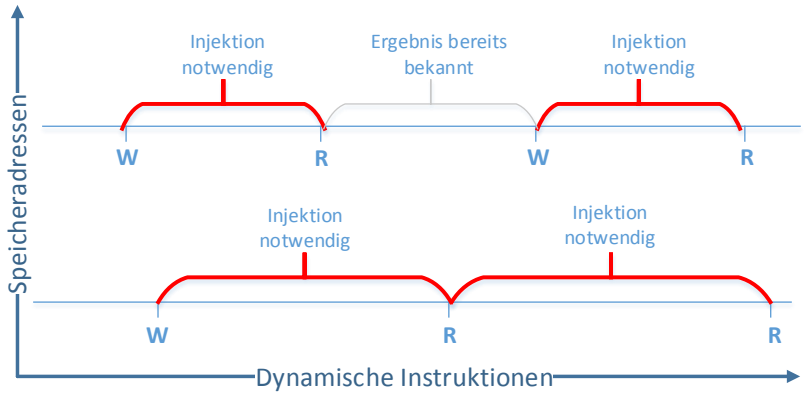

Abbildung 2.6.: Pruning von Speicherzugriffen im Fehlerraum nach [10].

Zum einen gibt es lesende Speicherzugriffe, auf die ein schreibender Speicherzugriff folgt. Für den gesamten Umfang der dynamischen Instruktionen zwischen dem lesenden und dem schreibenden Speicherzugriff muss kein Experiment durchgeführt werden. Der Grund dafür liegt darin, dass die Speicheradresse in jedem Fall als nächstes überschrieben wird bevor sie das nächste Mal gelesen wird. Somit wären Experimente, die sich auf diese Speicheradresse und diesen Bereich der dynamischen Instruktionen beziehen, überflüssig. Zum anderen gibt es den umgekehrten Fall. Es werden also erst Daten an einer Adresse geschrieben und anschließend gelesen. Für den gesamten Bereich der dynamischen Instruktionen zwischen diesen beiden Speicherzugriffen muss lediglich ein einziges Experiment durchgeführt werden, da angenommen werden kann, dass es irrelevant ist, wann zwischen den beiden Speicherzugriffen der Fehler injiziert wird.

Durch dieses Verfahren lässt sich die Anzahl der Experimente unter den richtigen Umständen erheblich reduzieren ohne, dass das Ergebnis an Aussagekraft verliert.

Sofern in der nachfolgenden Kapiteln von Pruning gesprochen wird, ist stets das *def/use-Pruning* damit gemeint.

- **Statistik-Verfahren**

  Bei den Statistik-Verfahren wird die Anzahl der notwendigen Experimente, auf Basis statistischer Merkmale der Fehler aus dem Fehlermodell, errechnet[15]. Dieses Vorgehen wird *Sampling* genannt. Um eine Abschätzung der benötigten Experimente vorzunehmen zu können wird angenommen, dass die Menge der Fehler

im Fehlermodell einer Normalverteilung unterliegt. Des Weiteren muss die Wahrscheinlichkeit, dass ein Fehler auftritt für alle Fehler aus dem Fehlerrraum gleich sein. Das Auftreten von Fehler muss also einer Gleichverteilung unterliegen. Beide Voraussetzungen wären für ein Fehlermodell, dass kippende Bits im Speicher betrachtet, erfüllt.

Der Nachteil eines solchen Vorgehens liegt darin, dass lediglich grobe Abschätzungen über die Fehleranfälligkeit unterschiedlicher Softwarekomponenten vorgenommen werden können, da ein Großteil der Experimente nicht durchgeführt wird. So ist es mit diesem Verfahren unmöglich eine Aussage darüber zu treffen, ob eine bestimmte lokale Variable einer Funktion besonders fehleranfällig ist. Entsprechend können keine Fehlertoleranzmechanismen entwickelt werden, die genau diese Softwarekomponente schützen.

Neben der Möglichkeit durch Pruning die Anzahl der notwendigen Experimente zu verringern kann auch die Dauer eines einzelnen Experimentes verringert werden. Notwendig ist dazu die Möglichkeit einen Zustand im Zielsystem sowohl sichern, als auch wiederherstellen zu können. In vielen Fällen liegt das Interesse lediglich bei Teilkomponenten der Software. Im Rahmen des *Golden-Run* kann der Zustand des Zielsystems an der Stelle, welcher das Interesse gilt, gesichert werden. Für die einzelnen Experimente kann so vor jeder neuen Fehlerinjektion der gesicherte Zustand wiederhergestellt werden und so der Programmcode davor übersprungen werden. Damit sich diese Optimierung wirklich vorteilhaft auf die benötigte Zeit, die zur Ausführung der gesamten Kampagne benötigt wird, auswirkt, muss das Wiederherstellen schneller erfolgen, als das Zielsystem für einen Neustart braucht.

Neben den hier genannten Verfahren, existieren weitere Verfahren zur Optimierung, auf die in dieser Arbeit nicht weiter eingegangen wird, da sie nicht relevant sind.

2.3. Fehlerinjektionsexperimenten mit FAIL*

Bei *FAIL** [2] handelt es sich um ein Fehlerinjektions-Framework, das im Rahmen des Forschungsprojektes *DanceOS* entstanden ist[4]. Im Unterschied zu anderen Fehlerinjektions-Frameworks ist *FAIL** flexibel anpassbar. Diese Anpassbarkeit zeichnet sich insbesondere durch folgende Eigenschaften aus:

- **Generische API / Generisches Backend**

  Der Simulator, der zur Fehlerinjektion genutzt wird, ist mit geringem Aufwand austauschbar. Dies wird insbesondere dadurch erreicht, dass die Schnittstellen, die von *FAIL** für Experimente bereitstellt werden, generisch geschrieben sind.

[2] *FAIL** steht für „FAult Injection Leveraged“. Der * steht für das generische Backend, dass mit dem Framework verbunden werden kann.[4]

- **Generische Experimente**

  Durch die generische API sind Fehlerinjektions-Experimente für jedes Backend gleich. Durch die Wiederverwendbarkeit der Experimente muss das Fehlermodell lediglich einmal formuliert werden, um es auf unterschiedlichen Architekturen (Simulatoren) zu verwenden.

- **Hohe Effizienz**

  *FAIL** ist nach einer Server-Client Architektur aufgebaut. Die Experimente, die mit Hilfe des Simulators durchgeführt werden (Client), sind somit in Hinblick auf die Kommunikation mit der Kampagne (Server) vollständig entkoppelt. Als Resultat dieser Architektur können Fehlermodelle durch die Verwendung mehrerer Clients beliebig parallelisiert werden. Neben der Parallelität wird durch das in Kapitel 2.2.2 vorgestellt Pruning-Verfahren eine deutliche Reduktion der notwendigen Experimente erreicht.

2.3.1. Struktur

Der strukturelle Aufbau von des FAIL*-Frameworks ist in Abbildung 2.7 dargestellt. Der grundlegende Aufbau gleicht dem allgemeinen Ansatz, der in 2.2.1 vorgestellt wurde. Die Kampagne verwaltet die Kommunikation und Parameter, die konkrete zu injizierende Fehler beschreiben. Die Parameter werden von dem Fehlerinjektions-Experiment entgegengenommen. Das Experiment hat durch die generische Simulator-Abstraktion Zugriff auf den gesamten Zustand des Simulators.

Zur Anbindung des Simulators an das FAIL*-Framework wird aspektorientierte Programmierung - hier konkret AspectC++[17][18] - verwendet. Durch den Einsatz aspektorientierter Programmierung wird eine Trennung der Belange erreicht. So kann hier die Anbindung des Simulators an das FAIL*-Framework erfolgen, ohne wesentliche Änderungen am Code des Simulators vorzunehmen. Zum Zeitpunkt der Erstellung dieser Arbeit ist der x86-Simulator Bochs[19] vollständig an FAIL* angebunden. Der Simulator Gem5[20], der primär zur Simulation der ARM-Architektur eingesetzt wird[3], befindet sich derzeit noch in der Anbindungsphase.

Neben dem eigentlichen Experiment ist es mögliche sogenannte *Plugins* einzuschalten. Plugins können genau wie Experimente auf den Zustand des Simulators zugreifen. Sie dienen beispielsweise dazu, die Funktionalität einen Trace aufzuzeichnen, zu kapseln.

2.3.2. Experiment-Ablauf

Der Simulator und das Experiment laufen in zwei unterschiedlichen Koroutinen. Dies ermöglicht es, dass entweder das Experiment oder aber der Simulator die Kontrolle hat. Um auf Ereignisse und Zustandsänderungen reagieren zu können, werden durch das

[3] Der Simulator Gem5 kann folgende Architekturen simulieren: ARM, ALPHA, MIPS, Power, SPARC, and x86. Im Rahmen des DanceOS Projektes wird derzeit Gem5 als Simulator für die ARM-Architektur eingesetzt.

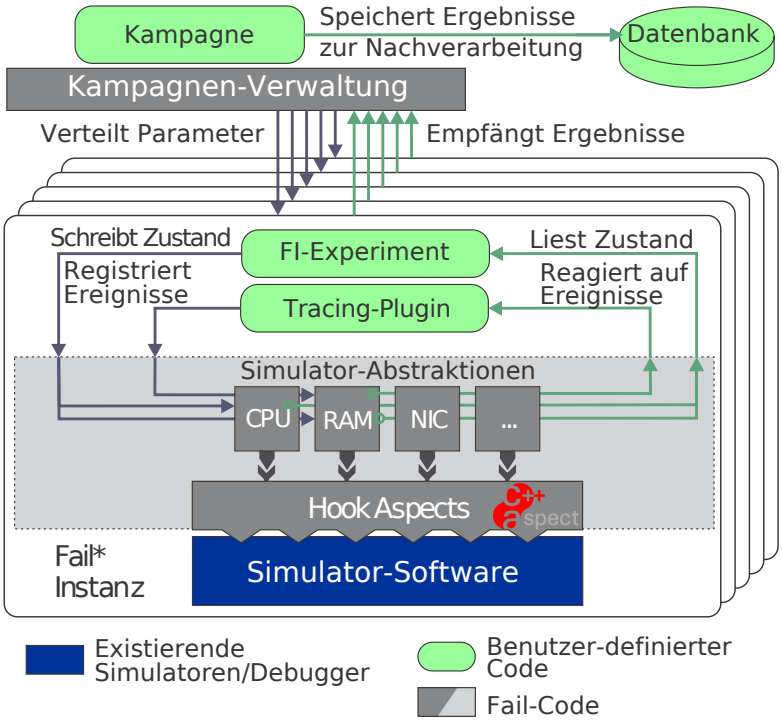

Abbildung 2.7.: Struktur und Aufbau des Fehlerinjektion-Framework FAIL* (Quelle: mit geringfügigen Änderungen übernommen aus [16].)

FAIL*-Framework sogenannte *Listener* zur Verfügung gestellt. Über diese Listener kann ein Experiment auf bestimmte Ereignisse oder Zustandsänderungen warten. Folgende *Listener* sind zum Zeitpunkt der Erstellung dieser Arbeit vorgesehen:

- **Breakpoint / BreakpointRange**

  Breakpoint-Listener können analog zu Breakpoints in einem Debugger gesetzt werden. Sobald die entsprechende Instruktion erreicht wurde, wird dies an das Experiment gemeldet. Außerdem kann ein Zähler definiert werden, der nach der Abarbeitung der angegebenen Anzahl Instruktionen dies an das Experiment meldet.

- **MemoryAccess**

  Mit Hilfe der MemoryAccess-Listener werden Speicherzugriffe registriert. Dabei können sowohl schreibende, als auch lesende Zugriffe registriert werden. Die MemoryAccess-Listener stellen eine signifikante Komponente des FAIL*-Frameworks dar, da so über aufgezeichnete Speicherzugriffe das *Pruning* ermöglicht wird.

- **Trouble**

  Der Trouble-Listener fast eigentlich zwei Listener zusammen. Zum einen können damit Interrupts registriert werde. Zum anderen kann damit auch auf Traps gelauscht werden.

- **Jump**

  Um Sprünge im Programmcode erkennen zu können, werden Jump-Listener verwendet.

- **Timer**

  Timer-Listener erzeugen ein künstlicher Ereignis, sobald eine definierte Zeit abgelaufen ist. Nützlich ist solch ein Listener, um festzustellen, ob das Gastsystem aufgrund einer Fehlerinjektion länger arbeitet, als es beim Golden-Run der Fall war.

- **Guest**

  Unter Umständen möchte man eine direkte Kommunikation zwischen Experiment und dem Gast-System, das im Simulator läuft, herstellen. Dazu kann auf Ereignisse auf der seriellen Schnittstelle des Simulators gelauscht werden.

Neben dieser Möglichkeit auf bestimmte Ereignisse zu warten kann über die von FAIL* bereitgestellte Schnittstelle, der Zustand des Simulators manipuliert werden. Dabei sind folgende Möglichkeiten zur Manipulation vorgesehen:

- **Speicher**

  Es kann zu jedem Zeitpunkt der Speicher an einer beliebigen Adresse manipuliert werden. Dem Fehlermodell entsprechend können so beispielsweise Einzelbit-Fehler oder Mehrbit-Fehler gesetzt werden.

- **Register**

  Analog zum Speicher lassen sich die Registerinhalte jederzeit auslesen und verfälschen.

- **Interrupt / Traps**

  Neben der Möglichkeit durch Trouble-Listener auf Interrupts und Traps zu lauschen können diese auch künstlich erzeugt werden.

- **Zustand Speichern und wiederherstellen**

  Ein weiterer wesentlicher Bestandteil des FAIL*-Frameworks stellt die Möglichkeit dar, den Zustand des Simulators zur Laufzeit zu speichern und wiederherzustellen. Somit kann eine weitere Optimierung aus 2.2.2 durch FAIL* genutzt werden.

- **Neustart**

  Ebenso lässt sich der Simulator jederzeit neu starten.

Mit Hilfe dieser *Listener* wird der gesamte Experiment-Ablauf beschrieben. Am Anfang eines Experimentes werden die Fehlerinjektions-Daten vom Kampagnen-Server geholt. Anschließend wird mit Hilfe der Listener der Fehler injiziert. Am Ende des Experimentes erfolgt die Nachuntersuchung. In dieser Nachuntersuchung wird untersucht, wie das Experiment ausgegangen ist. Abschließend werden die ERgebnis-Daten zurück zum Kampagnen-Server geschickt.

2.3.3. Kampagnen-Ablauf

Damit die Kampagne die Fehler-Parameter an die Experimente verteilen kann, müssen zunächst Informationen über das Fehlermodell, den Fehlerraum und die Speicherzugriffe im Programm gesammelt werden. Abbildung 2.8 stellt den allgemeinen Fluss der Daten und deren Verarbeitung dar. Zunächst wird ein Fehlermodell benötigt. Das Wissen über dieses Fehlermodell stammt vom Entwickler.

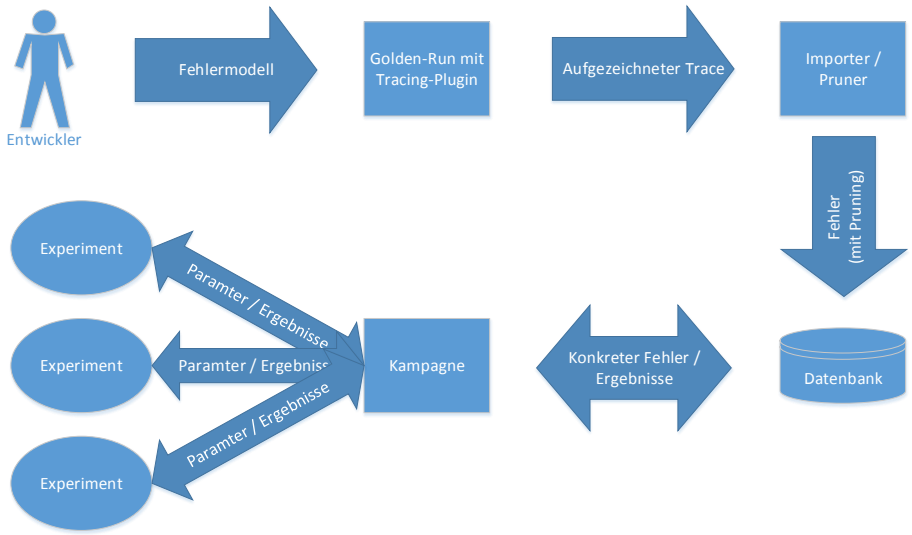

Abbildung 2.8.: Allgemeiner Ablauf einer Kampagne mit notwendiger Vorbereitung

Mit diesen Informationen wird über das Tracing-Pugin ein entsprechender *Trace* aufgezeichnet. Hier bedeutet dies, dass wir sämtliche dynamischen Instruktionen und dazugehörige Speicherzugriffe aufzeichnen. Der Trace wird anschließend von einem Importer bzw. Pruner verarbeitet. Importer/Pruner sind Tools, die aufgezeichnete Traces entsprechend dem Fehlermodell in die Datenbank importieren. Zusätzlich zum Import der Daten können die Daten durch den Pruner optimiert werden. Für diese Arbeit bedeutet dies konkret, das ein *Memory-Importer* sowie ein Pruner verwendet wird, der auf Basis der Daten aus dem Trace, die Pruning-Optimierung durchführt und die daraus resultierenden zu injizierenden Fehler in die Datenbank schreibt.

Die Kampagne ist als zentraler Server konzipiert, der die Daten auf Nachfrage von den Experimenten (Clients) versendet und die Ergebnisdaten entgegen nimmt. Jedes Ergebnis wird anschließend wieder zurück in die Datenbank geschrieben.

2.3.4. Ergebnisdaten

Ein wesentlicher Bestandteil dieser Arbeit besteht darin mit den Daten aus Datenbank neue Analysemöglichkeiten zu entwickeln. Im Folgenden wird erklärt, wie die Datenbank aufgebaut ist und welche Aussagekraft die Daten ohne weitere Analyse haben.

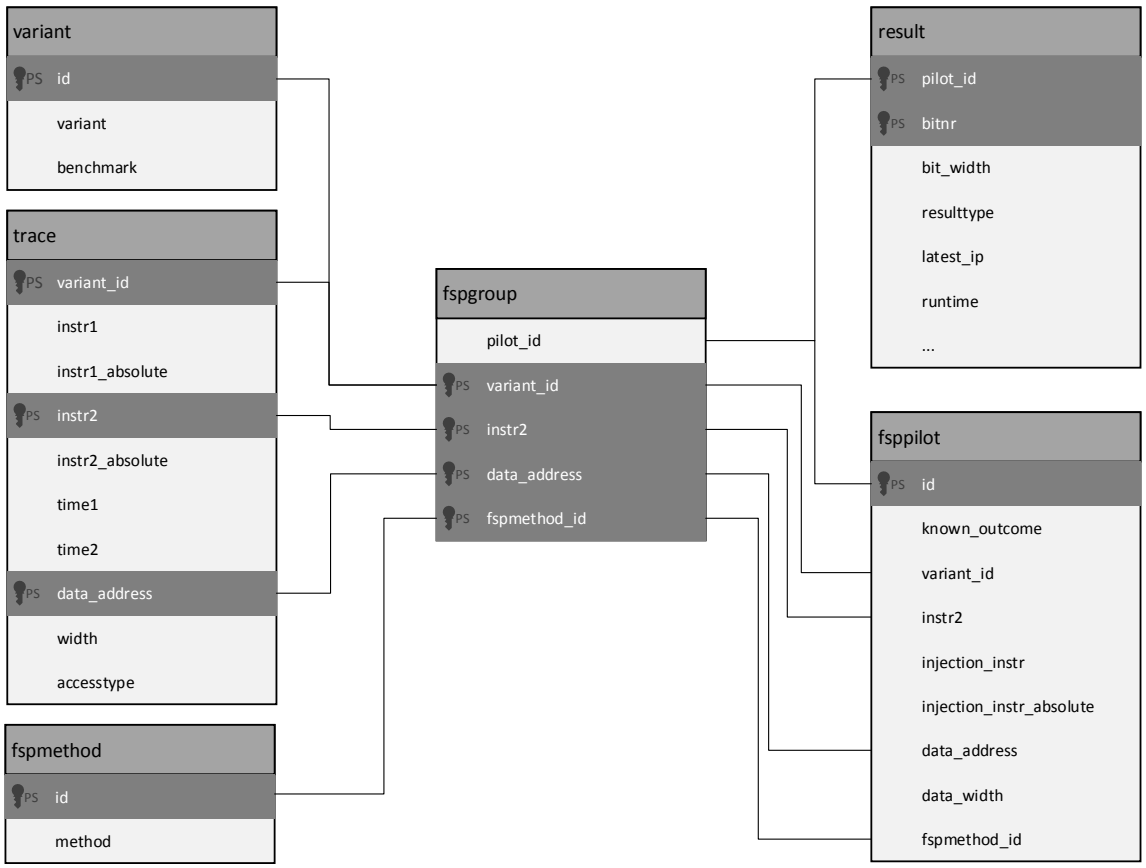

Abbildung 2.9.: Schema der Datenbank für die Parameter und Resultate der Fehlerinjektionen

Abbildung 2.9 stellt die komplette Datenbankstruktur wie sie von FAIL* verwendet wird, dar. Nachfolgend wird jede Tabelle erläutert:

- **variant**

  Häufig werden mehrere Benchmarks nacheinander auf ihre Fehleranfälligkeit untersucht. Zu jedem Benchmark können unterschiedliche Varianten existieren, welche unterschiedliche Fehlertoleranzmechanismen implementieren. Insbesondere für Vergleichszwecke ist diese Möglichkeit der Unterscheidung sinnvoll. Damit jeder Benchmark mit seinen unterschiedlichen Varianten eindeutig identifiziert werden kann, werden diese durchnummeriert (id).

- **trace**

  Wie zuvor beschrieben werden die Daten aus dem *Trace* über einen Importer und Pruner in die Datenbank importiert, die ein *Pruning* durchführen. In der Tabelle *trace* stehen die Bereiche, die aus dem Pruning resultieren. Die Spalten instr1 sowie instr2 decken genau den Bereich zwischen zwei Speicherzugriffen ab, wobei

instr1 die erste Instruktion nach einem Speicherzugriff ist und instr2 genau die Instruktion darstellt, die den nächsten Speicherzugriff bewirkt. Zusätzlich zu instr1 und instr2, die eine Zahl dynamischer Instruktionen seit Aufzeichnungsbeginn des Trace wiedergeben, wird ebenfalls die statische absolute Instruktionsadresse hinterlegt (instr1_absolute, instr2_absolute). Zusätzlich wir die Zeit gemessen, die vergeht während der Bereich des Programm abgearbeitet wird. time1 bzw. time2 repräsentieren den Zeitpunkt, an dem instr1 bzw. instr2 abgearbeitet wurde. Neben diesen Informationen zu dem Bereich, werden die Speicheradresse selbst, die Breite und die Zugriffsart in der Tabelle trace festgehalten.

- **fspmethod**

  Sollten unterschiedliche Pruning-Verfahren für unterschiedliche Benchmarks genutzt werden, kann dies in der Tabelle fspmethod unterschieden werden.

- **fsppilot**

  Die Tabelle fsppilot enthält eine Teilmenge der Einträge aus der trace-Tabelle. Alle Einträge in dieser Tabelle stellen einen konkreten zu injizierenden Fehler dar. Sie enthält alle Einträge aus der Trace-Tabelle, die sich auf einen lesenden Speicherzugriff beziehen. Wie zuvor beschrieben, sind genau das die interessanten Intervalle, bei denen unklar ist, wie es im weiteren Programmverlauf weitergeht bzw. wie es ausgeht. Zusätzlich zu diesen Einträgen ist ein Eintrag aus der Trace-Tabelle enthalten, der sich auf einen schreibenden Speicherzugriff bezieht und so alle anderen schreibenden Speicherzugriffe repräsentiert. Für diesen einen Eintrag wird die Spalte known_outcome auf true gesetzt, da der Ausgang des Experimentes bereits bekannt ist und sich nicht vom Golden-Run unterscheiden sollte. injection_instr und injection_inst_absolute enthalten die dynamische und statische Adresse der Instruktion, bei der der Fehler injiziert wird.

- **result**

  In der Tabelle result werden die Ergebnisse der Fehlerinjektionen gesammelt. Wie die Tabelle konkret gestaltet ist hängt vom durchzuführenden Experiment ab. In Abbildung 2.9 sind lediglich Spalten angedeutet, die häufig in einer result-Tabelle vorkommen. Neben des konkreten Bits (bitnr), in das der Fehler injiziert wurde und der gesamten Breite des Speicherzugriffs (bit_width) werden zusätzliche Informationen gespeichert. Die wichtigste Information stellt dabei der resulttype dar. Meist wird hier durch eine Zahl vermerkt, wie das Experiment ausgegangen ist. Zusätzlich kann es für spätere Analysem nützlich sein die Instruktionsadresse aufzuzeichnen, die zuletzt abgearbeitet wurde bevor die Kontrolle zum Experiment wechselt. Auch die Laufzeit wird häufig mit aufgezeichnet (runtime). Neben diesen Informationen können beliebige weitere Informationen hinterlegt werden.

- **fspgroup**

  fspgroup ist die Tabelle, in der alle Informationen zusammenlaufen. Für weitere Analysen ist diese Tabelle ein geeigneter Ansatzpunkt, da zu jeder Variante mit ei-

ner bestimmten Injektionsadresse und einer bestimmten Speicheradresse konkrete Experimente inklusive der Ergebnisse zugeordnet werden können.

Vollkommen unabhängig davon, wie viele Informationen in der result-Tabelle hinterlegt sind, können immer nur Aussagen auf der Ebene einzelner dynamische Instruktionen bzw. konkreten Speicheradressen getroffen werden. ES sind also allein mit diesen Daten keine Aussagen über die Fehleranfälligkeit von Datenstrukturen möglich, die im Stack-Segment liegen.

2.4. Das Binärformat ELF

Das *Executable and linkable format (ELF)* ist ein Binärformat, dass hauptsächlich unter Unix-ähnlichen Betriebssystemen eingesetzt wird[21]. Der wesentliche Vorteil von ELF gegenüber anderen Binärformaten besteht darin, dass das ELF-Binärformat für unterschiedliche Architekturen eingesetzt werden kann. Damit ist nicht gemeint, dass ein und die selbe Binärdatei auf unterschiedlichen Architekturen ausgeführt werden kann[4], sondern lediglich, dass die Struktur und der Aufbau des Formates der Binärdatei gleich ist. Im Rahmen dieser Arbeit ist es lediglich notwendig die allgemeine Struktur des ELF-Binärformates zu kennen. Das ELF-Binärformat kann für folgende Dateiarten eingesetzt werden:

- **Executable**

  Entspricht einem ausführbaren Programm. Es enthält alle Informationen, die ein Betriebssystem braucht, um einen neuen Prozess für die Binärdatei anzulegen und entsprechenden Speicher zu allozieren und zu befüllen.

- **Relocatable**

  Entspricht einer Objektdatei. Zusammen mit anderen Relocatable Elf-Binärdateien entsteht eine statische Bibliothek. Alternativ kann es mit einer Executable Elf-Binärdatei gelinkt werden.

- **Shared Object**

  Shared Objekt Binärdateien können entweder mit anderen Shared Object Binärdateien oder Relocatable Binärdateien gelinkt werden um eine Objektdatei (Relocatable) zu erstellen oder es wird mit einer ausführbaren (Executable) Binärdatei gelinkt, um eine ausführbare Datei zu erzeugen. Shared Object Binärdateien entsprechen dynamischen Bibliotheken.

2.4.1. Aufbau

In Abbildung 2.4.1 ist der strukturelle Aufbau einer ELF-Binärdatei zu sehen. Dabei wird zwischen der Linkersicht und der Ausführungssicht unterschieden.

[4]Eine Unterstützung für mehrere Architekturen in einer Binärdatei bietet beispielsweise das FatELF-Format.

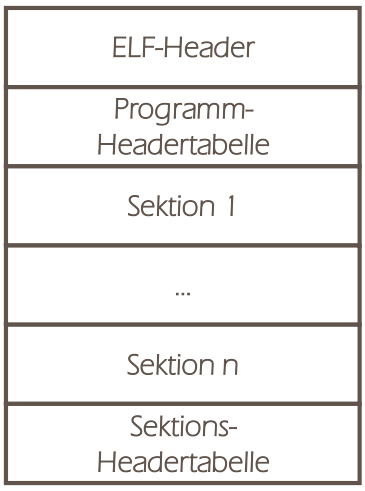

Abbildung 2.10.: Struktur einer ELF-Binärdatei aus der Sicht des Linkers (angelehnt an: [21])

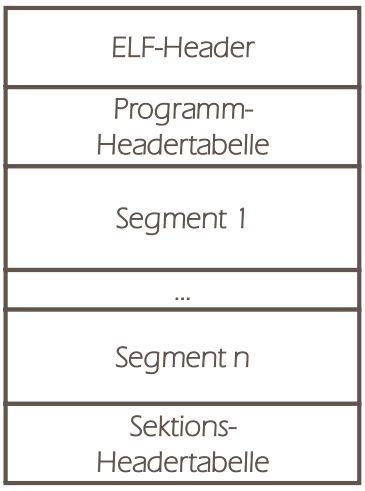

Abbildung 2.11.: Struktur einer ELF-Binärdatei in der Ausführungssicht (angelehnt an: [21])

Der ELF-Header enthält allgemeine Informationen über die Binärdatei. Dazu zählen Informationen wie die Magic Number[5], Zielarchitektur, Version des ELF-Formates[6], ob es sich um eine 32-Bit oder 64-Bit Architektur handelt, um welchen Typ (Executable / Relocatable / Shared Object) es sich handelt und unterschiedliche Informationen wie Größen und Anzahl der Sektionen sowie den Einsprungpunkt.

Die Programmheader-Tabelle ist nur in Binärdateien vom Typ Executable enthalten. Diese Tabelle beschreibt, wie sich das Programm im Speicher zusammensetzt. Dazu werden Abschnitte definiert und zu jedem Abschnitt eine virtuelle sowie physikalische Adresse[7] und die Größe des Abschnittes angegeben. Ebenso werden Rechte angegeben, die z.B. zeigen, dass der Inhalt eines Abschnittes ausgeführt werden darf. Für die Abschnitte gibt es unterschiedliche Typen, die hier nur beispielhaft aufgeführt werden:

- **PHDR**

  Ein Abschnitt mit dem Typen PHDR beschreibt die Programm-Headertabelle selbst.

- **INTERP**

  Der Typ INTERP zeigt an, welcher Interpreter aufgerufen werden muss, um unaufgelöste Referenzen im Programmcode aufzulösen. Dazu zählt beispielsweise die

[5] Jede ELF-Datei kann eindeutig als solche identifiziert werden, da sie immer mit den Bytes 0x7f 0x45 0x4c 0x46 beginnt. Die letzten drei Bytes stellen dabei den ASCII-Code für ELF dar.

[6] Version 1 ist heute noch aktuell.

[7] Die physikalische Adresse ist nur relevant sofern keine Speicherverwaltung durch das Betriebssystem stattfindet.

Bibliothek */lib/ld-linux.so*, die dazu verwendet wird um benötigte dynamische Bibliotheken im Speicher einzublenden.

- **LOAD**

  Der Typ LOAD gibt an, dass der Abschnitt aus der Binärdatei in den Speicher eingeblendet werden soll.

- **DYNAMIC**

  Der Abschnitt mit dem Typen DYNAMIC enthält Informationen, die vom Interpreter aus dem Abschnitt INTERP benötigt werden.

- **NOTE**

  Dieser Abschnitt kann herstellerspezifische Informationen enthalten

Neben den hier genannten Typen kann es für unterschiedliche Architekturen zusätzliche Typen geben. Die definierten Abschnitte setzen sich letztlich aus unterschiedlichen Sektionen zusammen, die in der ELF-Binärdatei enthalten sind. Eine entsprechende Zuordnung ist ebenfalls in der Programm-Headertabelle angegeben.

Sektionen enthalten Daten, die entweder für das Programm relevant oder selbst Bestandteil des Programms sind. Um die Sektionen zu verwalten, gibt es die Sektions-Headertabelle. Ähnlich wie in der Programm-Headertabelle enthält sie für jede Sektion die Informationen, Name, Typ, Adresse, Offset, Größe und ein Flag zur Beschreibung des Zwecks[8].Die wichtigsten Typen sind Folgende:

- **PROGBITS**

  Enthält Binärcode, der für die Ausführung des Programms bestimmt ist. Darunter zählen z.B. die Instruktionen sowie globale Variablen.

- **SYMTAB**

  Wie der Name schon andeutet, enthält die Sektion eine Symboltabelle

- **STRTAB**

  Enthält Strings, die für das eigentliche Programm unbedeutend, aber für das ELF-Format wichtig sind. Dazu zählt z.B. die symbolische Bezeichnung der einzelnen Sektionen wie .text.

- **REL**

  In Sektionen von Typ REL sind Relokations-Informationen hinterlegt.

Neben den hier genannten Informationen kann eine ELF-Binärdatei auch Debug-Informationen enthalten. Diese Debug-Informationen werden in zusätzlichen Sektionen in der Binärdatei untergebracht.

[8] z.B. das Flag A, um anzuzeigen, dass die Daten beim Laden der Binärdatei in den virtuellen Adressraum kopiert werden soll.

2.5. Debug-Informationen

Debug-Informationen können vom Compiler generiert werden. Sie enthalten Informationen, die ein Debugger benötigt, um zur Laufzeit eine Möglichkeit zu bieten das Programm zu untersuchen. Es gibt eine Vielzahl von Debug-Formaten wie z.B. STABS, COFF, PECOFF, OMF und IEEE695. Allerdings ist lediglich das Debug-Format DWARF weit verbreitet und bei vielen Compilern das standardmäßig eingesetzte Debug-Format. Aus diesem Grund werden in dieser Arbeit stets Debug-Informationen im DWARF-Format betrachtet.

2.5.1. DWARF

DWARF wurde im Zuge der Entwicklung des ELF-Binärformates spezifiziert und ist somit kompatibel zu diesem. DWARF kann jedoch auch mit anderen Binärformaten eingesetzt werden.

In Kombination mit einer ELF-Binärdatei werden die Debug-Informationen als zusätzliche Sektionen eingebunden. Um das Verständnis zu erleichtern und den Aufbau des DWARF-Formates anschaulich darstellen zu, können wird folgendes Beispiel betrachtet:

Listing 2.1: Beispielcode

```
1  inline bool checkAlter (int alter) {
2      if (alter > 0 && alter < 150) {
3          return true;
4      } else {
5          return false;
6      }
7  }
8
9  class Person
10 {
11     private:
12     int m_alter;
13
14     public:
15     void setAlter(int neu_alter) { if(checkAlter(neu_alter)) {m_alter =
           neu_alter;} };
16     int getAlter() { return m_alter; };
17 };
18
19 int main (int argc, char* argv[])
20 {
21     Person myPerson;
22     myPerson.setAlter(34);
23
24     return myPerson.getAlter();
25 }
```

In der Funktion main wird eine Person myPerson angelegt. Auf dem Objekt myPerson wird die Funktion setAlter aufgerufen. Diese Funktion speichert das Alter in der privaten

Member-Variable m_alter. Bedingung dafür ist, dass das Alter zwischen 0 und 150 liegt. Die Überprüfung wird durchgeführt durch einen Aufruf der checkAlter-Funktion. Zu beachten ist dabei das Schlüsselwort *inline*. Es deutet an, dass die Funktion an alle Stellen kopiert werden kann, an der diese Methode aufgerufen wird. Das Programm beendet sich mit der Rückgabe des gespeicherten Alters.

Betrachtet wird zunächst wie sich die Debug-Informationen[9] auf die Struktur der ELF-Binärdatei auswirkt. Hierzu wird das Tool *readelf* benutzt. Es erlaubt sämtliche Informationen einer ELF-Binärdatei auszulesen.

```
$ readelf -S beispiel
There are 38 section headers, starting at offset 0xeb0:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .hash             HASH            0804818c 00018c 000030 04   A  6   0  4
  [ 5] .gnu.hash         GNU_HASH        080481bc 0001bc 000020 04   A  6   0  4
  [ 6] .dynsym           DYNSYM          080481dc 0001dc 000070 10   A  7   1  4
  [ 7] .dynstr           STRTAB          0804824c 00024c 0000b6 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          08048302 000302 00000e 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         08048310 000310 000020 00   A  7   1  4
  [10] .rel.dyn          REL             08048330 000330 000008 08   A  6   0  4
  [11] .rel.plt          REL             08048338 000338 000010 08   A  6  13  4
  [12] .init             PROGBITS        08048348 000348 000026 00  AX  0   0  4
  [13] .plt              PROGBITS        08048370 000370 000030 04  AX  0   0 16
  [14] .text             PROGBITS        080483a0 0003a0 0001e0 00  AX  0   0 16
  [15] .fini             PROGBITS        08048580 000580 000017 00  AX  0   0  4
  [16] .rodata           PROGBITS        08048598 000598 000008 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        080485a0 0005a0 000034 00   A  0   0  4
  [18] .eh_frame         PROGBITS        080485d4 0005d4 0000c0 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      08049694 000694 000004 00  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      08049698 000698 000004 00  WA  0   0  4
  [21] .jcr              PROGBITS        0804969c 00069c 000004 00  WA  0   0  4
  [22] .dynamic          DYNAMIC         080496a0 0006a0 000108 08  WA  7   0  4
  [23] .got              PROGBITS        080497a8 0007a8 000004 04  WA  0   0  4
  [24] .got.plt          PROGBITS        080497ac 0007ac 000014 04  WA  0   0  4
  [25] .data             PROGBITS        080497c0 0007c0 000008 00  WA  0   0  4
  [26] .bss              NOBITS          080497c8 0007c8 000004 00  WA  0   0  4
  [27] .comment          PROGBITS        00000000 0007c8 000038 01  MS  0   0  1
  [28] .debug_aranges    PROGBITS        00000000 000800 000038 00      0   0  1
  [29] .debug_info       PROGBITS        00000000 000838 00018f 00      0   0  1
  [30] .debug_abbrev     PROGBITS        00000000 0009c7 00012b 00      0   0  1
  [31] .debug_line       PROGBITS        00000000 000af2 00007e 00      0   0  1
  [32] .debug_str        PROGBITS        00000000 000b70 0000d8 01  MS  0   0  1
  [33] .debug_loc        PROGBITS        00000000 000c48 0000e0 00      0   0  1
  [34] .debug_ranges     PROGBITS        00000000 000d28 000028 00      0   0  1
```

[9]Die Beispielprogramm wurde mit folgendem Befehl übersetzt: „g++ -g -O0 beispiel.cc -o beispiel “

```
[35] .shstrtab         STRTAB          00000000 000d50 00015f 00      0   0  1
[36] .symtab           SYMTAB          00000000 0014a0 0004c0 10     37  52  4
[37] .strtab           STRTAB          00000000 001960 000281 00      0   0  1
```

In diesem Beispiel befinden sich die kompletten Debug-Informationen in den Sektionen 28–34. Bei diesen Sektionen beginnt der Name immer mit .debug_ [22]. Insgesamt gibt es zwölf unterschiedliche Debug Sektionen, die nach Bedarf in die ELF-Binärdatei einfließen [23]. Für die Untersuchungen in dieser Arbeit sind die in diesem Beispiel vorhandene Debug-Sektionen ausreichend. Entsprechend wird auf eine ausführliche Erläuterung der hier nicht vorkommenden Debug-Sektionen verzichtet. Die Sektionen 28–34 haben folgenden Inhalt:

- **.debug_aranges**

  Enthält eine Abbildung von Speicheradressen auf Übersetzungseinheiten. Für diese Beispiel:

```
$ objdump --dwarf=aranges beispiel

beispiel:     file format elf32-i386

Contents of the .debug_aranges section:

  Length:                   52
  Version:                  2
  Offset into .debug_info:  0x0
  Pointer Size:             4
  Segment Size:             0

    Address    Length
    0804848c 0000002b
    080484b7 00000020
    080484d8 0000001f
    080484f8 0000000a
    00000000 00000000
```

  Zu sehen ist, dass z.B. die Instruktionen, die an Adresse 0x0804848c–0x080484b7 (Länge 0x2b) liegen, zu der Übersetzungseinheit gehören, die in der Sektion .debug_info unter dem Index 0x0 zu finden ist.

- **.debug_info**

  Die Sektion .debug_info ist die wichtigste aller Sektionen. Sie bündelt alle Debug-Informationen. Der Inhalt ist in einer Baumstruktur organisiert und orientiert sich an der Struktur des Quellcodes. Da die Debug-Informationen für dieses kleine Beispiel bereits 165 Zeilen umfasst, werden hier lediglich die Informationen betrachtet, die die main-Funktion betreffen[10]:

[10] In Anhnag A.1 ist der gesamte Inhalt dieser Sektion abgebildet.

```
...
<1><129>: Abbrev Number: 17 (DW_TAG_subprogram)
    <12a>   DW_AT_external    : 1
    <12b>   DW_AT_name        : (indirect string, offset: 0x8e): main
    <12f>   DW_AT_decl_file   : 1
    <130>   DW_AT_decl_line   : 19
    <131>   DW_AT_type        : <0x82>
    <135>   DW_AT_low_pc      : 0x804848c
    <139>   DW_AT_high_pc     : 0x80484b7
    <13d>   DW_AT_frame_base  : 0xa8 (location list)
    <141>   DW_AT_GNU_all_tail_call_sites: 1
    <142>   DW_AT_sibling     : <0x17b>
 <2><146>: Abbrev Number: 11 (DW_TAG_formal_parameter)
    <147>   DW_AT_name        : (indirect string, offset: 0x0): argc
    <14b>   DW_AT_decl_file   : 1
    <14c>   DW_AT_decl_line   : 19
    <14d>   DW_AT_type        : <0x82>
    <151>   DW_AT_location    : 2 byte block: 91 0  (DW_OP_fbreg: 0)
 <2><154>: Abbrev Number: 11 (DW_TAG_formal_parameter)
    <155>   DW_AT_name        : (indirect string, offset: 0xf): argv
    <159>   DW_AT_decl_file   : 1
    <15a>   DW_AT_decl_line   : 19
    <15b>   DW_AT_type        : <0x17b>
    <15f>   DW_AT_location    : 2 byte block: 91 4  (DW_OP_fbreg: 4)
 <2><162>: Abbrev Number: 18 (DW_TAG_lexical_block)
    <163>   DW_AT_low_pc      : 0x8048495
    <167>   DW_AT_high_pc     : 0x80484b5
 <3><16b>: Abbrev Number: 19 (DW_TAG_variable)
    <16c>   DW_AT_name        : (indirect string, offset: 0x54): myPerson
    <170>   DW_AT_decl_file   : 1
    <171>   DW_AT_decl_line   : 21
    <172>   DW_AT_type        : <0x29>
    <176>   DW_AT_location    : 2 byte block: 74 1c  (DW_OP_breg4 (esp): 28)
...
```

Jeder Knoten in der Baumstruktur der Sektion .debug_info ist ein sogenannter *Debug Information Entry*, im Nachfolgenden als *DIE* bezeichnet. Ein DIE hat eine Tiefe in der Baumstruktur, die durch die erste Zahl in spitze Klammern angedeutet wird. Des Weiteren haben DIEs einen Typen. Der Eintrag, der sich auf die main-Funktion bezieht hat den Typen DW_TAG_subprogram und deutet entsprechend an, dass es sich um eine Funktion handelt. Über diese Typen wird in DWARF jedes Element des Quellcodes abgebildet. Entsprechend ist die Anzahl der Möglichkeiten mit insgesamt 60 Typen recht umfangreich[24]. Jeder DIE enthält unterschiedliche Attribute, die mit DW_AT_ beginnen. Insgesamt gibt es in der aktuellen Spezifikation von DWARF 94 Attributtypen.

In diesem konkreten Beispiel hat die main-Funktion zwei Parameter (argc, argv) und eine lokale Variable mit dem Namen myPerson. Neben der Information, wo diese Variablen im Quellcode deklariert wurden, enthält das Attribut DW_AT_location eine Beschreibung, wo die Variable zur Laufzeit im Speicher liegt. Für den Parame-

ter argc und argv findet die Berechnung der Speicheradresse zur Laufzeit relativ mit einem Offset (0 bzw. 4 Bytes) zur Basis-Adresse der zugehörigen Funktion statt[25]. Die Basis-Adresse der main-Funktion steht im Parameter DW_AT_frame_base und verweist auf einen Index, der in der Sektion .debug_loc (location list) zu finden ist. Die lokale Variable myPerson wird dagegen über den Stackpointer (esp) mit dem Offset 28 berechnet. Darüber hinaus befindet sich in dem Attribut DW_AT_type ein Verweis auf einen anderen DIE innerhalb der Sektion .debug_info. Für argc steht in dem DIE, der die Typinformation enthält Folgendes:

```
...
<1><82>: Abbrev Number: 8 (DW_TAG_base_type)
    <83>   DW_AT_byte_size   : 4
    <84>   DW_AT_encoding    : 5 (signed)
    <85>   DW_AT_name        : int
...
```

Es handelt sich in diesem Fall also um einen 4 Byte großen vorzeichenbehafteten Integer. Neben diesen primitiven Typen werden in DWARF über den gleichen Weg Klassen-Typen beschrieben. Beispielweise ist Person ein solcher DW_TAG_class_type.

- **.debug_abbrev**

  Diese Sektion enthält die Struktur und den Typ jeder Information, die in der Sektion .debug_info zu finden ist.

- **.debug_line**

  Eine Abbildung von den Speicheradressen der Instruktionen auf die entsprechenden Zeilen und Dateien im Quellcode ist in der Sektion .debug_line zu finden. Für das Beispiel sieht die Tabelle wie folgt aus:

```
$ objdump --dwarf=decodedline beispiel
beispiel:     file format elf32-i386

Decoded dump of debug contents of section .debug_line:

CU: beispiel.cc:
File name                            Line number    Starting address
beispiel.cc                                    1           0x80484b7

beispiel.cc                                    2           0x80484ba
beispiel.cc                                    2           0x80484c0
beispiel.cc                                    3           0x80484c9
beispiel.cc                                    5           0x80484d0
beispiel.cc                                    7           0x80484d5
beispiel.cc                                   15           0x80484d8

beispiel.cc                                   15           0x80484de
beispiel.cc                                   15           0x80484ed
```

```
beispiel.cc                              15          0x80484f5
beispiel.cc                              16          0x80484f8

beispiel.cc                              16          0x80484fb
beispiel.cc                              20          0x804848c

beispiel.cc                              22          0x8048495
beispiel.cc                              24          0x80484a9
beispiel.cc                              25          0x80484b5
```

Zu beachten ist hierbei, dass zwei Zeilen immer einen Bereich von Instruktionen abdecken. Somit deckt die erste Zeile im Hochsprachen-Code die Instruktionen von der Adresse 0x80484b7 bis 0x80484ba (exklusive) ab.

- **.debug_str**

  Zeichenketten, auf die in .debug_info verwiesen wird, stehen in dieser Sektion.

- **.debug_loc**

  Unterschiedliche Attribute wie z.B. DW_AT_frame_base oder DW_AT_location verweisen auf die sogenannte *location list*. Damit ist der Inhalt der Sektion .debug_loc gemeint. Der Wert des Attributes stellt den Index der Einträge in der *location list* dar. Für die main-Funktion aus dem Beispiel sehen die entsprechenden Einträge folgendermaßen aus:

  ```
  $ objdump --dwarf=loc beispiel
  ...
  000000a8 0804848c 0804848d (DW_OP_breg4 (esp): 4)
  000000a8 0804848d 0804848f (DW_OP_breg4 (esp): 8)
  000000a8 0804848f 080484b6 (DW_OP_breg5 (ebp): 8)
  000000a8 080484b6 080484b7 (DW_OP_breg4 (esp): 4)
  000000a8 <End of list>
  ```

  Der erste Wert steht für den Index, auf den sich das Attribut bezieht. Die erste Zeile zeigt an, dass die Basis-Adresse der main-Funktion für die Instruktionsadressen 0x0804848c bis 0804848d (exklusive) berechnet wird, indem die Adresse auf die der Stackpointer zeigt, mit vier addiert wird. Da sich der Inhalt der Register während der Abarbeitung einer Funktion ändern kann, gibt es mehrere Bereiche mit unterschiedlichen Berechnungen.

- **.debug_ranges**

  Wie in .debug_loc enthält die Sektion .debug_ranges ebenfalls Informationen um Basis-Adressen oder Adressen von Variablen zu bestimmen. Der Unterschied besteht darin, dass die Informationen in .debug_loc sich immer auf einen kontinuierlichen Bereich von Instruktionsadressen bezieht. Im Gegensatz dazu müssen die Informationen in .debug_ranges nicht kontinuierlich sein. Auf Einträge in dieser Sektion wird in DW_AT_ranges verwiesen.

2.5.2. Komplexität der Ortsangaben

Die Ortsangaben für lokale Variablen durch das Attribut DW_AT_location sowie den Informationen in der Sektion .debug_loc können sehr komplexe Formen annehmen.

Handelt es sich um ein Objekt, das über die gesamte Laufzeit des Programms immer an ein und derselben Stelle im Speicher steht, wird dieser Ort direkt über das Attribut DW_AT_location ausgedrückt. Handelt es sich z.B. um eine lokale Variable, die durch den Auf- und Abbau des Stacks an unterschiedlichen Stellen im Speicher stehen kann, steht die Ortsangabe in der *location list* in .debug_loc.

Um den Ort, an dem beispielsweise eine lokale Variable gespeichert ist anzugeben, gibt es unterschiedliche Möglichkeiten[24]:

- **DW_OP_reg3 / DW_OP_regx 54**

  Die Variable steht im Register 3 bzw. 54.

- **DW_OP_addr 0x80d0045c**

  Die Variable steht an Adresse 0x80d0045c. Diese Angabe ist häufig für statische Variablen zu finden.

- **DW_OP_breg11 44**

  Die Variable steht an der Adresse, die in Register 11 steht, mit einem Offset von 44 Bytes.

- **DW_OP_fbreg -50**

  Die Variable steht an der Basis-Adresse der Funktion (Anfang des Stackframe) mit einem Offset von -50 Bytes.

- **DW_OP_bregx 54 32 DW_OP_deref**

  Es handelt sich um einen call-by-reference Parameter, der sich an der Adresse befindet, auf die Register 54 zeigt, mit einem Offset von 32 Bytes.

- **DW_OP_plus_uconst 4**

  Bei dieser Angabe handelt es sich um eine Member-Variable einer Klasse bzw. eines Structs. Die Variable befindet sich an der Adresse des Klassen-Objektes mit einem Offset von 4 Bytes.

Neben diesen trivialen Ortsangaben gibt es die Möglichkeit, dass Ortsangaben partiell definiert werden. Dazu beziehen sich triviale Ortsangaben auf einen bestimmten Teil einer Variable. So kann beispielsweise mit DW_OP_piece4 angegeben werden, dass sich die Ortsangabe, die direkt vor diesem Ausdruck steht auf 4 Bytes einer Variable beziehen. Einige Beispiele dazu sind nachfolgend aufgeführt[24]:

- **DW_OP_reg3 DW_OP_piece4 DW_OP_reg10 DW_OP_piece2**

  Die ersten vier Bytes des Objektes stehen in Register 3. Weitere zwei Bytes stehen in Register 10.

- **DW_OP_reg0 DW_OP_piece4 DW_OP_piece4 DW_OP_fbreg -12 DW_OP_piece4**

  Die ersten vier Bytes stehen in Register 0. Zu den mittleren vier Bytes kann keine Ortsangabe gemacht werden. Möglicherweise sind sie durch Optimierungen weggefallen. Die letzten vier Bytes stehen an der Adresse der zugehörigen Funktion mit einem Offset von -12 Bytes.

Die zusammengesetzten Ortsangaben können noch weitere komplexere Formen annehmen. Für weitere Informationen wird auf die DWARF Spezifikation in der Version 4 verwiesen[24]. Entsprechend ist die Bestimmung der Speicheradresse, an der beispielsweise eine lokale Variable liegt, nicht immer trivial.

2.5.3. Mehrdeutigkeit des DWARF-Formates

Die Spezifikation von DWARF ist sehr umfangreich und bietet viele Möglichkeiten die Debug-Informationen zu speichern. Neben dem großen Umfang gibt es zum Zeitpunkt der Erstellung dieser Arbeit insgesamt 4 Versionen der Spezifikation von DWARF. Genau durch diese Vielfalt entsteht das Problem, dass Compiler bzgl. der Art und Weise, wie sie Debug-Informationen ablegen, einen Spielraum haben.

Dies wird bereits deutlich, wenn man die Debug-Informationen aus dem Beispiel mit der Spezifikation von DWARF in der Version 4 vergleicht. Der Typ des DIE, der sich auf die main-Funktion bezieht lautet DW_TAG_subprogram. In der Spezifikation steht jedoch, dass es für die main-Funktion auch einen eigenen Typen DW_AT_main_subprogram gibt. Welche Variante letztendlich gewählt wird bleibt dem Compiler überlassen.

Ein weiteres Beispiel für die Mehrdeutigkeit der Spezifikation wird deutlich, wenn man sich die Erläuterungen zu den Attributen DW_AT_low_pc und DW_AT_entry_pc anschaut. Beide Attribute enthalten als Wert die erste Adresse der Instruktion, mit der eine Funktion anfängt. In der DWARF-Spezifikation steht dazu:

„While the DW_AT_entry_pc attribute might also seem appropriate for this purpose, historically the DW_AT_low_pc attribute was used before the DW_AT_entry_pc was introduced (in DWARF Version 3). There is insufficient reason to change this.“
Zitat aus [24], Seite 38.

2.6. Zusammenfassung

In diesem Grundlagen-Kapitel wurde zunächst betrachtet, wie transiente Hardware-Fehler entstehen und wie sie kategorisiert werden können. Anschließend wurde beschrieben, wie Fehlerinjektion im allgemeinen stattfinden kann, und wie diese konkret durch das Fehlerinjektions-Framwork *FAIL** umgesetzt wird. Die Erläuterung bzgl. des ELF-Binärformates zeigt auf, wie das Format aufgebaut ist, und bildete die Basis, um zu erläutern, wie Debug-Informationen innerhalb einer Binärdatei untergebracht werden. Abschließend wurde mit einem kleinen Beispiel erklärt, wie das Debug-Format DWARF

aufgebaut ist, und wie Informationen über das Programm gespeichert werden. Neben der allgemeinen Betrachtung wurde detailliert erläutert, wie sich die Ortsangaben z.B. bzgl. einer Variable zusammensetzen kann. Dabei wurde ersichtlich, dass diese Ortsangaben äußerst komplexe Formen annehmen können, was eine Analyse auf Basis der Lage von lokalen Variablen im Speicher erschwert. Neben den teils komplexen Angaben wurde deutlich, dass das DWARF-Debug-Format durch den enormen Umfang Ungenauigkeiten und somit Spielraum für den Compiler lässt. Diese Ungenauigkeiten erschweren eine Analyse auf Basis der Daten, die aus den Debug-Informationen stammen, zusätzlich.

3. Related Work

In diesem Kapitel wird Literatur vorgestellt, die mit dem Thema dieser Arbeit in Verbindung steht. Dabei wird jeweils die Kernaussage vorgestellt und die Relevanz für diese Arbeit erläutert.

Literatur, die sich in gleicher Art und Weise mit der Analyse von Fehlerinjektionsexperimenten, wie sie in dieser Arbeit untersucht wird, beschäftigen, konnte nicht gefunden werden. Ebenso konnte keine Literatur gefunden werden, die sich mit der Automatisierung auf Basis der hier vorgestellten oder ähnlichen Analysen befasst.

3.1. Debug-Informationen optimierter Programme

Die Debug-Informationen, die vom Compiler generiert werden, stellen, wie in Kapitel 2 bereits angedeutet, die Basis für aussagekräftige Analysen der Ergebnisdaten dar. Daher ist es naheliegend allgemein bekannte Probleme bzgl. der Debug-Informationen mit entsprechender Literatur zu erläutern.

Wenn in einem Programm zur Laufzeit nach einem Fehler gesucht wird, wird dazu ein Debugger verwendet. Dieser Debugger greift auf die Debug-Informationen zu, die vom Compiler in der Binärdatei hinterlegt wurden. Um die Suche nach dem Fehler zu vereinfachen, können über den Debugger Breakpoints auf bestimmte Zeilen des Hochsprachencodes gesetzt werden. Wird dieser Breakpoint erreicht, hält der Debugger das Programm an und ermöglicht dem Entwickler weitere Analysen. Damit durch den Breakpoint das Programm angehalten werden kann, muss die Zeile des Hochsprachencodes auf eine konkrete Instruktion des Assembler-Codes abgebildet werden. Diese Abbildungs-Informationen stehen, wie in den Grundlagen erläutert, beim DWARF-Format in der .debug_line Sektion. Durch vom Compiler durchgeführte Optimierungen, werden die Abbildungs-Informationen zwischen Hochsprachencode und Assembler ungenau. Dieses Problem bezeichnet man als *Code-Location-Problem* [26].

Geht es darum den Wert einer Variable zu einem bestimmten Ausführungszeitpunkt herauszufinden, beschreibt dies das *Data-Value-Problem* [26]. Auch bei diesem Problem werden die Debug-Informationen, in Abhängigkeit zu den Optimierungen, ungenauer. Da in dieser Arbeit Verfahren zur Analyse des Stacks aus Basis der Debug-Informationen untersucht werden, spielt dieses Problem eine wesentliche Rolle.

Debug-Informationen dienen primär dazu ein Programm zur Laufzeit zu untersuchen. Für diesen Zweck wird das Programm bevorzugt ohne Optimierungen übersetzt, da angenommen werden kann, dass die Semantik des Programms identisch ist [27].

Die Debug-Informationen werden, wie bereits angedeutet, im Rahmen dieser Arbeit u.a. dazu verwendet einzelne Ergebnisse der Fehlerinjektionen den entsprechenden lo-

kalen Variablen zuzuordnen. Optimierungen beeinflussen die Art und Weise wie die Variablen im Programmfluss verwendet werden. Beispielsweise werden Variablen durch Optimierungen komplett vom Speicher in Register verschoben oder so optimiert, dass einzelne Variablen komplett wegfallen. Daher ist es für diese Arbeit von größter Bedeutung, dass die Debug-Informationen auch bei einer Übersetzung mit Optimierungen korrekt bleiben.

Es gibt mehrere Publikationen, die den Ansatz verfolgen, die Debug-Informationen auch während der Durchführung von Optimierungen in einem korrekten Zustand zu halten [28, 29, 27, 26, 30, 31]. Um dies zu erreichen wird entweder der Compiler oder der Optimierer entsprechend angepasst. Aufgrund der Tatsache, dass es eine Vielzahl an Optimierungen gibt, wird in dieser Literatur lediglich eine kleine Teilmenge dieser möglichen Optimierungen untersucht.

Neben dem Instandhalten der Debug-Informationen gibt es weitere Ansätze. Beispielweise wird durch Hölzle et al. [32] ein Ansatz verfolgt, durch den die Optimierungen dynamisch rückgängig gemacht werden. Dazu wird allerdings eine spezielle Sprache *(SELF)* verwendet. Somit ist dieser Ansatz nicht einfach auf aktuelle Compiler und Programmiersprachen übertragbar. Dennoch zeigen die Ergebnisse, dass es möglich ist, korrekte Debug-Informationen für Programme zu erzeugen, die mit Optimierungen übersetzt wurden. Ein weiterer Ansatz versucht die fehlerhaften Debug-Information durch Interpretation des Hochsprachencodes zur Laufzeit zu kompensieren [33].

3.2. Analyse von Ergebnissen aus Fehlerinjektionsexperimenten

Das Fehlerinjektionsframework FAIL* wurde bereits in mehreren Publikationen verwendet (u.a. in [5, 34, 35]). Für diese Arbeit ist vor allem interessant, wie die Ergebnisse der Fehlerinjektionsexperimente analysiert wurden.

Das Fehlermodell dieser Veröffentlichungen betrachtet, genau wie in dieser Arbeit, kippende Bits im Speicher. Entsprechend haben die Publikationen gemeinsam, dass sie die Auswirkungen der Fehlerinjektionen auf Variablen im Speicher analysieren. Die Zuordnung zwischen Speicheradresse und Variable erfolgt mit manuellem Aufwand.

Eine der Publikationen untersuchte Speicherfehler im Allgemeinen [35]. Für die Analyse wurden die Fehler den globalen Variablen zugeordnet. Für den Stack erfolgte keine detaillierte Aufschlüsselung auf der Granularität von Variablen. Entsprechend können lediglich Aussagen darüber getroffen werden, wie viele Fehler dem Speicherbereich des Stacks zuzuordnen sind. Für zwei Benchmarks, die in dieser Publikation verwendet wurden, werden 22.4 % bzw. 19.1 % der Fehler dem Stack zugeordnet. Der Stack stellt somit einer der anfälligsten Speicherbereiche dar [35]. Dennoch wurde in dieser Publikation für genauere Analysen des Stacks auf zukünftige Arbeiten verwiesen.

Eine weitere Publikation beschreibt, wie der vptr-Pointer geschützt werden kann, den der Compiler für virtuelle Funktionen erzeugt [34]. Dazu wurde ein Benchmark verwendet, der eine Wetterstation implementiert bzw. simuliert. Die Objekte, die über virtu-

elle Funktionen verfügen wurden global angelegt. Entsprechend ist die Betrachtung des Stacks für diese Arbeit uninteressant. Um Aussagen über die Anfälligkeit dieser Datenstrukturen treffen zu können, wurden diese manuell den Speicheradressen zugeordnet. In der Evaluierung in Kapitel 3 wird auf diese Publikation detaillierter eingegangen.

Eine weitere Publikation von Borchert et al. untersucht die Anfälligkeit von Rücksprungadressen, die für jeden Funktionsaufruf auf dem Stack angelegt werden [5]. Im dazu verwendeten Benchmark betreffen 42.6 % aller Fehler den Stack. Abbildung 1.2 aus der Einleitung zeigt eine grafische Analyse der Ergebnisse aus den Fehlerinjektionsexperimenten. In dieser Abbildung wurden die Rücksprungadressen *(Return Addresses)* und die Basisadressen *(Frame Pointer)* manuell gekennzeichnet. Die Färbung deutet an, dass die Injektion eines Speicherfehlers an dieser Adresse zu diesem Zeitpunkt einen ungewolltes Ende des Programms – also eine Abweichung vom *Golden Run* – herbeigeführt hat. Darüber hinaus wurde analysiert, wie oft und wie lange ein Stackframe einer Funktion auf dem Stack lag. Trotzdem es in dieser Publikation ausschließlich um Variablen bzw. vom Compiler generierte Datenstrukturen auf dem Stack geht, gibt es keine Aussagen darüber wie viele Fehler konkret diesen Datenstrukturen zuzurechnen sind. Eine Aggregation dieser Daten auf der Granularitäts-Ebene von Variablen bzw. Datenstrukturen war ohne hohen manuellen Aufwand nicht möglich. Auch diese Publikation wird im Evaluations-Kapitel 6 näher untersucht.

3.3. Platzierung von Fehlertoleranzmechanismen

Wie in der Einleitung beschrieben, ist es aufgrund von Ressourcenbeschränkungen notwendig, Fehlertoleranzmaßnahmen lediglich für Komponenten anzuwenden, die besonders anfällig für Fehler sind. Im Gegensatz zu der in dieser Arbeit verwendeten Methode, stützen sich die nachfolgenden Verfahren ausschließlich auf Informationen, die bereits durch das System gegeben sind [36, 12, 37, 38, 39]. Es werden also keine Fehlerinjektionsexperimente durchgeführt.

Der Ansatz von Benso et al. [12] errechnet auf Basis der Lebenszeiten von Variablen, wie fehleranfällig diese ist. Dabei geht die Lebenszeit der Variable, je nachdem welche Ereignisse eintreten, unterschiedlich stark in die Berechnung der Fehleranfälligkeit ein. So geht beispielsweise die Lebenszeit zwischen der Deklaration und dem ersten Schreiben der Variable nicht mit in die Berechnung ein. Die Zeit zwischen einem Schreiben mit anschließenden Lesen einer Variable geht dagegen mit einer zuvor festgelegten Konstante in die Berechnung ein. Mit diesen Berechnungsregeln für alle möglichen Ereigniskombinationen ergibt sich die Fehleranfälligkeit für eine bestimmte Variable.

Eine weitere Möglichkeit zur Berechnung einer Fehleranfälligkeit wird durch Sridharan et al. aufgezeigt [37]. Der Berechnung liegt eine bestimmte Bitbreite einer bestimmten Hardwarekomponente zugrunde. Dabei wird für jede Instruktion die Anzahl der Bits errechnet, die für eine korrekte Ausführung notwendig sind. Über diese Informationen lassen sich über den Programmverlauf für jede Instruktion ein Wert errechnen, der ausdrückt, wie Fehleranfällig das Programm in einer bestimmten Programmphase ist. Der errechnete Wert bezieht sich dabei lediglich auf die zugrunde gelegte Hardwarekompo-

nente. Ein ähnlicher Ansatz, auf Basis von Prozessor-Zyklen wurde durch Mukherjee et al. beschrieben [39].

In [38] wird die Software als eine Vielzahl von Modulen betrachtet, die miteinander über Variablen kommunizieren. Für diese Kommunikation lassen sich unterschiedliche Graphen erstellen, für die wiederum Metriken berechnet werden. Diese Metriken geben Aufschluss darüber, welche Kommunikation bzw. Variable als besonders fehleranfällig einzustufen ist. Die Wahl der Granularität der Module hat einen signifikanten Einfluss auf die Berechnung.

Auf Basis von Instruktionen wird ein dynamischer Abhängigkeitsgraph im Ansatz von Pattabiraman et al. erstellt [36]. Darauf werden eine Vielzahl von Metriken berechnet. Die Metriken geben Hinweise darauf, welche Variablen-Arten (z.B. lokale Variablen) besonders fehleranfällig sein könnten. Auskunft über die Fehleranfälligkeit bzgl. konkreter Variablen, geben diese Metriken nicht. Ebenso ist eine Ordnung von Variablen bzgl. der Fehleranfälligkeit nicht möglich.

3.4. Zusammenfassung

Zunächst wurde festgestellt, dass keine Literatur gefunden werden konnte, die Analysen, in der Form wie sie in dieser Arbeit untersucht werden, für Fehlerinjektionsexperimente vorstellt. Insbesondere konnte keine Literatur gefunden werden, die Analysen auf Basis der Debug-Informationen ermöglicht. Aufgrund der Verwendung von Debug-Informationen in dieser Arbeit stellen diese ein verwandtes Themengebiet dar. Die Literatur hat gezeigt, dass Debug-Informationen insbesondere in Kombination mit Optimierungen ungenau werden. Es existieren zwar Ansätze zur Lösung dieses Problems für Teilmengen der Optimierungen, allerdings können diese aufgrund des zu hohen Aufwandes nicht im Rahmen dieser Arbeit betrachtet werden. Die Betrachtung bisheriger Analysen von Fehlerinjektionsexperimenten hat deutlich gezeigt, dass eine detaillierte Analyse des Stacks notwendig ist. Abschließend wurden Verfahren aus der Literatur vorgestellt, die es ermöglichen Aussagen über die Fehleranfälligkeit einzelner Softwarekomponenten zu treffen.

4. Analyse und Entwurf

Dieses Kapitel beschreibt sowohl die Analyse der Anforderungen an neue Analyseverfahren, als auch deren Entwurf. In Kapitel 4.1 werden zunächst die derzeitigen Probleme untersucht, die bei der Analyse von Ergebnisdaten existieren. Darauf aufbauend werden Anwendungsfälle betrachtet, die sich aus den Problemen ableiten. Abschließend wird ein entsprechender Lösungsansatz vorgestellt. In Kapitel 2, 4.4, 4.5, 4.6 werden die einzelnen Lösungen im Detail vorgestellt. Kapitel 4.7 enthält den Softwarearchitektur-Entwurf, in dem erläutert wird, wie die neuen Analysen in die vorhandene Struktur des FAIL*-Frameworks eingebracht werden können.

4.1. Anforderungsanalyse

Wie in den Grundlagen beschrieben, ist es mit dem FAIL*-Framework möglich, zu einem bestimmten Zeitpunkt in einem bestimmten Ort im Fehlerraum einen Fehler zu injizieren. Für das hier betrachtete Fehlermodell, das Fehlerinjektion in Speicheradressen vorsieht, erhält man so für eine Vielzahl von Speicheradressen Ergebnisdaten. Werden pro Speicheradresse alle acht Bits in einem Experiment gekippt, existiert für jede Speicheradresse lediglich ein Ergebnis. Wird jedes Bit einzeln gekippt, beträgt die Anzahl der Ergebnisdaten das Achtfache. So müssen pro Benchmark trotz der in den Grundlagen beschriebenen Pruning-Technik meist mehrere Millionen Experimente durchgeführt werden.

Für eine derart große Menge von Ergebnisdaten ist es nicht mehr möglich durch manuelle Durchsicht der Daten, Aussagen über die Fehleranfälligkeit einzelner Software-Komponenten zu tätigen.

Sofern man die Adresse einer globalen Variable kennt, können dieser entsprechend Fehler zugeordnet werden. Aber selbst dieser Schritt erfordert zusätzlichen Aufwand, indem die Adresse dieser Variable nachgeschaut werden muss.

Die Fehlerinjektionen erfolgen zu einem Zeitpunkt, der durch eine dynamische Instruktion beschrieben wird. Wie in den Grundlagen erläutert, kann jede dynamische Instruktion auf exakt eine statische Instruktion abgebildet werden. Somit lassen sich Aussagen bzgl. der Fehleranfälligkeit lediglich auf Ebene des Assembler-Codes treffen.

4.1.1. Anforderungen

Aufgrund der Komplexität und der geringen Aussagekraft der Ergebnisdaten ist es ohne Weiterverarbeitung nicht möglich herauszufinden, welche Software-Komponenten gegen

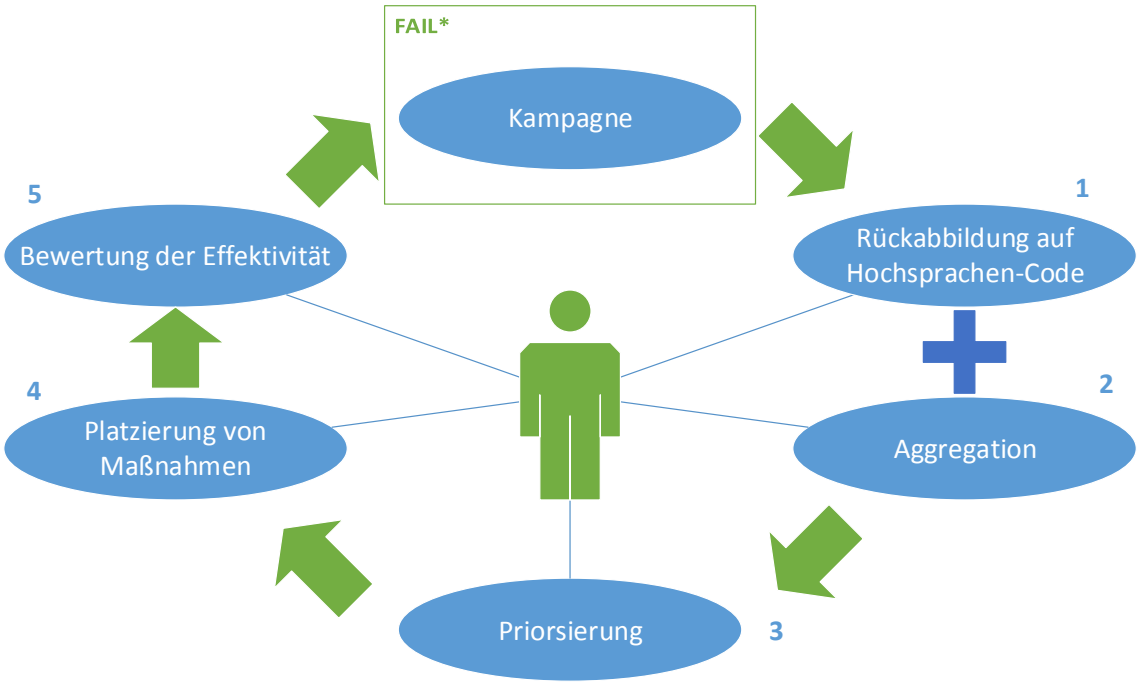

Abbildung 4.1.: Darstellung des Kreislaufs zur Härtung von Software mit den sich daraus ergebenden Anwendungsfällen

Einflüsse von Fehlern gehärtet werden müssen. Die Weiterverarbeitung kann auf unterschiedliche Art und Weise geschehen und hängt vom konkreten Analyse-Verfahren ab. Neben der konkreten Analyse ist es notwendig die Software-Komponenten nach ihrer Fehleranfälligkeit zu priorisieren. Mit Hilfe der Priorisierung können Fehlertoleranzmechanismen auf eine Komponente angepasst und angewandt werden. Nach der Härtung der fehleranfälligen Komponenten ist es notwendig zu bewerten, in wie weit die ergriffene Maßnahme zu einer verbesserten Fehlertoleranz beigetragen hat. Aus diesem Ablauf ergeben sich folgende Anforderungen:

1. **Abbildung der Fehler auf Hochsprachencode**

   Eine Analyse durch Abbildung der Fehler auf den Assembler-Code ist trivial, da die injizierten Fehler sich durch die zeitliche Beschreibung mit einer statischen Instruktion assoziieren lassen. Für die Härtung mittels Fehlertoleranzmaßnahmen werden jedoch Software-Komponenten auf der Ebene von Klassen oder Funktionen benötigt. Daher ist es naheliegend eine Verbindung zwischen Maschinencode und Hochsprache herzustellen. Durch die Rückabbildung wird eine Instruktion – und somit auch die dazugehörigen Fehler – auf eine Zeile des Hochsprachen-Codes abgebildet. Auf diese Anforderung wird in Kapitel 4.3 näher eingegangen.

2. **Aggregation der Ergebnisdaten auf Software-Komponenten**

Wie bereits aus der Betrachtung der Literatur in Kapitel 3.2 ersichtlich wurde, ist es notwendig die Ergebnisdaten mit Hilfe von Zusatzinformationen zu aggregieren. Insbesondere die Aggregation über lokale Variablen, die auf dem Stack im Speicher liegen, war bislang ohne manuelle Suche nicht möglich. Neben den lokalen Variablen kann eine Aggregation über weitere Granularitäts-Ebenen sinnvoll sein. Auf diese Anforderung wird in Kapitel 4.4 näher eingegangen.

3. **Priorisierung der (aggregierten) Ergebnisdaten**

   Durch die Aggregation der Ergebnisdaten können Aussagen zur Fehleranfälligkeit im Bezug auf eine Vielzahl von Software-Komponenten getroffen werden. Um zu entscheiden, welche dieser Komponenten gehärtet werden sollen, muss eine Ordnung über die Fehleranfälligkeit dieser Komponenten erfolgen. Diesbezüglich sind unterschiedliche Metriken denkbar. Auf diese Anforderung wird in Kapitel 4.5 näher eingegangen.

4. **Platzierung von Fehlertoleranzmechanismen**

   Nachdem die Priorisierung der fehleranfälligsten Komponenten erfolgt ist, müssen die Fehlertoleranzmechanismen auf diese Komponente spezialisiert werden. Dabei entsteht die Möglichkeit abzuschätzen, wie viele Fehler durch Anwendung der spezialisierten Fehlertoleranzmaßnahme zukünftig erkannt bzw. behoben werden können. Auf diese Anforderung wird in Kapitel 4.6 näher eingegangen.

5. **Bewertung der Effektivität der Maßnahmen**

   Die Anwendung der Fehlertoleranzmaßnahmen führt zu einer Änderung der Fehleranfälligkeit der Software. Um zu beurteilen, in wie weit die Fehleranfälligkeit einzelner Komponenten gestiegen oder gefallen ist, ist ein Vergleich notwendig.

4.1.2. Qualität der Ergebnisdaten

Die Ergebnisse aus der Kampagne sind nur dann wirklich aussagekräftig, wenn ein Großteil der Pfade innerhalb des Quellcodes des Benchmarks betreten wurde. Aus Basis der Daten aus dem *Trace* (nicht zu verwechseln mit dem *Stack-Trace*) lässt sich errechnen, wie viele der Instruktionen aus dem Benchmark ausgeführt wurden. Je größer der Anteil der ausgeführten Instruktionen desto aussagekräftiger sind die Ergebnisdaten aus der Kampagne. Diese Verfahren sind ähnlich zu denen, die im Bereich von Software-Tests angewandt werden (z.B. *statement-coverage*). Entsprechend existieren mehrere Möglichkeiten die Qualität der Ergebnisdaten bzw. deren Aussagekraft zu beurteilen.

4.1.3. Lösungsansatz

Da die Ergebnisdaten allein lediglich Informationen auf der Ebene von Instruktionenadressen enthalten, müssen zusätzliche Informationen hinzugezogen werden, um die Aussagekraft der Daten zu erhöhen.

Sowohl für die Rückabbildung auf Hochsprachen-Code, als auch für die Aggregation der Daten, können die Debug-Informationen, die vom Compiler erzeugt werden, hinzugezogen werden. Die dadurch verfügbaren Informationen zur Lage von Software-Komponenten im Speicher ermöglicht es nachzuvollziehen wie sich der Stack durch die Funktionsaufrufe auf- bzw. abbaut. Mit dieser Information kann einmalig ein *Stacktrace* aufgezeichnet werden, der die Lage der lokalen Variablen im Speicher über den zeitlichen Ablauf des Programms aufzeichnet. Mit dem Wissen, wann eine lokale Variable wo im Speicher gelegen hat, ist es nach Durchführung der Kampagne möglich die Ergebnisdaten über die lokalen Variablen zu aggregieren. Durch zusätzliche Informationen, wie z.B. den Typ einer Variable kann über weitere Software-Komponenten aggregiert werden.

Für die Priorisierung der (aggregierten) Ergebnisdaten können Metriken eingesetzt werden. Beispielsweise wäre eine einfache Metrik die Kategorisierung der Software-Komponenten sowie die Sortierung in den einzelnen Kategorien nach der Fehleranfälligkeit. Dabei hängt es von den existierenden Fehlertoleranzmechanismen ab, welche Kategorisierung sinnvoll ist.

Die Platzierung von Fehlertoleranzmechanismen erfolgt auf der Grundlage der Priorisierung. In Verbindung mit dem Wissen über die Wirkung eines Fehlertoleranzmechanismus lässt sich abschätzen, in wie weit dieser die Fehleranfälligkeit der Software beeinflusst. Durch eine erneute Durchführung der Kampagne für die gehärtete Software wird die Effektivität der Maßnahme zur Steigerung der Fehlertoleranz ersichtlich. Dies ist jedoch unter Umständen mit einem erheblichen Zeitaufwand verbunden. Sofern die Abschätzung der Wirkung eines Fehlertoleranzmechanismus genau genug erfolgt, kann unter Umständen auf eine erneute Durchführung einer Kampagne verzichtet werden.

Sofern die Härtung der Software über mehrere Iterationen mit mehreren Kampagnen stattfindet, ist es erforderlich zu messen, in wie weit die Fehlertoleranz gestiegen oder gefallen ist. Dazu können einerseits die Metriken aus der Priorisierung verwendet werden, andererseits können spezielle Vergleichsmetriken eingesetzt werden, die Differenzen in der Fehlertoleranz sichtbar machen. In eine solche Vergleichsmetrik könnte beispielsweise eingehen wie groß der Speicher- bzw. Laufzeitunterschied zwischen der gehärteten und der nicht gehärteten Software ist.

4.2. Klassifikation und Terminologie

Innerhalb eines Fehlerinjektions-Framework wie FAIL* gibt es eine Vielzahl von Rohdaten, die zur Definition der Experimente bzw. der Ergebnisdaten entstehen. Um die weiteren Untersuchungen zu vereinfachen werden diese Rohdaten in diesem Kapitel klassifiziert. Gleichzeitig wird eine Terminologie eingeführt.

Alle Rohdaten lassen sich in drei Kategorien unterteilen. Kategorien werden nachfolgend als *Dimensionen* bezeichnet. Dabei wird zunächst die räumliche von der zeitlichen Dimension unterschieden. Zu der zeitlichen Dimension zählen alle Rohdaten, die den zeitlichen Programmablauf abbilden. Zu der räumlichen Dimension zählen alle Rohdaten, die als Ort für eine Fehlerinjektion dienen können. Über die zeitliche und räumliche Dimension wird ein Raum aufgespannt, der dem Fehlerraum entspricht. Innerhalb des

Fehlerraums werden Fehler injiziert. Das Resultat der Injektion wird nachfolgend als Ergebnis-Dimension bezeichnet.

Zur Verdeutlichung werden die Rohdaten aus FAIL* folgendermaßen klassifiziert:

- **Räumliche Dimension (Wo?)**
  - Register
  - Speicheradresse
  - Statische Instruktion
  - Hochsprachen-Code

  Alle Rohdaten aus der räumlichen Dimension lassen sich in unterschiedlicher Granularität betrachten. Die Register lassen sich z.B. in Allzweckregister und Spezialregister unterteilen. Ebenso kann eine Speicheradresse zu einem bestimmten Speichersegment (z.B. Stack-Segment) gehören. Eine statische Instruktion gehört zu einer Funktion und Übersetzungseinheit. Ebenso gehört eine Zeile aus dem Hochsprachencode beispielsweise zu einer Funktion, Klasse oder Namespace.

- **Zeitliche Dimension (Wann?)**
  - Dynamische Instruktionen
  - Zeit

  Dynamische Instruktionen sowie zeitliche Messungen erlauben eine exakte Beschreibung eines Zeitpunktes einer Fehlerinjektion.

- **Ergebnis-Dimension (Wie?)**
  - Ergebnis der Injektion
  - Laufzeit
  - Zusätzliche Ergebnisse

  Das Ergebnis aus der Nachuntersuchung nach einer Fehlerinjektion ist die wichtigste Information. Sie gibt Auskunft darüber wie sich die Fehlerinjektion auswirkt. Neben dieser Information kann die Laufzeit als zusätzlicher Indikator hinzugezogen werden. Beispielsweise könnte die Laufzeit eines Experimentes doppelt so lang im Vergleich zum *Golden-Run* sein, aber trotzdem mit einem korrekten Ergebnis enden. Dies könnte darauf hinweisen, dass sich durch die Fehlerinjektion der Programmpfad geändert hat. Je nach Experiment können weitere Informationen über den Ausgang des Experimentes relevant sein.

4.3. Rückabbildung auf Hochsprachencode

Die Rückabbildung von Assembler-Code auf Hochsprachen-Code dient eigentlich zum Debuggen von Software während der Laufzeit. Dazu können sogenannte *Breakpoints* gesetzt werden. Der Benutzer setzt diese häufig auf eine Zeile des Hochsprachen-Codes. Der Debugger braucht allerdings eine konkrete Instruktion, an der er das Programm anhalten soll. Daher ist es erforderlich, dass der Assembler-Code auf den Hochsprachen-Code in den Debug-Information, abgebildet wird (siehe Grundlagen 2.5.1).

Diese Abbildung kann für die Analyse der Ergebnisdaten zweckentfremdet werden. Dadurch, dass in den Ergebnisdaten der Ausgang des Experimentes mit einer konkreten Instruktion zusammenhängt, wird nicht nur der Assembler-Code auf die Zeilen des Hochsprachen-Codes abgebildet, sondern auch die zugehörigen Fehler.

Neben dem Ausgang eines konkreten Experimentes können weitere Informationen auf den Hochsprachen-Code abgebildet werden. Da nicht offensichtlich ist, welche dieser Varianten sinnvoll ist, werden diese nachfolgend näher untersucht:

- **Experiment-Ergebnis**

  Durch das zuvor vorgestellt *Pruning*-Verfahren werden über die Analyse der Speicherzugriffe sogenannte Äquivalenzklassen erstellt. Für jede dieser Äquivalenzklassen ist es unerheblich zu welchem Zeitpunkt innerhalb dieses Bereichs der Fehler injiziert wird. Im FAIL*-Framework erfolgt die Injektion immer am „rechten Rand“. Der „rechten Rand“ entspricht somit der letzten Instruktion der Äquivalenzklasse.

- **Letzte Instruktion**

  Statt den Fehler auf eine der Instruktionen aus der Äquivalenzklasse abzubilden, könnte alternativ eine Abbildung auf die letzte bzw. vorletzte Instruktion vor Erkennung des Fehlers erfolgen. Dadurch wird es möglich zu verstehen wo und wie sich der Fehler letztlich ausgewirkt hat. So könnte beispielsweise erkannt werden, dass durch ein gekipptes Bit eine Sprung-Instruktion in das Data-Segment im Speicher führt. Sofern sich herausstellt, dass Sprung-Instruktionen besonders anfällig sind, können diese mit entsprechenden Fehlertoleranzmechanismen geschützt werden. Die letzte Instruktion wäre in diesem Beispiel nicht hilfreich, da sie eine Adresse im Data-Segment darstellt. Hilfreich wäre also lediglich die vorletzte Instruktion, da diese die fehlerhafte Sprung-Instruktion darstellt. Ob die letzte Instruktion oder die vorletzte Instruktion in den Ergebnisdaten gespeichert wird, ist eine Entscheidung, die durch das Experiment individuell getroffen werden muss.

Da die Abbildung der Ergebnisdaten auf die kompletten Äquivalenzklassen zu einer *Pseudo-Fehleranfälligkeit* führen kann, ist diese Darstellungsvariante nicht sinnvoll. Somit bleiben lediglich die Möglichkeiten die Ergebnisdaten auf die Instruktion abzubilden, in die der Fehler tatsächlich injiziert wurde und die Abbildung auf die letzte bzw. vorletzte Instruktion.

In einer grafischen Analyse wäre daher eine hybride Darstellung am sinnvollsten. Durch die Auswahl einer Zeile aus dem Hochsprachen-Code, auf die Ergebnisdaten abgebildet wurden, werden die Auswirkungen in Form der letzten ausgeführten Instruktionen ersichtlich. So kann eine Injektion eines Fehlers direkt mit den Auswirkungen verbunden werden.

4.4. Aggregation über Software-Komponenten

Es gibt Software-Komponenten, wie z.B. lokale Variablen, die in unterschiedlichen Programmphasen an unterschiedlichen Stellen im Speicher liegen. An diesen Speicherstellen werden über den Verlauf der Kampagne Fehler injiziert. Möchte man also wissen, welche Fehler diese Software-Komponenten betreffen, ist es notwendig diese Fehlerdaten zusammenzuführen. Dieser Vorgang wird als *Aggregation* bezeichnet.

Die Aggregation der Ergebnisdaten für eine globale Variable ist trivial, da diese sich im Normalfall während des gesamten Programmablaufs an der selben Speicherstelle befindet. Entsprechend müssen lediglich die Fehler, die diese Speicherstelle betreffen, aufsummiert werden.

Das Aggregieren ist für alle Informationen aus der räumlichen Dimension sinnvoll, da sich so Aussagen über die Fehleranfälligkeit der Software-Komponenten tätigen lassen. Eine Aggregation der Ergebnisdaten über die Informationen der zeitlichen Dimension ist nur begrenzt sinnvoll, weil sich so lediglich Erkenntnisse über die Fehleranfälligkeit von Programmphasen ergeben. Die Aggregation der Informationen aus der Ergebnis-Dimension ist trivial und entspricht einer üblichen Statistik über die Vorkommen der Ergebnis-Typen.

4.4.1. Der Stack-Trace

Um die Software-Elemente, die an variierenden Orten im Speicher liegen, aggregieren zu können, müssen sowohl der Auf–, als auch Abbau des Stackframes im Stack-Segment aufgezeichnet werden. Dieser Vorgang wird nachfolgend *Stack-Trace* genannt. Abbildung 4.2 zeigt den typischen Aufbau von Daten im Stack-Segment.

Ein Stackframe wird mit jedem Aufruf einer Funktion auf dem Stack angelegt. Ein Stackframe kann folgende Daten enthalten:

- **Übergabeparameter**

  Die Parameter der aufgerufenen Funktion werden als erster Bestandteil des Stackframe auf den Stack gelegt.

- **Generierte Daten des Compilers**

  Nach den Übergabeparametern folgen architektur- und compilerspezifische Daten. Bei den meisten Architekturen/Compilern werden in jedem Fall der Instruction Pointer sowie der Base-Pointer gesichert. Beide Informationen sind notwendig, damit die CPU nach dem Verlassen der Funktion ermitteln kann, welche Instruktion

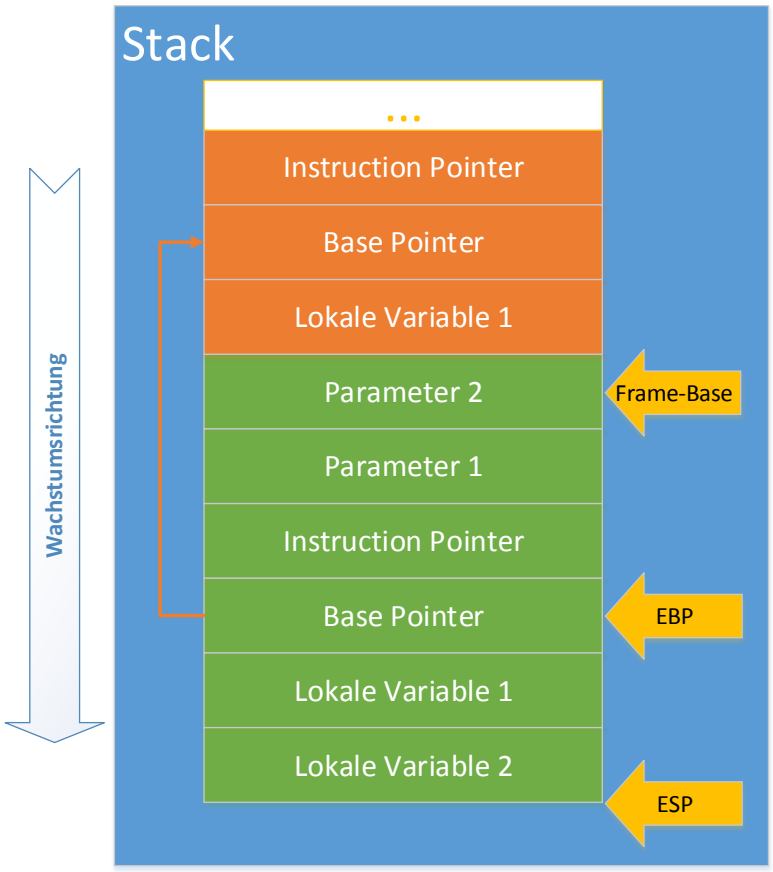

Abbildung 4.2.: Ein Stack mit zwei Stackframes (jeweils grün bzw. orange markiert). Zusätzlich sind die Register ESP und EBP (x86 spezifisch) eingezeichnet.

sie abarbeiten soll sowie welcher der entsprechende Stackframe ist.[1] Zusätzlich kann der Compiler weitere Daten ablegen. Sollte die Funktion zu einem Objekt einer Klasse gehören, wird zusätzlich ein Zeiger (*this* Pointer) auf das Klassenobjekt abgelegt. Dies ist notwendig, um auf entsprechende Member-Variablen dieser Klasse zugreifen zu können.

- **Lokale Variablen**

  Abschließend werden die lokalen Variablen in dem Stackframe abgelegt. Lokale Variablen mit undefinierter Größe sind davon ausgenommen, da diese im Heap-Segment verwaltet werden.

Um über jede die Datenstrukturen in einem Stackframe aggregieren zu können, muss deren Speicheradresse während der Laufzeit der Funktion bekannt sein. Dabei muss

[1] Eine Ausnahme hierfür stellt z.B. die IA-64 Architektur dar. Bei dieser Architektur können ganze Stackframes in Registerfenstern gehalten werden.

bspw. eine lokale Variable nicht über die gesamte Laufzeit der Funktion existieren. Aktuelle Compiler, wie z.B. der gcc ab Version 4.8.0, unterstützen standardmäßig eine Optimierung, die es erlaubt, mehrere Variablen auf die gleiche Speicheradresse zu legen. Dabei wird die Lebenszeit der Variablen untersucht. Sobald eine Variable nicht mehr benötigt wird, kann eine andere Variable, die danach verwendet wird, an der selben Speicheradresse liegen. Daher muss zusätzlich zu der Speicheradresse einer Variable auch deren Lebenszeit aufgezeichnet werden. Ruft die aktuelle Funktion eine andere Funktion auf, muss dies bei der Aufzeichnung der Lebenszeit beachtet werden. Denn auch während eine aufgerufene Funktion läuft, liegen die Datenstrukturen der aufrufenden Funktion weiterhin in ihrem Stackframe. Somit sind sie auch weiterhin anfällig für Fehlerinjektionen, da auf die entsprechenden Datenstrukturen nach Beendigung aurgerufenen Funktion verwendet werden könnten.

Die Lebenszeit eines Stackframe und somit auch die der Datenstrukturen endet mit dem Verlassen der Funktion, da dies einen Abbau des Stackframe bedeutet.

Der *Stack-Trace* muss lediglich ein einziges Mal mit einem korrektem Programmablauf, beispielsweise mit dem Golden-Run, aufgezeichnet werden, da für jedes Experiment der Programmablauf identisch ist.

Die notwendigen Informationen über die Lage des Stackframe und den entsprechenden Variablen kann aus den Debug-Informationen, wie im Grundlagen-Kapitel beschrieben, bezogen werden.

Aus den Debug-Informationen können Informationen über die aktuelle *Frame-Base* bezogen werden. Diese *Frame-Base* gibt für jeden Stackframe an, wo sie beginnt. In der Abbildung 4.2 ist diese für die grün markierte Funktion eingezeichnet.

Damit die Lebenszeiten des Stackframe und somit die der Variablen korrekt aufgezeichnet werden können, muss erkannt werden, wann ein Stackframe abgebaut wurde. Dies ist der Fall, sobald sich die *Frame-Base* auf eine Adresse bezieht, die höher ist als die des zuletzt betrachteten Stackframe.[2] Statt der *Frame-Base* aus den Debug-Informationen kann alternativ auch das Base-Pointer-Register (für x86: EBP) verwendet werden. Allerdings existieren Compiler-Optimierungen, die es ermöglichen ein Programm ohne Base-Pointer-Register auszuführen.[3]

4.4.2. Das Multi-Stack-Problem

In vielen Programmen werden mehrere Threads verwendet, um Nebenläufigkeit bzw. Parallelität zu implementieren. Selbst Anwendungen, die lediglich mit einem Thread ablaufen, können aufgrund des verwendeten Betriebssystems mehrere Threads verwenden. Beispielsweise ist es bei dem eingebetteten Betriebssystem *embedded Configurable operating system* (eCos) so, dass in der Standardkonfiguration zusätzlich ein sogenannter *Idle-Thread* existiert, der die Kontrolle übernimmt, wenn keine Anwendung mehr lauffähig ist.

[2] Diesem Verfahren liegt die Annahme zugrunde, dass der Stack von hohen Adressen in Richtung niedriger Adresse wächst.

[3] Für den GCC heißt die entsprechende Optimierung –fomit–frame–pointer.

Jeder Thread hat seinen eigenen Stack im Speicher, um mit Funktionsaufrufen umgehen zu können. Zusätzlich zu den Stacks der Threads kann es weitere Stacks geben. Beispielsweise ist in manchen Architekturen ein eigener Stack für Unterbrechungen vorgesehen.

Die Tatsache, dass es mehrere Stacks geben kann, muss beim Aufzeichnen des *Stack-Trace* beachtet werden. Durch den möglichen Wechsel des Stacks ergibt sich ein Problem bei der Bestimmung der Lebenszeit des Stackframe und den darin enthaltenen Datenstrukturen.

Das zuvor beschriebene Verfahren zur Erkennung, ob ein Stackframe abgebaut wurde, funktioniert nicht korrekt, wenn mehrere Stacks existieren, zwischen denen hin und her gesprungen wird. Ein Wechsel zu einem Stack, der an einem höheren Adressbereich liegt, würde dazu führen, dass die Lebenszeit des Stackframes und deren Datenstrukturen aus dem aktuellen Stack fälschlicherweise als abgebaut erkannt werden würden.

Somit ist es erforderlich zu erkennen, wann ein Wechsel des Stacks erfolgt. Dazu sind mehrere Verfahren denkbar, die nachfolgend diskutiert werden:

- **Rückverfolgung des Base-Pointer**

  Das Base-Pointer-Register zeigt auf eine Speicheradresse. Diese Speicheradresse enthält die Base-Pointer-Adresse der Funktion, welche die aktuelle Funktion aufgerufen hat. Dies ist in Abbildung 4.2 mit einem Pfeil angedeutet. Somit entsteht eine Kette von Zeigern, die auf den vorherigen Base-Pointer zeigen. Entsprechend kann diese Kette solange zurück verfolgt werden, bis der Anfang des aktuellen Stacks erreicht ist. Somit ließe sich prinzipiell feststellen, welcher Stack der aktuelle ist. Das Problem hierbei ist jedoch, dass dafür ein Kriterium gebraucht wird, anhand dessen erkannt werden kann, dass man den Anfang des Stacks erreicht hat. Welche Daten am Anfang des Stacks liegen ist jedoch Betriebssystem-abhängig. So wäre es denkbar, dass der Instruction-Pointer sowie der Base-Pointer, die am Anfang des ersten Stackframe liegen, mit NULL initialisiert wurden. Ebenso könnten sie Adressen enthalten, die sich auf den *Scheduler* des Betriebssystems beziehen. Es gibt vielfältige Möglichkeiten, wie die Daten am Anfang des Stacks aussehen. Somit wäre eine Zurückverfolgung des Base-Pointer aufgrund des notwendigen Abbruchkriteriums abhängig vom Betriebssystem.

- **Manuelle Angaben**

  Der Benutzer, der den *Stack-Trace* aufzeichnen möchte, könnte manuell durch Untersuchungen des Assembler-Code und der Konfiguration des Betriebssystems herausfinden, wie viele Stacks existieren und in welchem Speicherbereich diese liegen. Entsprechend könnte während der Aufzeichnung zu jeder dynamischen Instruktion (z.B. über den Base-Pointer oder die *Frame-Base*) geprüft werden, welcher Stack aktuell ist.

- **Heuristischer Ansatz**

  Ein trivialer heuristischer Ansatz zur Erkennung eines Wechsels des Stacks wäre die Angabe einer Mindestdifferenz zweier Adressen im Base-Pointer bzw. der

Frame-Base. Liegt die Differenz aus der alten Adresse und der neuen über dieser Mindestdifferenz handelt es sich wahrscheinlich um einen Wechsel des Stacks. Liegt sie darunter, wurde lediglich der Stackframe abgebaut. Dies erfordert jedoch, dass man diese Mindestdifferenz zuvor ermittelt hat. Dies könnte entweder manuell oder automatisiert passieren.

Zur Automatisierung könnten die Debug-Informationen herangezogen werden. Für jede Funktion gibt es Informationen darüber, welche Variablen innerhalb des Stackframe abgelegt werden. Ebenfalls ist die Größe der Variablen bekannt. Somit ließe sich errechnen, wie große der größte Stackframe sein kann. Dies kann entsprechend als Wert für die Mindestdifferenz verwendet werden. Liegt die Differenz der alten und der neuen Adresse im Base-Pointer bzw. der *Frame-Base* über dieser Mindestdifferenz muss es sich um einen Wechsel des Stacks handeln.

Das Verfahren funktioniert jedoch nur, solange es keine fast vollständig befüllten Stacks gibt, da dann die Differenz zwischen Base-Pointer bzw. der *Frame-Base* zum nächsten Stack unter Umständen zu klein ist.

- **Beobachtung des Base-Pointer und Stack-Pointer**

  Die Möglichkeiten, wie sich die Adresse des Base-Pointer ändern kann, sind beschränkt. Folgende Fälle können dabei eintreten:

  1. Stackframe wird abgebaut

     Sobald die aktuelle Funktion abgearbeitet ist, wird zum Aufrufer zurück gekehrt. Dabei wird der aktuelle Stack-Pointer auf die Adresse gesetzt, auf die er zeigt.

  2. Stackframe wird aufgebaut

     Eine Funktion wird aufgerufen und der entsprechende Stackframe auf dem Stack angelegt. Für jede Datenstruktur wird der Stack-Pointer weiter gesetzt. Er wird auf den Anfang der letzten beschriebenen Adresse gesetzt. Sobald die Übergabeparameter auf dem Stack angelegt sind und der Instruction-Pointer gesichert wurde, wird der Base-Pointer selbst gesichert und umgesetzt. Somit muss es eine Instruktion geben, bei der der Base-Pointer und der Stack-Pointer auf die gleiche Adresse zeigen.

  3. Der Stack wird gewechselt

     Bei einem Wechsel des Threads wird der Stack ausgetauscht. In diesem Fall entspricht der neue Base-Pointer weder dem letzten Stack-Pointer noch der Adresse, auf die er zuvor zeigte. Somit trifft also weder Fall eins noch Fall zwei zu.

  Die Fallunterscheidung macht deutlich, dass es reicht den Base-Pointer sowie den Stack-Pointer zu beobachten, um einen Wechsel des Stacks zu erkennen. Trifft weder Fall eins noch Fall zwei zu, ist der Stack ein anderer.

Damit die Lebenszeiten den Stackframes und Datenstrukturen des aktuellen Stacks zugeordnet werden können, müssen nachgehalten werden, welche Stacks bereits bekannt sind. Dies kann über die Sicherung des letzten Base-Pointers aus jedem Stack geschehen, denn sobald zu einem bereits bekannten Stack gewechselt wird, muss der Base-Pointer für mindestens eine Instruktion genau den letzten bekannten Base-Pointer aus diesem Stack annehmen.

Sofern keine Optimierung angewandt wird, die den Base-Pointer für andere Zwecke verwendet, ist das zuletzt beschriebene Verfahren sowohl im Bezug auf die Benutzerfreundlichkeit, als auch in Hinsicht auf die Zuverlässigkeit am besten geeignet. Sollte dieses Verfahren nicht anwendbar sein, dürfte der heuristische Ansatz für einen Großteil der Fälle funktionieren, da es eher unwahrscheinlich ist, dass ein fast voller Stack direkt im Speicher vor einem nahezu leeren Stack steht. Dabei hängt die Zuverlässigkeit wesentlich von der Mindestdifferenz ab.

Sowohl der manuelle Ansatz, als auch die Rückverfolgung des Base-Pointer sind stark abhängig vom Betriebssystem und entsprechend unflexibel.

4.4.3. Erweiterungen

Mit dem beschriebenen Verfahren zum Aufzeichnen des Stacks lassen sich weitere Informationen sammeln, über die aggregiert werden kann.

Neben den lokalen Variablen können die vom Compiler generierten mit geringem Zusatzaufwand mit aufgezeichnet werden. Beispielsweise ist der Base-Pointer der aufrufenden Funktion leicht durch den aktuellen Base-Pointer zugänglich. Gleiches gilt für den Instruction Pointer der aufrufenden Funktion. Somit können für die Datenstrukturen des Compilers künstliche Variablen angelegt werden, um die Ergebnisdaten darüber zu aggregieren.

Sofern Klassen-Objekte innerhalb eines Stackframe liegen, können diese Member-Variablen enthalten. Über die Debug-Informationen können sowohl Informationen über das Vorkommen solcher Member-Variablen, sowie deren relative Lage zum Klassen-Objekt im Speicher bestimmt werden. Entsprechend kann auch über diese Informationen mit Hilfe des *Stack-Trace* aggregiert werden.

Nicht nur die Variablen selbst, sondern auch deren Typ kommt für eine Aggregation in Frage. Der Typ einer Variable kann ebenfalls aus den Debug-Informationen bezogen werden. Entsprechend werden Aussagen darüber möglich, welche Typen besonders fehleranfällig sind.

Letztlich ist es mit Hilfe des Stack-Trace und den Debug-Informationen möglich über jede Granularitäts-Ebene aus der räumlichen Dimension zu aggregieren und Aussagen über die Fehleranfälligkeit zu treffen.

4.5. Priorisierung der Analyseergebnisse

Die vielfältigen Möglichkeiten, die sich durch die Aggregation über Software-Komponenten ergeben, vereinfachen die Entscheidung, welche Komponenten gehärtet werden müssen,

wesentlich. Dennoch sind es insbesondere für größere Benchmarks viele Daten, die betrachtet und verglichen werden müssen.

Daher ist es erforderlich die aggregierten Software-Komponnenten zu priorisieren. Durch die Priorisierung wird für den Benutzer ersichtlich, welche Software-Komponenten fehleranfälliger sind als andere. Es wird somit also eine Ordnung über die Fehleranfälligkeit hergestellt.

Dazu können unterschiedliche Metriken genutzt werden, die nachfolgend vorgestellt werden:

- **Top-Listen**

  Für jede Granularitätsebene der Informationen aus der räumlichen Dimensionen können die Komponenten nach ihrer Fehleranfälligkeit sortiert werden. Entsprechend erhält man beispielsweise Auskunft darüber, welche Variablen oder Funktionen besonders fehleranfällig sind. Auf welcher Granularitätsebene die Betrachtung einer solchen Top-Liste sinnvoll ist, hängt von den verfügbaren Fehlertoleranzmechanismen ab.

- **Laufzeit und Aufrufhäufigkeit von Funktionen**

  Als weitere Metrik kann die Aufrufhäufugkeit sowie die Laufzeit der Funktionen ermittelt werden und für die Priorisierung genutzt werden.

Neben den beiden genannten Metriken sind weitere denkbar, wie z.B. wie Korrelation der aggregierten Software-Komponenten auf den unterschiedlichen Granularitätsebenen miteinander. Welche Metrik zur Priorisierung herangezogen werden sollte, hängt letztlich von den verfügbaren Fehlertoleranzmechanismen ab.

4.6. Platzierung von Fehlertoleranzmechanismen

Die Grundlage für die Platzierung der Fehlertoleranzmechanismen bilden die priorisierten aggregierten Ergebnisdaten. Die Fehlertoleranzmechanismen können auf Basis der Priorisierung für besonders fehleranfällige Software-Komponenten spezialisiert werden. Allerdings ist eine Härtung von Software-Komponenten aufgrund der entstehenden Kosten nur bis zu einem bestimmten Grad sinnvoll. Des Weiteren kann die Platzierung automatisiert werden.

4.6.1. Kosten der Fehlertoleranz

Wenn ein Fehlertoleranzmechanismus eingesetzt wird, um eine Software-Komponenten zu härten, entstehen dafür Kosten. Im Wesentlichen lassen sich diese Kosten in zwei Kategorien aufteilen:

1. **Zusätzlicher Ressourcenverbrauch**

Durch den Einsatz von Fehlertoleranzmechanismen wird der Ressourcenverbrauch des Programms beeinflusst. Beeinflusst werden vor allem die Faktoren Speicherplatz, Laufzeit sowie Energiebedarf. Alle drei Ressourcen stehen insbesondere in eingebetteten Systemen meist in nur sehr begrenzter Form zur Verfügung.

Während der erhöhte Energiebedarf keine weitere Auswirkungen haben[4], beeinflusst die verlängerte Laufzeit direkt die Fehleranfälligkeit des Programms. Denn während die Fehlertoleranzmaßnahme abgearbeitet wird, gibt es Datenstrukturen, die bereits im Speicher liegen (Beispielsweise durch eine Funktion, die zuvor aufgerufen wurde). Mit der gesteigerten Laufzeit wird es wahrscheinlicher, dass diese Datenstrukturen durch einen Fehler beeinflusst werden.

2. **Fehleranfälligkeit der Fehlertoleranz**

   Fehlertoleranzmaßnahmen verwenden selbst ebenfalls Datenstrukturen, um Software-Komponenten zu härten. Entsprechend resultiert daraus eine zusätzliche Fehleranfälligkeit, da diese Datenstrukturen ebenfalls von Fehlern betroffen sein können.

Aus diesen Kosten kann grob abgeschätzt werden, ob der Einsatz einer Fehlertoleranzmaßnahme in Hinsicht auf die Fehleranfälligkeit sinnvoll ist. Solange die folgende Bedingung erfüllt ist, lohnt sich der Einsatz der Fehlertoleranzmaßnahme:

$$\mathbf{Fehler}_{\mathbf{zus.Laufzeit}} + \mathbf{Fehler}_{\mathbf{der\ FTM}} < \mathbf{Fehlertoleranz}_{\mathbf{durch\ FTM}}$$

Diese Abschätzung setzt voraus, dass die Zahl der Fehler bestimmt werden kann, die zusätzlich durch die verlängerte Laufzeit auftreten können. Dies wäre über das FAIL*-Framework möglich, indem die Veränderung der Länge der Äquivalenzklassen, die durch das Pruning entstehen, betrachtet werden. Eine längere Äquivalenzklasse wäre entsprechend mit einer höheren Fehleranfälligkeit gleichbedeutend.

Zusätzlich ist die Bestimmung der Fehleranfälligkeit der Fehlertoleranzmaßnahme nicht trivial. Die Fehleranfälligkeit der Datenstrukturen innerhalb der Fehlertoleranzmaßnahme lässt sich zwar bestimmen, aber für die Fehleranfälligkeit des Programms sind lediglich deren Auswirkungen relevant. Denn nur solche Fehler, die zu einer negativen Beeinflussung des Toleranzmechanismus führen, sind Fehler, die in der Abschätzung relevant sind.

4.6.2. Automatisierte Platzierung

Um Fehlertoleranzmechanismen automatisch zu platzieren, müssen zwei Kriterien erfüllt sein:

[4] Dies gilt für das in dieser Arbeit betrachtete Fehlermodell.

1. Es muss möglich sein abzuschätzen, ob die Anwendung einer Fehlertoleranzmaßnahme sinnvoll ist.

2. Wenn mehr als eine Fehlertoleranzmaßnahme für eine Software-Komponente in Frage kommt, muss es möglich sein diese nach ihrer Effektivität zu ordnen.

Die vorgestellte Abschätzung erfüllt das erste Kriterium. Dazu muss allerdings eine Abbildung von einer Fehlertoleranzmaßnahme auf konkrete Fehler erfolgen. Es muss also bekannt sein, wie sich der Fehlertoleranzmechanismus auf den konkreten Benchmark auswirkt. Diese Abbildung zu automatisieren ist nicht trivial. Für einen Fehlertoleranzmechanismus, der beispielsweise einen Zeiger auf eine Tabelle virtueller Methoden sichert, ist die automatische Abbildung relativ einfach. Dazu reicht es den Fehlertoleranzmechanismus auf die aggregierten Ergebnisdaten des entsprechenden Zeigers der Funktion abzubilden. Wird jedoch eine Fehlertoleranzmechanismus betrachtet, der Funktionen durch eine dreifache Ausführung mit anschließendem *Voting* schützt, ist eine solche automatisierte Abbildung nicht trivial.

Als Alternative kann der Prozess der Härtung von Software als iterativer Prozess betrachtet werden. Dazu wird nach jeder Platzierung einer Fehlertoleranzmaßnahme erneut die Kampagne durchgeführt. Anhand der Veränderung der Fehlertoleranz lässt sich die Effektivität der angewandten Maßnahme nachvollziehen.

Um die Effektivität der angewandten Fehlertoleranzmechanismen zu bewerten, kann auf die Mechanismen zurückgegriffen werden, die bereits zur Priorisierung verwendet wurden.

4.7. Softwarearchitektur-Entwurf

Im Wesentlichen besteht der Entwurf aus den Komponenten, die bereits in diesem Kapitel erläutert wurden. Daher wird im Folgenden nicht näher auf die einzelnen Komponenten eingegangen. Abbildung 4.3 zeigt ein Komponentendiagramm zum Entwurf. Ersichtlich ist, dass das FAIL*-Framework um eine Komponente erweitert wird, die Debug-Informationen zur Verfügung stellt. Diese Debug-Informationen werden durch eine angepasste Variante des FAIL*-Frameworks dazu verwendet, um den *Stack-Trace* aufzuzeichnen. Das erweiterte FAIL*-Framework stellt somit sowohl die bereits bekannten Ergebnisdaten, als auch den *Stack-Trace* sowie den Assembler– und Hochsprachen-Code zur Verfügung.

Das Analyse-Tool kann somit vollständig vom FAIL*-Framework entkoppelt werden. In der Analyse findet zunächst die Rückabbildung und Aggregation der Ergebnisdaten auf den Hochsprachen-Code statt. Diese Informationen werden durch die Priorisierung genutzt um über Metriken eine entsprechende Strategie zur Platzierung von Fehlertoleranzmechanismen vorzuschlagen. Der Benutzer kann auf alle Informationen, die durch die einzelnen Komponenten im Analyse-Tool erstellt werden, zugreifen um eigene Untersuchungen vorzunehmen.

4.8. Zusammenfassung

In diesem Kapitel wurde zunächst analysiert, welche Anforderungen an neue Analyse-Methoden existieren. Aufbauend auf den Anforderungen wurde ein Lösungsansatz vorgestellt, der eine Hinzunahme von Debug-Informationen vorsieht. Anschließend wurden die verfügbaren Informationen in drei Dimensionen kategorisiert und eine Terminologie eingeführt. Auf Basis dieser Kategorisierung wurden die einzelnen Anforderungen genauer untersucht. Die Untersuchung hat ergeben, dass es möglich ist die Daten im Stack-Segment des Speichers zu aggregieren. Dadurch können Aussagen über die Fehleranfälligkeit jeder Software-Komponente in beliebiger Granularität getroffen werden. Mit diesen aggregierten Daten kann über eine Priorisierung, die Platzierung bzw. Spezialisierung von Fehlertoleranzmaßnahmen durchgeführt werden. Beim Ansatz dieses Verfahren zu automatisieren, hat sich herausgestellt, dass die notwendige Abbildung von Fehlertoleranzmaßnahmen auf konkrete Fehler nicht trivial ist. Abschließend wurde eine Architektur entworfen, die über es über zwei entkoppelte Komponenten ermöglicht, die neuen Analyse-Verfahren umzusetzen.

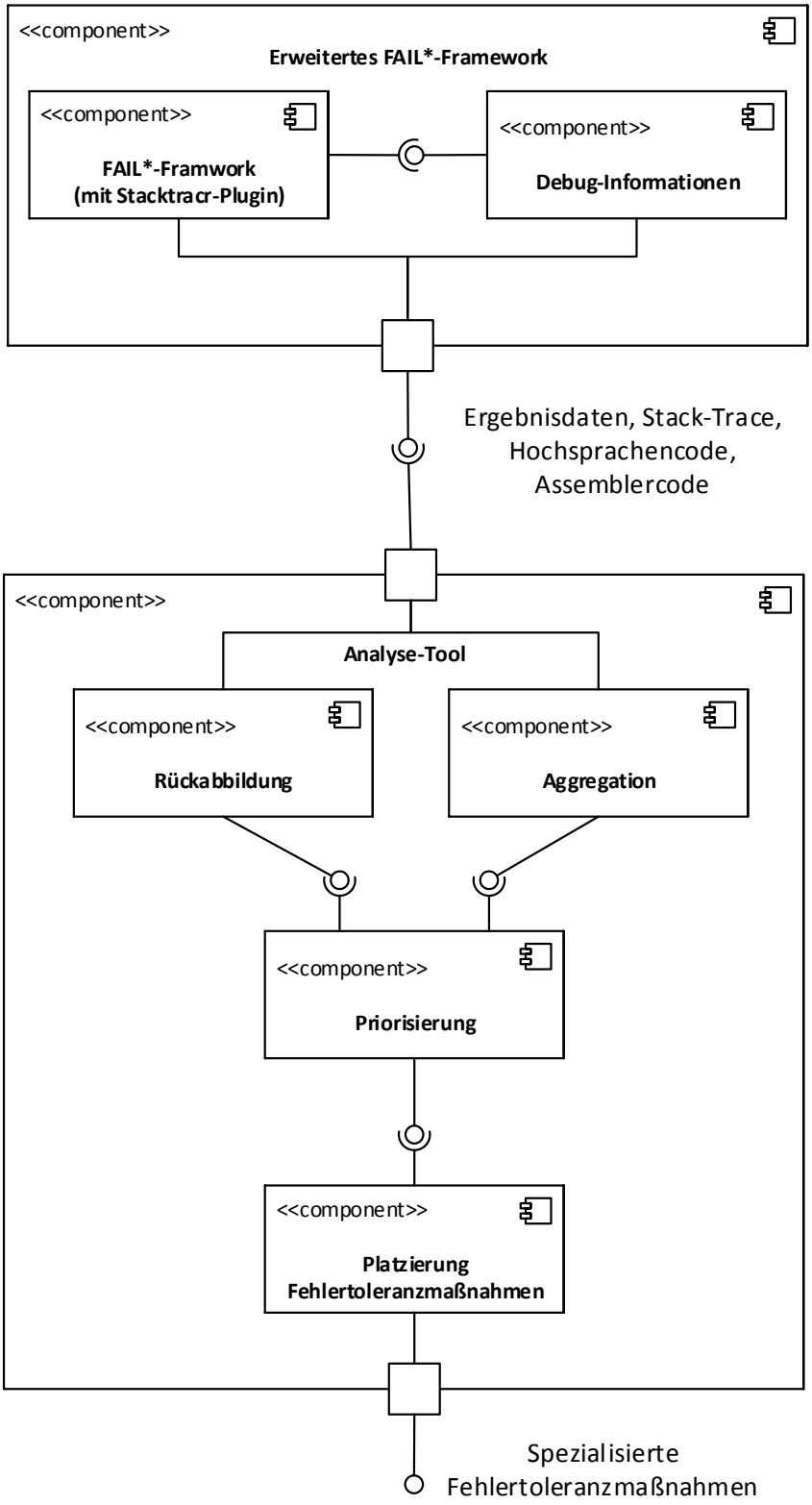

Abbildung 4.3.: UML–Komponentendiagramm des Entwurfs für die neuen Analysen und der Platzierung der Fehlertoleranzmechanismen

5. Implementierung

In diesem Kapitel wird die prototypische Implementierung der neuen Analyse-Verfahren aus dem Kapitel „Analyse und Entwurf“vorgestellt. Dazu wird zunächst auf die Technologieauswahl eingegangen. Insbesondere wird die Auswahl begründet und kurz erläutert, wie die entsprechende Technologie funktioniert. Anschließend wir darauf eingegangen, welche Anpassungen am FAIL*-Framework notwendig sind, um die benötigten Informationen für das Analyse-Tool bereitzustellen. In Kapitel 5.3 wird die Implementierung des Analyse-Tools *VisualFAIL** vorgestellt. Abschließend wird in Kapitel 5.4 darauf eingegangen, welche Probleme bei der Implementierung aufgetreten sind und wie diese gelöst werden können.

5.1. Technologieauswahl

Für die Anpassungen am FAIL*-Framework ist es notwendig den Zugriff auf die Debug–Informationen zu ermöglichen. Dazu wird im Folgenden die Library *libdwarf* vorgestellt.

Die Implementierung des Analyse-Tools erfolgt im Rahmen dieser Arbeit als Web-Applikation. Der Grund dafür liegt darin, dass im Laufe der Zeit viele Frameworks entwickelt wurden, die eine vielfältige Visualisierung von Daten erlauben. Somit sind die Möglichkeiten der Visualisierung der neuen Analyse-Methoden durch eine Web-Applikation einfacher und Vielfältiger, als es bei einer Implementierung als normale Anwendung der Fall wäre. Darüber hinaus entfällt der Übersetzungsvorgang. In Kapitel 5.1.2 wird auf die verwendeten Technologien eingegangen.

5.1.1. libdwarf

Damit die neuen Analyse-Verfahren über das FAIL*-Framework implementiert werden können, müssen die Debug-Informationen einer Binärdatei zugänglich sein. Da das DWARF-Format sehr komplex ist, wird in der Implementierung die Library *libdwarf* [40] aus der ELF-Toolchain eingesetzt. Diese Library bietet im Wesentlichen eine Möglichkeit über die Baumstruktur der Debug-Informationen zu iterieren. Dazu werden Funktionen zur Verfügung gestellt, die es erlauben zu prüfen, ob ein DIE[1] Geschwister– bzw. Kind–Elemente hat. Jedes Attribut in dem DIE kann abgerufen werden. Dazu existieren für wenige Attribute Spezialfunktionen, wie z.B. für das Attribut *DW_AT_lowpc* oder den Namen des DIE. Für alle anderen Attribute kann der Wert über eine Funktion abgefragt

[1] Zur Erinnerung: DIE steht für Debug Information Entry

werden, der als Eingabe den Attribut-Namen erhält.

Um die Funktionsweise der Library zu verdeutlichen[2], ist nachfolgend ein Code-Ausschnitt zu sehen, in dem der Name und die Größe von Variablen ermittelt wird:

Listing 5.1: Libdwarf–Beispiel

```
1  //Welchen Typ hat der aktuelle DIE?
2  if (dwarf_tag(die, &tag, 0) == DW_DLV_OK) {
3      //Ist DIE vom Typ DW_TAG_variable oder DW_TAG_formal_parameter?
4      if (tag == DW_TAG_variable || tag == DW_TAG_formal_parameter) {
5          //Speicher Namen des DIE, sofern vorhanden in ret_name
6          if (dwarf_diename(die, &ret_name, 0)==DW_DLV_OK) {
7              //Der Name des DIE steht nun in ret_name
8
9              //Hat der DIE ein DW_AT_type Attribut?
10             if(dwarf_attr(die, DW_AT_type ,&atp , 0)==DW_DLV_OK) {
11                 //Übersetze die Referenz des Typen in einen Offset
12                 if(dwarf_global_formref(atp, &retaddr,0)==DW_DLV_OK) {
13                     //Springe an den Offset des Typen
14                     if(dwarf_offdie(dbg, retaddr, &type_die,
                           0)==DW_DLV_OK) {
15                         //Speicher die Größe des Types in byte_size
16                         dwarf_bytesize(type_die, &byte_size, 0);
17                     }
18                 }
19             }
20         }
21     }
22 }
```

Des Weiteren ermöglicht es die *libdwarf* auch auf Informationen zur Rückabbildung auf Hochsprachen-Code sowie auf die Location-List Informationen zuzugreifen. Allerdings muss auch dafür über die Baum-Struktur der Debug-Informationen iteriert werden.

5.1.2. Web-Technologien

Bootstrap

Das sehr mächtige Framework Bootstrap wird von Twitter entwickelt und besteht im Wesentlichen aus drei Teilen [41]:

- **CSS-Komponenten**

  Bootstrap bietet für viele Anwendungsfälle bereits fertige CSS-Komponenten an. Beispielsweise sind Styles für Komponenten wie Buttons, Formulare oder Tabellen enthalten. Dabei sind diese stark anpassbar. Die Komponenten lassen sich z.B. in

[2]Es sei an dieser Stelle angemerkt, dass die Library libdwarf schlecht bis gar nicht dokumentiert ist. Lediglich unvollständige Manpages dienen als „Dokumentation“

der Größe und Farbe ändern, ohne, dass man selbst in den CSS-Code eingreifen muss.

Die mächtigste CSS-Komponenten ist das Grid-System. Über das Grid-System lässt sich das gesamte Layout und Verhalten der Seite steuern. Dabei wird das Layout über sogenannte *Container* unterteilt. Die Breite eines solchen Containers wird über sogenannte *Rows* gesteuert. Eine *Row* hat eine feste Breite. Entsprechend gibt es eine begrenzte Anzahl Rows, die auf die Container, die nebeneinanderliegen aufgeteilt werden können.

Die CSS-Komponenten sind so geschrieben, dass sie für alle gängigen Browser kompatibel sind und entsprechend gleich aussehen. Darüber hinaus können mit geringen Anpassungen gleich mehrere Styles für unterschiedliche Bildschirm-Auflösungen angegeben werden.

- **Bedienelemente**

  Die Bedienelemente umfassen alles von Symbolen für Buttons über Buttons selbst bis hin zu einer kompletten Navigationsleiste. Auch diese Komponenten sind stark anpassbar.

- **Javascript-Komponenten**

  Die Javascript-Komponenten stellen beispielsweise modale Dialoge, Dropdown-Funktionen sowie Alerts und Tooltips.

Der Umfang von Bootstrap ist sehr groß und vielfältig. Es wird in dieser Arbeit verwendet, da es durch die vielen Komponenten einen Entwurf der Webseite in reinem HTML oder CSS überflüssig macht.

JQuery

JQuery ist ein mächtiges Javascript-Framework [42]. Im Fokus steht dabei die Manipulation des *Document Object Model* (DOM). JQuery ermöglicht es über sogenannte Selektoren beliebige Elemente des DOM zu selektieren. Anschließend können auf dieser Selektion nahezu beliebige Manipulationen durchgeführt werden. Ebenso lassen sich beliebige DOM-Elemente erstellen und in der vorhandenen Struktur einbinden.

Beispielsweise würde die Zeile „$("div").addClass("blue");“dazu führen, dass alle div-Elemente im DOM die Klasse „blue“bekommen. Diese könnte wiederum über CSS so gestaltet werden, dass alle Elemente mit dieser Klasse blau werden.

Die Selektoren können sehr komplex werden und entsprechend nahezu beliebige Elemente des DOM selektieren.

Zusätzlich bietet JQuery Funktionen, um über Arrays zu iterrieren sowie Funktionen zur Kommunikation via Ajax mit anderen Diensten.

JQuery erleichtert durch die Möglichkeit DOM-Elemente zu erstellen und zu manipulieren das Darstellen von Daten. Beispielweise kann mit nur einer JQuery-Zeile eine Auswahl von Zeilen eines Hochsprachen-Quellcodes gefärbt werden.

PHP
PHP ist eine Skriptsprache zur Erstellung von Webseiten bzw. Webanwendungen. Es gibt u.a. eine Schnittstelle, um mit Datenbanken zu kommunizieren oder Daten über Ajax auszutauschen.

PHP wird im Rahmen dieser Arbeit eingesetzt, um die Kommunikation mit der Datenbank zu implementieren. Ebenfalls wird über PHP die Aufbereitung der Daten übernommen, um diese letztlich über JQuery darzustellen.

5.2. FAIL* Anpassungen

Die Anpassungen an der Struktur des FAIL*-Frameworks sind in Abbildung 5.1 als UML-Klassendiagramm dargestellt. Die grün gekennzeichneten Klassen sind bereits Teil des Frameworks. Die blau markierten Klassen *DwarfReader*, *Stacktracer* sowie *FulltraceImporter* sind Klassen, um die das Framework erweitert wurde, um die neuen Analyseverfahren zu ermöglichen. Die Klasse *ELFImporter* existierte bereits. Allerdings wurde sie im Rahmen dieser Arbeit angepasst. Im Folgenden wird auf jede zusätzliche sowie geänderte Klasse eingegangen:

- **DwarfReader**

  Die Klasse DwarfReader ist eine weitere Abstraktion der *libdwarf*-Library. Da der Code bzgl. der Dwarf Debuginformationen aufgrund des strukturellen Aufbaus der *libdwarf* recht unübersichtlich und umfangreich ist, wurde dieser in eine eigene Klasse ausgelagert. Die Klasse DwarfReader stellt Funktionen zur Verfügung, die sowohl vom StackTracer, als auch vom ELFImporter genutzt werden. Folgende Funktionen und Strukturen stellt die Klasse DwarfReader zur Verfügung:

  – *struct addrToLine*

    addrToLine ist die Struktur, die eine Rückabbildung von einer Instruktions-Adresse auf eine Zeile in einer Quellcode-Datei enthält. Entsprechend enthält sie eine Adresse, eine Zeilennummer und einen Pfad einer Quellcodedatei.

  – *struct loclistentry*

    In der Struktur loclistentry werden die *location*-Informationen aus den Debug-Informationen abgebildet. Den Orts-Informationen entsprechend enthält diese Struktur die statische Start– und Endadresse, für die der Eintrag gültig ist, sowie das Register und den Offset. Das Register bezieht sich auf die Nummerierung in der libdwarf. Entsprechend muss eine Übersetzung des libdwarf-Registers auf ein FAIL*-Register erfolgen. Ob die location-Informationen, wie im Kapitel Grundlagen beschrieben, partiell ist, wird in einem Boolean festgehalten. Die Nutzung von partiell definierten *location*-Informationen ist in

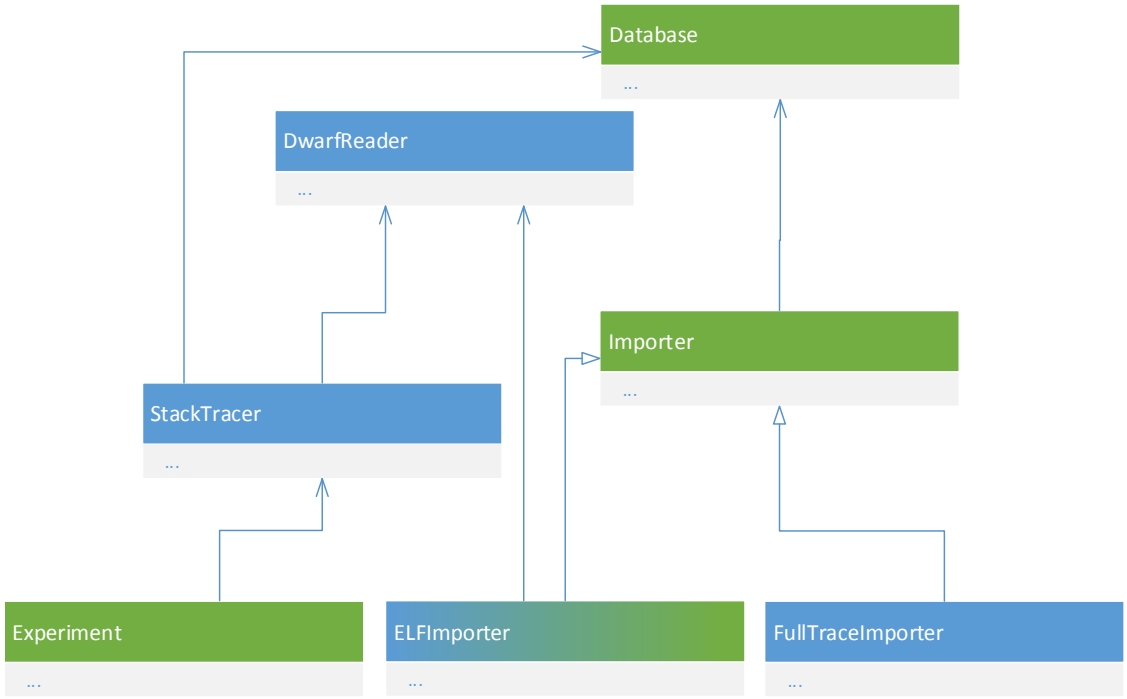

Abbildung 5.1.: UML–Klassendiagramm bzgl. der Anpassungen am FAIL*–Framework. Klassen, die zum FAIL*-Framework gehören sind grün markiert. Klassen, die durch die neuen Analyse-Verfahren hinzukommen sind blau markiert. Auf die Ausführung der Variablen und Methoden wurde zwecks Übersichtlichkeit verzichtet.

diesem Prototypen nicht vorgesehen, da es die Abbildung dieser Informationen erheblich erschweren würde.

– *struct variable*

  In der Struktur variable werden, wie der Name schon sagt, Variablen abgebildet. Für jede Variable wird festgehalten : Name, Größe sowie die *location*-Informationen. Letztere werden über std::vector<loclistentry> abgebildet. Diese Struktur ist so konzipiert, dass sie auch Informationen enthält, die erst zur konkreten Laufzeit bekannt werden. Entsprechend werden die Informationen dynamische Start– und Endadresse sowie der Index in der Datenbank und die Adresse ebenfalls durch diese Struktur abgebildet. Somit kann diese Struktur während der Aufzeichnung des *Stack-Trace* kopiert werden und mit konkreten Laufzeit-Informationen befüllt werden.

– *struct method*

  Die Struktur variable und method sind sehr ähnlich aufgebaut. Alle Informationen, die variable enthält, sind auch in der Struktur method enthalten. Zusätzlich wird die statische Start– und Endadresse sowie die lokalen Variablen über std::vector<variable> gespeichert.

– *read_source_files()*

  Diese Funktion erhält als Parameter einen Pfad zu einer Datei und eine Referenz auf eine Liste vom Typ string. Die Liste wird mit den Pfaden der Quellcode-Dateien gefüllt, die am Compile-Vorgang des Programms beteiligt waren. Ein Boolean Rückgabewert gibt den Erfolg der Funktion an.

– *read_mapping()*

  Die Funktion *read_mapping()* erhält als Parameter einen Pfad zu einer Quellcode-Datei und eine Liste vom Typ *loclistentry*. Die Liste wird dann mit allen Rückabbildungs-Informationen befüllt, die diese Quellcode-Datei betreffen. Ein Boolean Rückgabewert gibt den Erfolg der Funktion an.

– *get_location_information()*

  Diese Funktion hat drei Parameter: der Pfad des Programms, eine Referenz auf eine Struktur vom Typ method und eine Instruktionsadresse. Der Pfad des Programms wird benötigt, damit auf die Debug Informationen zugegriffen werden kann. Die Struktur wird den Informationen der Funktion befüllt, die an der übergebenen Adresse liegt. Entsprechend werden neben den location-Informationen auch lokale Variablen mit den zugehörigen Informationen in der Struktur gespeichert. Diese Funktion stellt dem Stack-Tracer alle Debug-Informationen zur Verfügung, die er braucht. Ein Boolean Rückgabewert gibt den Erfolg der Funktion an.

– *print_struct()*

  Um zu überprüfen, welche Informationen in einer *method*-Struktur stehen, kann diese Funktion genutzt werden. Sie gibt alle Informationen aus, die in

der Struktur enthalten sind.

- **StackTracer**

  Die StackTracer-Klasse ist ein FAIL*–Plugin, dass die Aufzeichnung der Bewegungen im Stack-Segment vornimmt. Die notwendigen Debug-Informationen werden durch die Klasse DwarfReader bezogen. Die Aufzeichnungen werden über die Klasse Database in die Datenbank importiert. Eine detaillierte Erläuterung des Ablaufs erfolgt in Kapitel 5.2.1.

- **ELFImporter**

  Die ELFImporter-Klasse existierte bereits im FAIL*–Framework und bietet die Möglichkeit unterschiedlichste Informationen einer ELF-Binärdatei, wie z.B. den Assembler-Code, in die Datenbank zu importieren. Diese Klasse wird nicht gebraucht, um Experimente oder Aufzeichnungen durchzuführen. Sie dient lediglich zum Import von Informationen in die Datenbank.

  Im Rahmen dieser Arbeit wurde diese Klasse um die Möglichkeit erweitert, Rückabbildungs–Informationen in die Datenbank zu importieren. Dazu wird auf die DwarfReader–Klasse zugegriffen. Über die Funktionen *read_source_files()* und *read_mapping()* werden die Debug-Informationen abgerufen und anschließend importiert.

- **FullTraceImporter**

  Für die Implementierung im Rahmen dieser Arbeit wurde auch die Möglichkeit der Betrachtung von Äquivalenzklassen, wie sie im Grundlagen-Kapitel diskutiert wurde, praktisch untersucht. Um eine Abbildung von Äquivalenzklassen auf statische Instruktionen vornehmen zu können, müssen die gesamten statischen Instruktionen in der Datenbank stehen. Da dies bislang aufgrund des Pruning nicht notwendig war, wurde mit der Klasse *FullTraceImporter* eine Möglichkeit geschaffen, um jede statische Instruktions-Adresse aus dem *Trace* in die Datenbank zu importieren.

5.2.1. StackTracer-Plugin

Das StackTracer-Plugin verwendet die Debug-Informationen, die durch die Klasse DwarfReader bereitgestellt werden. Auf Basis dieser Debug-Informationen wird der Stack-Trace erstellt. Der Grundlegende Ablauf ist in einem Flussdiagramm in Abbildung 5.2 dargestellt.

Als erstes wird über einen *Breakpoint*-Listener auf die Änderung des Instruktion-Pointer gewartet. Danach wird geprüft welcher Stack aktuell genutzt wird (siehe Multi-Stack-Problem in Kapitel 4.4.2).Anschließend wird, auf Basis der Debug-Informationen, versucht zu ermitteln, um welche Funktion es sich handelt. Dabei werden drei Fälle unterschieden:

1. **Die Funktion ist unverändert**

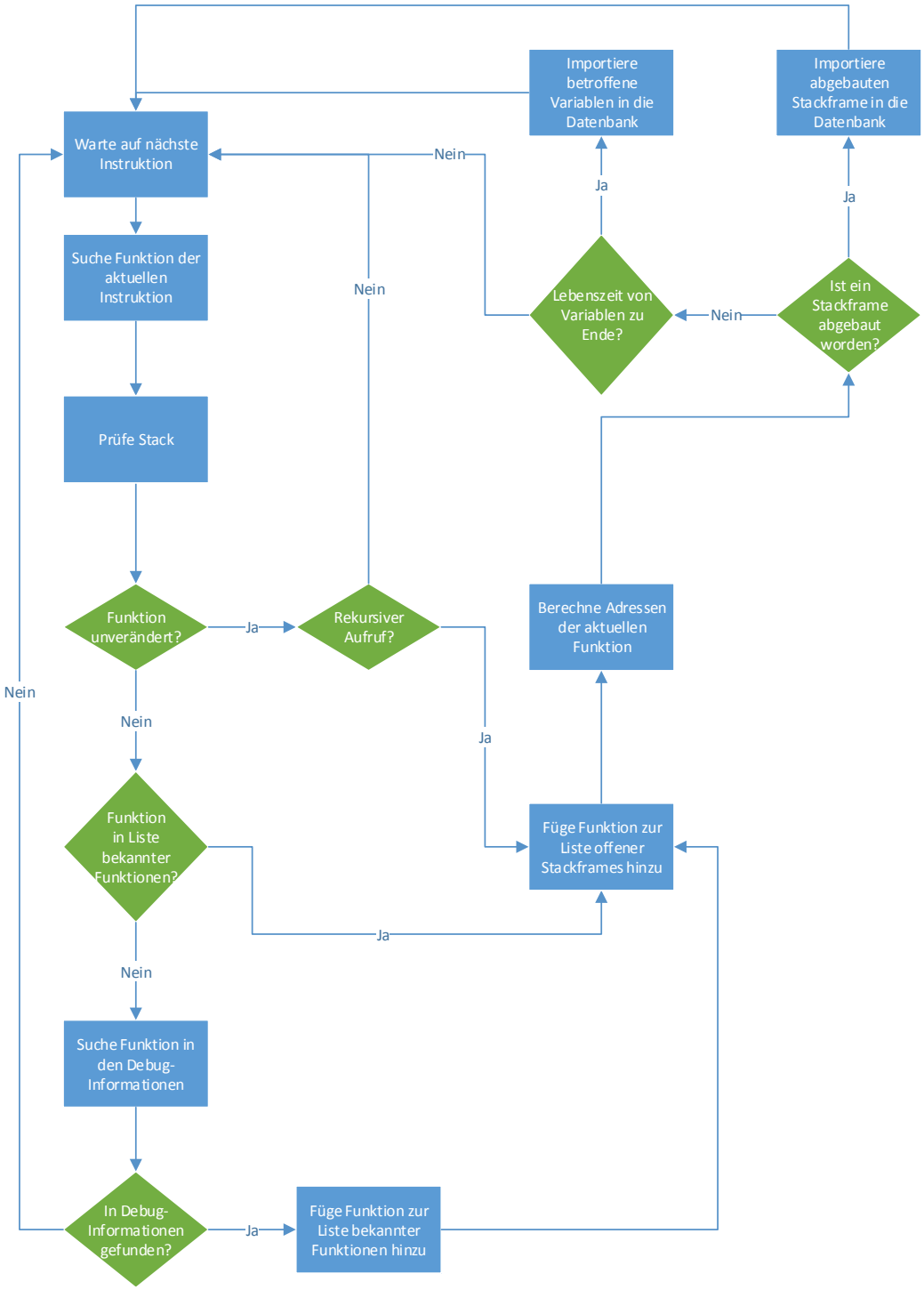

Abbildung 5.2.: Flussdiagramm zur Darstellung des grundlegenden Ablaufs im StackTracer-Plugin

Die aktuelle Instruktion betrifft die gleiche Funktion, die bereits bei der Funktion zuvor zutraf. Entsprechend stehen die Debug-Informationen zu dieser Funktion bereits zur Verfügung.

In diesem Fall muss untersucht werden, ob es sich um einen rekursiven Aufruf der Funktion handelt. Festgestellt werden kann dies über die framebase, die aus den Debug-Informationen berechnet werden kann. Hat sich die framebase geändert, handelt es sich um einen rekursiven Aufruf.

2. **Die Funktion ist bereits bekannt**

   Alle Funktionen, die im Programmverlauf betreten wurden, werden in einer Liste gespeichert. Dazu werden die gefüllten Strukturen aus der Klasse DwarfReader verwendet.

   Sofern sich die Funktion geändert hat, wird zunächst diese Liste durchlaufen und überprüft, ob die Debug-Informationen zu dieser Funktion bereits vorliegen.

   Dieses Vorgehen wurde zur Steigerung der Effizienz eingeführt, da ein Abfragen der Debug-Informationen aus der Binär-Datei mehr Laufzeit-Kosten verursacht, als diese Liste zu durchlaufen.

3. **Die Funktion wird in den Debug-Informationen gesucht**

   Sofern die Suche in der Liste erfolglos ausging, müssen die Debug-Informationen aus der Binär-Datei geholt werden. Dazu wird eine Anfrage über die Funktion *get_location_information()* der Klasse DwarfReader gestellt.

   Sofern Debug-Informationen in der Binär-Datei vorlagen, werden diese der Liste der bekannten Funktionen hinzugefügt.

   Es kann z.B. im Fall einer Funktion, die in Assembler geschrieben wurde, vorkommen, dass keine Debug-Informationen gefunden werden können. In diesem Fall kann keine Aufzeichnung des Stackframe erfolgen und es wird erneut auf eine Änderung der Instruktion gewartet.

Sofern der aktuellen Instruktion eine Funktion zugeordnet werden konnte, wird diese als Kopie einer weiteren Liste hinzugefügt. Diese Liste enthält alle offenen Stackframes. Zusätzlich wird abgelegt, zu welchem Stack die Stackframe gehört. Im restlichen Verlauf des StackTracer-Plugins werden immer ledliglich Stackframes betrachtet, die zum aktuellen Stack gehören.

Anschließend werden die Adressen der offenen Stackframes berechnet. Dies umfasst nicht nur die Berechnung der framebase, sondern auch die Berechnung der Adressen der lokalen Variablen der Stackframe.

Dabei unterscheidet sich die Berechnung der framebase von der Berechnung der Adressen für die lokalen Variablen. Die framebase für die Stackframe muss lediglich ein einziges Mal für ihre gesamte Lebensdauer berechnet werden. Für lokale Variablen existieren mehrere Adressbereiche mit teils unterschiedlichen Berechnungsvorschriften (siehe Grundlagen Kapitel 2.5.2). Daher wird eine Variable in die Datenbank geschrieben, sobald ihre Lebenszeit endet.

Sobald eine Stackframe abgebaut wurde, wird diese in die Datenbank geschrieben. Alle Variablen, die zu dieser Funktion gehören und noch nicht in die Datenbank geschrieben wurden, werden ebenfalls importiert.

Abschließend wird erneut auf eine Änderung des Instruktion-Pointer gewartet.

5.2.2. Datenbank-Struktur

Um die Debug-Informationen zu speichern, wurde die Struktur der Datenbank angepasst. Abbildung 5.3 zeigt die neue Struktur. Zur Steigerung der Übersichtlichkeit wurde auf die Darstellung der bereits existierenden Tabellen verzichtet. Ebenso ist der globale Zusammenhang der Tabellen über die *variant_id* nicht eingezeichnet. Folgende Tabellen sind hinzugekommen:

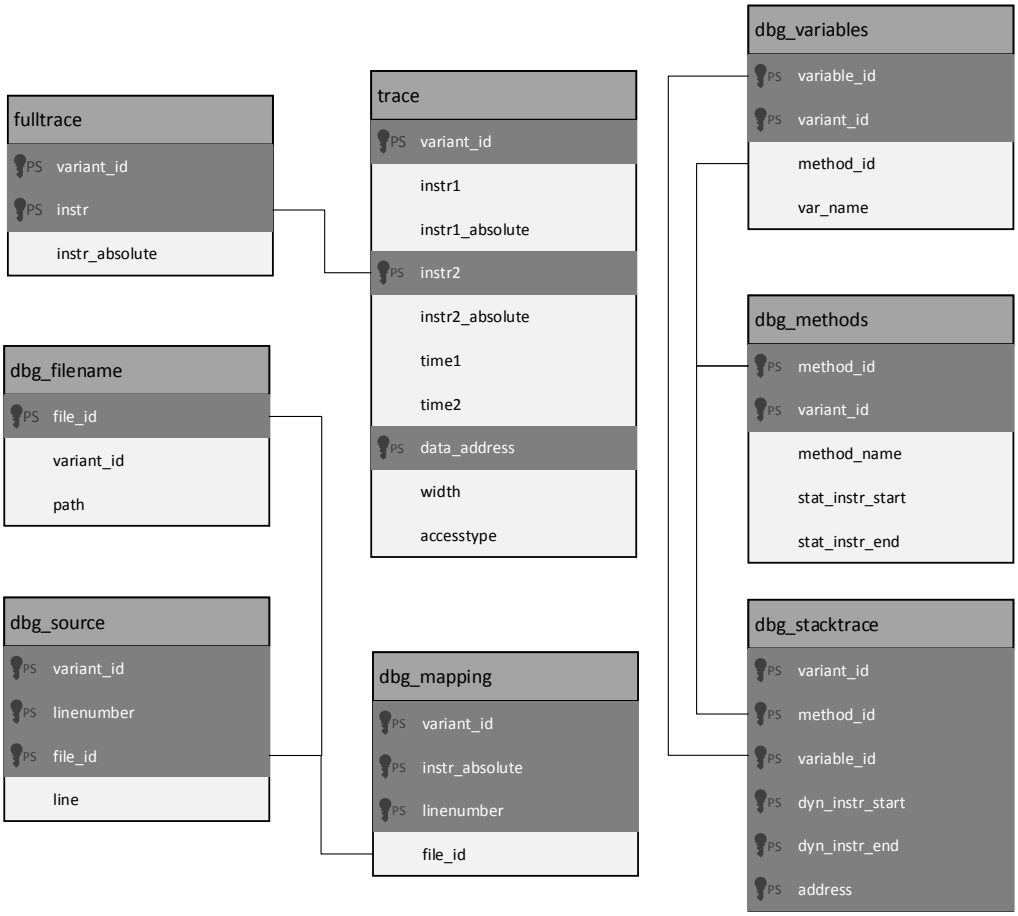

Abbildung 5.3.: Angepasstes Schema der Datenbank für die Debug-Informationen. Alle Komponenten sind über die *variant_id* verbunden. Zum Zweck der Übersichtlichkeit wurde auf diese Verbindungen sowie die bereits existierenden Tabellen verzichtet(ausnahme trace).

- **fulltrace**

  Die Tabelle *fulltrace* wird durch den Fulltrace-Importer befüllt. Sie enthält jede dynamische Instruktion sowie die entsprechende statische Instruktion aus dem Trace.

- **dbg_filename**

  Diese Tabelle wird vom *ElfImport* befüllt. Das Feld *path* enthält den Pfad zu einer Quellcodedatei, die über *file_id* durchnummeriert werden.

- **dbg_source**

  Die *dbg_source*-Tabelle enthält zeilenweise den Quellcode der Quellcode-Dateien. Diese Tabelle wird ebenfalls durch den ElfImporter befüllt.

- **dbg_mapping**

  Die Abbildung von Instruktionen auf Zeilen des Hochsprachen-Code wird in der Tabelle *dbg_mapping* abgelegt. Auch diese Informationen werden durch den ElfImporter importiert.

- **dbg_methods**

  Die Funktionen, die während der Aufzeichnung des Stack-Trace ausgeführt werden importiert das StackTracer-Plugin und legt sie jede Funktion ein einziges Mal in dieser Tabelle ab. Der Name sowie die statische Start- sowie Endadresse werden für jede Funktion in der Datenbank abgelegt.

- **dbg_variables**

  Die lokalen Variablen einer Funktion stehen in der Tabelle *dbg_variables*. Dabei wird für jede Variable gespeichert, wie sie heißt und welchen Typ sie hat. Auch diese Tabelle wird durch das StackTracer-Plugin befüllt.

- **dbg_stacktrace**

  Der eigentliche Stack-Trace, der durch das StackTracer-Plugin aufgezeichnet wird, steht in der Tabelle *dbg_stacktrace*. Jedes Mal, wenn eine Variable auf dem Stack liegt und durch die Debug-Informationen beschrieben ist, werden Einträge für diese Variable in dieser Tabelle angelegt. Für jede Variable wird pro definierter Lebendauer in den Location-Informationen ein Eintrag in der *dbg_stacktrace* Tabelle angelegt.

Aus dem Datenbank-Schema wird ersichtlich, dass die Tabellen in drei Gruppen unterteilt werden können. Die Tabelle *fulltrace* enthält die Daten aus dem *Fulltrace*-Importer und enthält alle Instruktions-Adressen, die der *Trace* enthält. Die drei Tabellen *dbg_filename*, *dbg_source*, *dbg_mapping* enthalten die Debug-Informationen, die dazu notwendig sind, um eine Instruktions-Adresse auf eine Zeile im Hochsprachen-Code abzubilden. Sie werden durch den *ElfImporter* erzeugt und befüllt. Die Tabellen *dbg_methods*,

dbg_variables, *dbg_types* sowie *dbg_stacktrace* enthalten den *Stack-Trace*. Die konkreten *location*–Informationen aus den Debug-Informationen werden nicht in der Datenban abgelegt, da sie nur zum Zeitpunkt der Erzeugung des *Stack-Trace* benötigt werden. Das Datenbank-Schema ist so angelegt, dass es leicht erweitert werden kann. Beispielsweise könnten Klassen als eigene Tabelle hinzugefügt werden. Die dadurch relevante *class_id* könnte als zusätzliches Feld in die Tabellen *dbg_variables* sowie *dbg_methods* eingefügt werden. So könnten *member*–Variablen sowie zur Klasse gehörende Funktionen entsprechend abgelegt werden. Ähnlich könnten die Typen der Variablen in der Datenbank abgelegt werden. Über eine eigene Tabelle *dbg_types* könnten die Typ-Informationen gespeichert werden. Über die ID der Typen könnte in dbg_variables jeder Variable ein Typ zugeordnet werden. Analog könnten weitere Konstrukte des Hochsprachen-Code in der Datenbank abgebildet werden.

5.3. VisualFAIL*

*VisualFAIL** ist ein webbasiertes Analyse-Tool, welches im Rahmen dieser Arbeit implementiert wurde. Es setzt auf die Daten auf, die mit Hilfe des angepassten FAIL*-Frameworks erzeugt werden können. In Abbildung 5.4 ist der Aufbau von *VisualFAIL** veranschaulicht.

Die Server-Seite wurde durch die Skriptsprache PHP implementiert. Der Server holt die notwendigen Daten mit geeigneten Abfragen aus der Datenbank und bereitet die Daten so weit es geht für den Client vor.

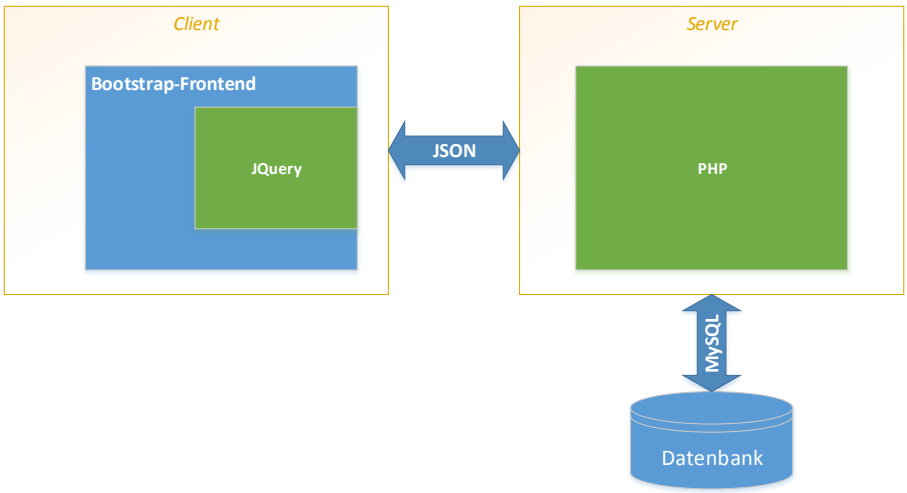

Abbildung 5.4.: Server-Client-Architektur von *VisualFAIL**.

Der Client besteht aus einen Bootstrap-Frontend, dessen Funktionalität komplett durch JQuery implementiert wurde. Da die Daten durch den Server so aufbereitet werden, dass die lediglich auf der Client-Seite dargestellt werden müssen, ist der Rechen-

aufwand für den Client gering. Diese Entscheidung zur Lastenverteilung wurde aktiv getroffen, da Operationen auf großen Daten in PHP erheblich schneller abgearbeitet werden, als es bei JQuery bzw. JavaScript der Fall wäre.

Die Kommunikation zwischen Server und Client erfolgt über *JavaScript Object Notation (JSON)*. Daten, die im JSON-Format ausgetauscht werden sind sowohl für Menschen, als auch für Maschinen einfach lesbar. Insbesondere zum Debuggen stellt dies eine Erleichterung dar.

Der Client stellt in Abhängigkeit von dem was der Benutzer über das Frontend auswählt eine Anfrage an den Server. Der Server stellt wiederum Services in Form von Funktionen bereit. Beispielsweise die Funktion „*function dbTest()*“. Beim Aufrufen der Funktion prüft der Server, ob alle notwendigen Tabellen in der Datenbank vorhanden sind. Ist dies der Fall wird über ein „echo *json_encode('ok');*“die Antwort an den Client übermittelt. Der nimmt die Daten entgegen und prüft, ob die Daten ein „ok“enthalten. Sind nicht alle Tabellen vorhanden wird statt eine „ok“eine passende Meldung an den Client übermittelt. Diese wird dann entsprechend als Warnung über das Bootstrap-Frontend ausgegeben.

Neben einfachen Zeichenketten, können beliebig komplexe Datenstrukturen wie z.B. ungleichmäßig verschachtelte Arrays übergeben werden. Davon ist allerdings aufgrund des hohen Aufwandes zur Verarbeitung der Daten auf der Client-Seite abzuraten.

5.3.1. Features

In VisualFail* wurden zwei Visualisierungen implementiert:

- **Rückabbildung**

  Für diese Visualisierung wurden die Rückabbildungs-Informationen aus den Debug-Informationen genutzt, um die Fehler auf den Assembler-Code bzw. den Hochsprachen-Code abzubilden. Für die Abbildung auf den Hochsprachen-Code wird für jede Zeile der zugehörige Assembler-Code eingeblendet.

  Die Ergebnisdaten der Fehlerinjektions-Experimente werden durch eine rot Färbung der betroffenen Zeile kenntlich gemacht. Entsprechend ist eine Zeile umso stärker rot gefärbt, desto mehr Fehler dieser Zeile zuzuordnen sind. Dabei steigt der rot-Wert nicht linear, sondern logarithmisch zu den Fehlerzahlen der Quellcode-Zeilen. Dadurch soll eine bessere Übersicht über die Fehleranfälligkeit der einzelnen Zeilen erreicht werden.

- **Aggregation**

  Durch die Möglichkeit der Aggregation der Ergebnisdaten über Software-Komponenten können für jede lokale Variable konkrete Zahlen zu den Fehlern angegeben werden, die diese Komponenten betreffen.

  Im Rahmen dieser Implementierung wurde eine tabellarische Übersicht über alle lokalen Variablen implementiert. Diese Tabelle ist absteigend nach Anzahl der Fehler sortiert. Somit ist schnell ersichtlich, welche Variablen besonders fehleranfällig sind.

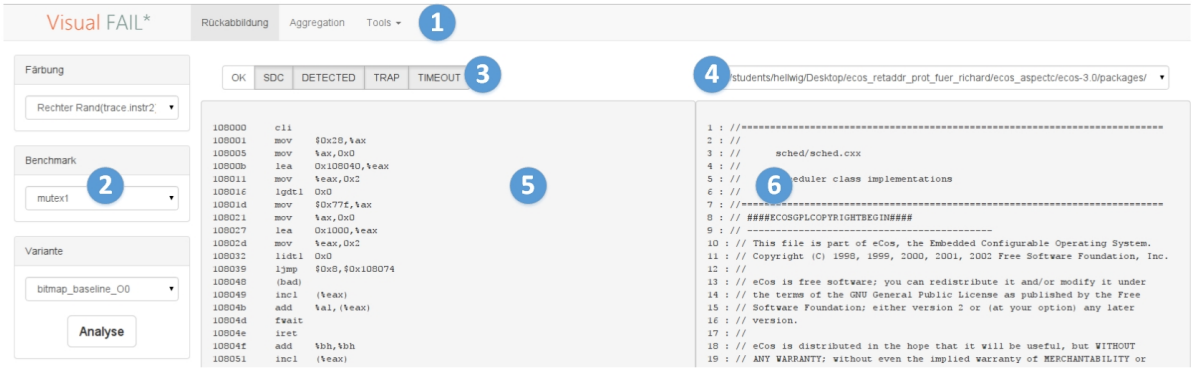

Abbildung 5.5.: Überblick über *VisualFAIL**

Abbildung 5.5 zeigt einen Ausschnitt des Frontend von VisualFAIL*. Zu sehen ist die Oberfläche zur Rückabbildung der Ergebnisdaten auf den Assembler- bzw. Hochsprachen-Code:

1. **Navigationsleiste**

   Über die Navigationsleiste kann zwischen der Rückabbildung und der Darstellung der aggregierten Ergebnisdaten gewechselt werden.

2. **Auswahlmenü**

   Über das Auswahlmenü auf der linken Seite kann ausgewählt werden, welcher Benchmark und welche Variante des Benchmarks visualisiert werden sollen. Außerdem kann für die Rückabbildung ein Verfahren zur Färbung gewählt werden. Möglich ist die Färbung des rechten Randes einer Äquivalenzklasse, die letzte Instruktion sowie komplette Äquivalenzklassen (siehe Kapitel zur Rückabbildung).

3. **Fehlerklassen**

   Nach Auswahl der Komponenten im Auswahlmenü erscheint eine Auswahl zu den Fehlerklassen. Diese ermöglicht es die Fehlerklassen, die in die Visualisierung mit einbezogen werden, zu selektieren. Entsprechend können über die Visualisierung auch einzelne Fehlerklassen betrachtet werden.

4. **Auswahl der Quellcode-Datei**

   Da die Einblendung des gesamten Hochsprachen-Code zu unübersichtlich wäre, gibt es die Auswahl der Quellcode-Datei.

5. **Assembler–Code**

   Der Assembler–Code wird immer komplett eingeblendet und enthält die in Abhängigkeit zu den ausgewählten Fehlerklassen gefärbten Zeilen.

6. **Hochsprachen–Code**

In Abhängigkeit von der ausgewählten Quellcode-Datei wird der Hochsprachen–Code eingeblendet. Zu jeder Zeile wird soweit vorhanden der zugehörige Assembler-Code eingefügt und gefärbt. Die betroffene Zeile des Hochsprachen-Code wird ebenfalls eingefärbt.

VisualFAIL* ist so entworfen, dass es sich flexibel an die Daten in der Datenbank anpasst. Beispielsweise wird die Auswahl der Fehlerklassen (3) dynamisch aus den Fehlerklassen erzeugt, die in der *result*-Tabelle in der Datenbank zu finden sind.

5.4. Probleme und Lösungen

Bei der Implementierung sind mehrere Probleme aufgetreten, die im Folgenden erläutert werden. Entsprechende Lösungen für diese Probleme, soweit möglich, werden ebenfalls vorgestellt.

5.4.1. Effiziente Umgang mit Daten aus der Datenbank

Durch die Anpassung des FAIL*-Frameworks werden, zusätzlich zu den bereits existierenden Ergebnisdaten der Fehlerinjektionsexperimente, die Debug-Informationen importiert. Um die Daten bspw. zu über variablen zu aggregieren sind komplexe Datenbank-Abfragen notwendig. Ohne jegliche Optimierung benötigt eine solche Abfrage für einen kleinen Benchmark mit etwa 5000 Instruktionen bereits mehrere Minuten, bis die Ergebnisse bereitstehen. Insbesondere, wenn die aggregierten Ergebnisdaten von mehreren Benchmarks analysiert werden sollen bedeutet dies einen erheblichen Zeitaufwand. Für größere Benchmarks ist zu erwarten, dass die notwendige Zeit zur Beantwortung der Anfrage deutlich steigt.

Um diesem Problem entgegen zu treten, wurden die Tabellen sowie die Abfragen optimiert. Durch Optimierungstechniken wie z.B. Umsortierung der Reihenfolge des Primärschlüssels einer Tabelle oder das Einfügen eines Index, führen zu einer deutlich Beschleunigung, die zu einer Beantwortung innerhalb von mehreren Sekunden führt. Für die in dieser Arbeit betrachteten Benchmarks waren diese Optimierungen ausreichend, weswegen keine weiteren Techniken zur Beschleunigung der Abfragen implementiert wurden.

Allerdings sind auch diesen Optimierungen Grenzen gesetzt. Entsprechend ist davon auszugehen, dass diese Optimierungen ab einer gewissen Größe der Benchmarks nicht mehr effizient genug sind.

Als Lösung um mit größeren Benchmarks umzugehen, könnten Teile der Abfragen, die von mehreren Abfragen verwendet werden, vorberechnet werden. Dies würde unter Umständen zu einer Beschleunigung führen, sofern der größte Rechenaufwand in dieser Teilabfrage entsteht. Alternativ könnten alle Abfragen komplett vorberechnet und das Ergebnis gespeichert werden. Der Zeitaufwand wäre der selbe, allerdings bestünde so die Möglichkeit alles vor zu berechnen und anschließend ohne größere Verzögerung alle gewünschten Untersuchungen vorzunehmen.

5.4.2. Komplexität der Debug-Informationen

Wie bereits in dem Kapitel 2.5 angedeutet wurde, sind die Debug-Informationen sehr komplex. Dies führt dazu, dass Informationen sehr unterschiedlich und vor allem verteilt abgelegt werden. Entsprechend entstehen dadurch sehr viele Möglichkeiten, wie bzw. wo nach diesen Informationen mit Hilfe der libdwarf gesucht werden kann. Im Rahmen dieser Arbeit wurden Fälle, die häufig auftreten, abgedeckt. Wie gut die Abdeckung dieser Fälle im Bezug auf die Aggregation von Fehlern auf lokale Variablen ist, wird im Kaptitel Evaluation 6 nähe untersucht.

Die Komplexität der Debug-Informationen bringt ein weiteres Problem mit sich. Die komplexen Daten müssen in Datenstrukturen gehalten werden, um sie zu verarbeiten. Ein gutes Beispiel dafür sind die *location*-Informationen. Die Information wo eine Variable zur Laufzeit im Speicher liegt, kann wie bereits erläutert, partiell definiert sein. Entsprechend aufwendiger wäre die Überprüfung der Position der einzelnen Teile der Variable sowie dessen Speicherung. Im Rahmen dieser Arbeit werden die partiell definierten *location*-Informationen vernachlässigt, da sie eher selten vorkommen.

5.5. Zusammenfassung

In diesem Kapitel wurden zunächst die verwendeten Technologien vorgestellt, die zur Implementierung verwende wurden. Anschließend wurde auf die Anpassungen des FAIL*-Frameworks eingegangen und dessen Struktur anhand eines Klassendiagramms dargestellt. Sowohl die Funktionalität des StackTracer-Plugins, als auch die erweiterte Struktur der Datenbank wurde in einem Unterkapitel genauer erläutert. Zu den neuen Analyse-Verfahren wurde die Oberfläche VisualFAIL* implementiert, dessen Aufbau ebenfalls genauer erläutert wurde. Zum Schluss dieses Kapitels wurden die Probleme, die während der Implementierung aufgetreten sind, erläutert und entsprechende Lösungen vorgestellt.

6. Evaluation

Die Evaluierung der in dieser Arbeit vorgestellten Verfahren ist Inhalt dieses Kapitels. In Kapitel 6.1 wird die Qualität der Daten gemessen, die durch den *Stack-Trace* aufgezeichnet wurden. Die in diesem Kapitel verwendeten Benchmarks mutex1 und thread1 sind identisch mit den Benchmarks aus der Veröffentlichung „Return-Address Protection in C/C++ Code by Dependability Aspects“ von Borchert et al. [35]. Der Vergleich zwischen den Ergebnissen dieser Veröffentlichung und den in dieser Arbeit erzielten Ergebnisse wird in Kapitel 6.2 durchgeführt. Abschließend werden in Kapitel 6.3 Einschränkungen bzgl. der Ergebnisqualität aufgrund von Optimierungen erläutert.

6.1. Qualität der vorgestellten Verfahren

Eine wesentliche Frage, die sich stellt, ist wie gut die hier vorgestellten Verfahren funktionieren. Dazu wird die Qualität über mehrere Qualitätsmetriken untersucht. Diese Metriken werden in Kapitel 6.1.2 vorgestellt und anschließend in Kapitel 6.1.3 auf die Benchmarks angewandt und bewertet.

6.1.1. Effizienz der Rückabbildung auf Äquivalenzklassen

Wie im Kapitel Analyse und Entwurf 4.3 beschrieben, können die Fehler in Form kompletter Äquivalenzklassen durch Färben des Assembler- bzw. Hochsprachen-Code dargestellt werden.

Theoretisch müsste der Fehler jede der Instruktionen der Äquivalenzklasse zugeordnet werden, da das Ergebnis entsprechend dasselbe wäre. Dies würde jedoch dazu führen, dass der Fehler auf viele Zeilen des Hochsprachen-Codes abgebildet werden würde. Dies ist prinzipiell kein Problem, solange die Äquivalenzklasse lediglich eine Funktion betrifft. In diesem Fall wäre klar zu erkennen, dass diese Funktion fehleranfällig ist und durch entsprechende Fehlertoleranzmaßnahmen geschützt werden sollte.

Häufig ist es jedoch so, dass sich die Äquivalenzklasse dadurch, dass sie sich lediglich auf Speicherzugriffe bezieht, über mehrere Funktionen erstreckt. Dies tritt z.B. durch folgenden Programmablauf auf:

1. Funktion **x** speichert einen Wert in einer Variable **var**
2. Funktion **x** ruft Funktion **y** auf
3. Funktion **y** kehrt zurück, ohne auf Variable **var** zugegriffen zu haben

4. Funktion **x** greift lesend auf die Variable **var** zu

In diesem Beispiel würde die Funktion **y** zu der Äquivalenzklasse gehören, die sich durch die Speicherzugriffe auf die Variable **var** ergibt. Würde man in diesem Beispiel jeder Instruktion aus der Äquivalenzklasse den Fehler aus dem Experiment zurechnen, würde der Eindruck entstehen, dass Funktion **y** fehleranfällig ist. Demzufolge würde der Eindruck entstehen, dass Funktion **y** gehärtet werden müsste. Der Entwickler zieht letztlich falsche Schlüsse aus der Färbung der Äquivalenzklassen.

Das Problem liegt darin, dass eine zeitliche Dimension auf eine räumliche Dimension abgebildet wird und dies in diesem Fall nicht sinnvoll ist. Die dynamischen Instruktionen stellen den zeitlichen Ablauf des Programms dar. Die statischen Instruktionen beschreiben dagegen eine räumliche Dimension.

6.1.2. Qualitätsmetrik für die Aggregation

Metrik 1: Abdeckung der Debug-Informationen auf Instruktionen

Durch das StackTracer-Plugin wird für jede Instruktion nach Debug-Informationen gesucht, die die passende Funktion abbilden. Die Suche nach diesen Funktionen ist verhältnismäßig trivial, denn es gibt nur wenige DIE-Typen, die eine solche Funktion in den Debug-Informationen abbilden. Der DwarfReader durchsucht alle DIEs nach diesen Typen. Entsprechend kann davon ausgegangen werden, dass die Funktion zu einer Instruktion gefunden wird, sofern sie existiert. Mit dieser Annahme können die Instruktionen, zu denen Debug-Informationen gefunden wurden, von denen ohne Debug-Informationen getrennt werden. Dies führt dazu, dass Aussagen bzgl. der Qualität der Debug-Informationen möglich werden, denn wenn zu einer Instruktion keine Debug-Informationen gefunden wurden, existieren sie nicht. Entsprechend kann die Abdeckung der Debug-Informationen auf Instruktionen wie folgt berechnet werden:

$$Abdeckung \ = \ (Instruktionen_{\ ohne\ Dbg-Info} / Instruktionen_{gesamt})$$

Metrik 2: Abdeckung der Variablen des Stack-Trace auf die Äquivalenzklassen

Die Äquivalenzklassen der Ergebnisdaten, die aus dem Pruning entstehen, sind für eine bestimmte Speicheradresse über eine bestimmte Länge in dynamischen Instruktionen definiert. Die Aufzeichnungen bzgl. der Variablen in der Tabelle dbg_stacktrace sind ebenfalls für eine bestimmte Speicheradresse über eine bestimmte Länge in dynamischen Instruktionen definiert. Wenn diese Daten im Stack-Segment des Fehlerraums übereinander gelegt werden, bildet die Schnittfläche eine Anzahl von Fehlern ab. Diese Fläche spiegelt die Abdeckung der Variablen des Stack-Trace auf die Äquivalenzklassen und somit auf die konkreten Fehler wieder.

6.1.3. Qualitätsbewertung der Aggregation

Die Evaluierung bzgl. der Abdeckung der Debug-Informationen über die Instruktionen liefert gute Ergebnisse (siehe Abschnitt 6.1). Für den Benchmark mutex1 konnten für

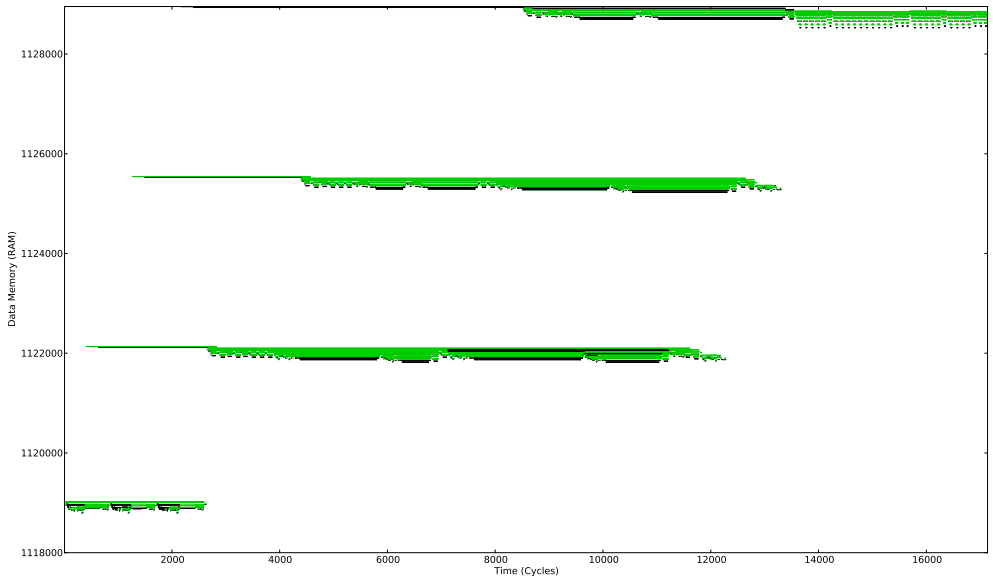

Abbildung 6.1.: Überdeckung für den Benchmark mutex1 ohne Optimierungen. Überdeckte Äquivalenzklassen sind grün gekennzeichnet. Nicht überdeckte Bereiche sind schwarz gefärbt.

Benchmark	Instruktionen gesamt	Instruktionen mit Debug-Informationen	%
mutex1 (O0)	17139	16286	95.02
thread1 (O0)	8969	8457	94.29

Tabelle 6.1.: Auswertung von Metrik 1 für die Benchmarks mutex1, thread1 ohne Optimierungen

95.02% aller Instruktionen in diesem Benchmark Debug-Informationen gefunden werden. Für thread1 ist die Abdeckung nahezu identisch. Dies bedeutet, dass für nicht optimierte Benchmarks davon ausgegangen werden kann, dass es zu jeder Instruktion des Hochsprachen-Code eine entsprechende Funktion aus den Debug-Informationen zugeordnet werden kann. Der Großteil der fehlenden 5% ergibt sich aus Funktionen, die in Assembler implementiert wurden und daher keinen Debug-Informationen zugeordnet sind. Dieses Ergebnis bringt Aufschluss darüber, dass Debug-Informationen vorhanden sind. Allerdings kann hieraus keine Aussagen darüber getroffen werden, wie gut die Qualität dieser Debug-Informationen ist.

Tabelle 6.2 zeigt die Überdeckung nach Metrik 2 für die beiden unoptimierten Benchmarks. Für die Evaluierung wurden die Variablen auf dem Stack aufgezeichnet, soweit

Benchmark	Fehler insgesamt	Fehler abgedeckt	%
mutex1 (O0)	5884642	4508518	76.84
thread1 (O0)	5759306651	4372647271	74.00

Tabelle 6.2.: Auswertung von Metrik 2 für die Benchmarks mutex1, thread1 ohne Optimierungen. Betrachtet wurde für diese Evaluierung lediglich das Stack-Segment.

die dafür notwendigen Debug-Informationen abgerufen werden konnten. Es sei an dieser Stelle nochmals angemerkt, dass in der Implementierung für diese Arbeit aufgrund der Vielfalt der Debug-Informationen nicht alle Möglichkeiten, nach Debug-Informationen zu suchen, ausgeschöpft wurden. Zusätzlich zu den lokalen Variablen wurde der Base-Pointer sowie der Instruction-Pointer für jedes Stackframe als künstliche Variablen mit aufgezeichnet. Insgesamt ergibt sich eine Abdeckung von 76.61% bzw. 75.92% aller Fehler im Stack-Segment durch die aufgezeichneten Variablen. Entsprechend können somit 76.84% bzw. 74.00% der Fehler auf die lokalen bzw. künstlichen Variablen aggregiert werden. Es kann davon ausgegangen werden, dass sich durch die Implementierung von mehr Möglichkeiten zur Suche nach lokalen Variablen in den Debug-Informationen, das Ergebnis weiter verbessern lässt.

6.2. Vergleich mit Return-Address Protection in C/C++ Code by Dependability Aspects

Wie im Kapitel Related Work bereits erwähnt wurde, wird in einer Veröffentlichung von Borchert et al. [35] der Stack untersucht. Bei der Untersuchung wurden lediglich dem gesamten Stack-Segment Fehler zugeordnet und durch genauere manuelle Betrachtung festgestellt, dass die Rücksprungadresse und Base-Pointer häufig von Fehlern betroffen sind. Im weiteren Verlauf der Untersuchung wurde klar, dass es nicht effizient genug ist die Rücksprungadresse für *jede* Funktion zu härten, da es einige Funktionen in dem Benchmark mutex1 gibt, die häufig aufgerufen werden, aber dafür nur kurz laufen. Eine solche Funktion mit Fehlertoleranzmechanismen zu schützen würde die Laufzeit des gesamten Benchmarks erheblich verlängern. Deshalb wurde die Laufzeit der Funktionen evaluiert, um herauszufinden, welche Funktionen lang genug laufen, damit die Anwendung einer Fehlertoleranzmaßnahme sinnvoll ist. Neben einer Erkennungsmaßnahme wurde eine Korrekturmaßnahme für die Rücksprungadresse evaluiert. Mit der Korrekturmaßnahme konnten für den Benchmark mutex1 die Fehler, die den Stack betreffen, um 24% gesenkt werden.

Funktion	Fehler insgesamt	%
BP,IP	3878169	86.02
unlock_inner	82747	1.83
thread_entry	13220	0.29
entry0	8787	0.19
wait_inner	8565	0.19

Tabelle 6.3.: Top5 der auf Funktionen aggregierten Ergebnisdaten für den Benchmark mutex1(O0).

Variable	Fehler insgesamt	%
IP	2020720	44.82
BP	1857449	41.20
current	74458	1.65
thread	13220	0.29
data	8787	0.19

Tabelle 6.4.: Top5 der auf Variablen aggregierten Ergebnisdaten für den Benchmark mutex1(O0).

6.2.1. Ergebnis

Mit den in dieser Arbeit entwickelten Verfahren können die Ergebnisdaten wesentlich genauer auf Variablen und Funktionen abgebildet werden, als es in der Veröffentlichung möglich war. Die auf Funktionen aggregierten Ergebnisdaten für den Benchmark mutex1 sind in Tabelle 6.3 zu sehen. Für die künstlich aufgezeichneten Variable gibt es auch eine künstliche Funktion, so dass die Fehler, die den Base-Pointer, als auch den Instruction-Pointer betreffen, aggregiert werden können. Das Ergebnis ist recht eindeutig. Die Fehler, die den abgelegten Base-Pointer und den Instruction-Pointer betreffen, bilden 86.02% der gesamten Fehler. Daher sollte der Fokus bei der Härtung des Benchmarks auf der Steigerung der Fehlertoleranz für den abgelegten Base-Pointer bzw. Instruction-Pointer liegen.

Für die aggregierten Ergebnisdaten auf der Granularität von Variablen, wie in Tabelle 6.4 abgebildet ergibt sich ein ähnliches Ergebnis. Sowohl der Instruction-Pointer (IP), als auch der Base-Pointer (BP) sind in etwa gleich fehleranfällig.

Somit ist klar, dass sowohl der Base-Pointer, als auch der Instruction-Pointer geschützt werden müssen, um die Fehlertoleranz des Benchmarks zu steigern. Dennoch stellt sich

Benchmark	O0	O1	O2	O3
mutex1	76.84	7.02	4.53	5.91
thread1	74.00	5.26	0.001	0.0008

Tabelle 6.5.: Auswirkungen von Optimierungen auf die Abdeckung der Fehler im Stack-Segment in %.

wieder die Frage, welche Funktionen lang genug laufen, damit sich die Anwendung eines Fehlertoleranzmechanismus lohnt. Diese Informationen stehen ebenfalls in der Tabelle *dbg_stacktrace*, da für jeden Funktionsaufruf die framebase in dieser Tabelle abgelegt wird. Durch die dynamische Start- und Endadresse ist auch die Laufzeit der Funktionen bekannt. Somit ist das Ergebnis der neuen Analysen über den Stack-Trace identisch mit denen auf der Veröffentlichung. Der Vorteil der neuen Analyse liegt darin, dass die Aussagekraft der aggregierten Ergebnisdaten stärker ist. Es kann für jede Variable bzw. Funktion festgestellt werden, wie fehleranfällig sie ist. Darüber hinaus wurde die Messung der Zeit sowei die Aufrufhäufigkeit, der einzelnen Funktionen, mit großem Aufwand gemessen. Bei der hier evaluierten Analyse werden diese Informationen automatisch mitgeliefert.

6.3. Einschränkungen durch Optimierungen

Das hier vorgestellte Verfahren unterliegt Einschränkungen, die aus dem *Code-Location-Problem*, das im Kapitel Related Work beschrieben wurde, resultieren. Durch Optimierungen werden die Debug-Informationen ungenauer bzw. unvollständiger. Dies hat direkte Auswirkungen auf die Effizienz der neuen Analyse-Verfahren. Tabelle 6.5 zeigt wie sie die Fehler-Abdeckung nach Metrik 2 mit unterschiedlichen Optimierungen verändert. Die Abdeckung nimmt bereits bei dem Optimierungslevel O1 erheblich ab und liegt lediglich bei 7% bzw. 5%. Somit sind die neuen Analyse-Verfahren nicht mehr anwendbar, sobald Optimierungen während des Compile-Vorgangs angewendet werden.

Der Grund dafür liegt darin, dass sich ein Großteil der Fehler auf denabgelegten Base-Pointer sowie Instrction-Pointer bezieht. Diese Werte werden in der Implementierung des Stacktracer-Plugins über den Inhalt des EBP-Register bezogen. Sobald eine der Optimierungsstufen eingeschaltet werden, wird der EBP nicht mehr gesetzt. Entsprechend ist es dann nicht mehr möglich die Lage dieser Daten aufzuzeichnen.

Ein weiterer Grund dafür wird ersichtlich, wenn man betrachtet, zu wie vielen Instruktionen noch Debug-Informationen gefunden werden können, wenn Optimierungen angewandt werden (siehe Tabelle 6.6). Mit den Optimierungen stehen für immer weniger Instruktionen Debug-Informationen zur Verfügung. Für den Optimierungslevel O3 sind es lediglich 41.09% bzw. 53.27%. Dies wirkt sich deutlich auf die Abdeckung nach

Benchmark	O0	O1	O2	O3
mutex1	95.02	80.91	76.49	41.09
thread1	94.29	79.58	69.96	53.27

Tabelle 6.6.: Auswirkungen von Optimierungen auf die Abdeckung der Debug-Informationen auf Instruktionen in %.

Metrik 2 aus, da somit für einen Großteil des Benchmarks keine Debug-Informationen zu Verfügung stehen.

Noch ein weiterer Grund liegt darin, dass die Debug-Informationen durch die Optimierungen teilweise fehlerhaft werden, wie im Kapitel Related Work beschrieben.

6.4. Zusammenfassung

In diesem Kapitel wurde die Qualität der in dieser Arbeit entwickelten Analyse-Verfahren evaluiert. Im Bezug auf die Rückabbildung wurde ersichtlich, dass die Abbildung von Äquivalenzklassen auf Assembler- bzw. Hochsprachen-Code nicht sinnvoll ist. Für die Aggregation wurde festgestellt, dass die Abdeckung durch die verwendete Implementierung für nicht optimierte Benchmarks bei etwa 76% liegt. Unter Verwendung der neuen Analysen wurde die Evaluierung zweier Benchmarks aus einer Veröffentlichung wiederholt. Das Ergebnis ist identisch. Dennoch bieten die neuen Analyse-Verfahren umfangreiche Informationen über die Daten, die im Stack-Segment liegen. Zum Abschluss wurde evaluiert, welche Auswirkungen Optimierungen auf die neuen Analyse-Verfahren haben. Es wurde festgestellt, dass die neuen Analysen bei der Verwendung von Optimierungen nicht mehr anwendbar sind, da aufgrund des *Code-Location-Problem* die Anzahl und Qualität der Debug-Informationen erheblich abnimmt.

7. Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es neue Analyse-Verfahren für Ergebnisse aus Fehlerinjektionsexperimenten zu entwickeln und Ansätze für eine automatisierte Analyse bzw. Härtung von Software zu erarbeiten. Zu diesem Zweck wurde das Fehlerinjektions-Framework FAIL* erweitert, um durch zusätzliche Informationen aussagekräftigere Analysen zu ermöglichen. Im Folgenden wird in Abschnitt 7.1 auf die Ergebnisse, die aus dieser Arbeit resultieren, eingegangen. Anschließend werden in Abschnitt 7.2 Möglichkeiten betrachtet, um die in dieser Arbeit vorgestellten Verfahren zu verbessern, bzw. um den Kreislauf zur Härtung von Software zu komplettieren.

7.1. Zusammenfassung

Die Anforderungsanalyse hat ergeben, dass es notwendig ist, Aussagen über die Fehleranfälligkeit von Software-Komponenten auf Ebene des Hochsprachen-Code treffen zu können. Durch das bisherige FAIL*-Framework konnte dies lediglich für globale Objekte durch eine manuelle Zuordnung der Speicheradresse erfolgen. Wie bisherige Veröffentlichungen gezeigt haben ist insbesondere das Stack-Segment im Speicher von Fehlern betroffen. Daher wurde in dieser Arbeit ein Verfahren entwickelt, um die Aggregation von Ergebnisdaten auf Software-Komponenten zu ermöglichen, die im Stack-Segment liegen.

Dazu wurde das FAIL*-Framework um die Möglichkeit erweitert einen Stack-Trace aufzuzeichnen. Dazu war es notwendig auf die Debug-Informationen, die durch den Compiler erzeugt werden können, zuzugreifen und diese entsprechend zu verarbeiten. Dabei sind im Wesentlichen zwei Probleme aufgetreten. Zum einen sind die Debug-Informationen sehr komplex, sodass es einen sehr hohen Implementierungs-Aufwand erfordert, um die Debug-Informationen komplett nach den gewünschten Informationen durchsuchen zu können. Zum anderen können die Debug-Informationen aufgrund des *Code-Location-Problems* fehlerhaft bzw. lückenhaft sein.

Die Implementierung, die im Rahmen dieser Arbeit entstanden ist, besteht aus zwei Teilen. Einerseits wurde das Stacktracer-Plugin implementiert, dass zum Aufzeichnen der Lage der lokalen Variablen im Stack-Segment dient. Aufgrund des Umgangs mit wenigen fehlerhaften Debug-Informationen ist diese Plugin derzeit x86-spezifisch. Dennoch dürfte sich das Plugin mit wenig Aufwand für andere Plattformen portieren lassen. Andererseits wurde mit VisualFAIL* eine Web-Oberfläche zur Visualisierung von Ergebnisdaten auf Assembler- sowie Hochsprachen-Code implementiert. Die Architektur nach der VisualFAIL* entworfen wurde ist sehr flexibel und somit auch für zukünftige Implementierungen von Visualisierungen geeignet.

Das Problem der fehlerhaften bzw. lückenhaften Debug-Informationen wurde insbesondere für Benchmarks deutlich, die mit Optimierungen übersetzt wurden. Dennoch hat die Evaluierung gezeigt, dass der Ansatz der Aggregation über Software-Komponenten aus dem Hochsprachen-Code zumindest für nicht optimierte Benchmarks aussagekräftige Ergebnisse liefert. Da nicht alle Möglichkeiten die Debug-Informationen zu durchsuchen implementiert werden konnten, kann angenommen werden, dass sich die Variablen- bzw. Stackabdeckung aus der Evaluierung weiter steigern lassen.

Diese Arbeit zeigt, dass der Ansatz Ergebnisdaten von Fehlerinjektions-Experimenten auf Software-Komponenten zu aggregieren, funktioniert. Die Ergebnisse zeigen, dass mit diesem Verfahren eine verbesserte und genauere Analyse möglich wird.

7.2. Ausblick

Durch die Anforderungsanalyse wurde festgestellt, welche Schritte erfolgen müssen, um Software, möglicherweise auch automatisch, zu härten. Nach der Analyse und Priorisierung erfolgt eine Platzierung der Fehlertoleranzmaßnahmen. Im Rahmen dieser Arbeit wurden diesbezüglich Vorüberlegungen angestellt, um einen derartigen Mechanismus umzusetzen. Die Umsetzung dieses Schrittes des Kreislaufs zur Härtung von Software (siehe Kapitel 1.1) ist die nächste notwendige Implementierung, um eine vollständige Härtung auf Basis von Analysen zu ermöglichen. Je nachdem, ob nach der Anwendung der Fehlertoleranzmaßnahmen ein weiterer Kampagnen-Durchlauf erfolgen muss, ist es zusätzlich notwendig die Effektivität der angewandten Maßnahmen zu bewerten.

Für die Aggregation von Ergebnisdaten ist es von größter Bedeutung, dass die dafür genutzten Debug-Informationen möglichst erschöpfend genutzt werden. Entsprechend ist eine Verbesserung der in dieser Arbeit entstandenen Implementierung ein lohnender Aufwand, der zu einer Verbesserung der Funktionalität des Stacktracer-Plugins führt.

Mit den in dieser Arbeit entwickelten Verfahren können Aussagen über die Fehleranfälligkeit von Software-Komponenten im DATA-, BSS- und Stack-Segment getätigt werden. Der letzte fehlende Speicherbereich ist das Heap-Segment. Die Entwicklung von Verfahren, um Aussagen über die Fehleranfälligkeit von Software-Komponenten treffen zu können, kann für zukünftige Entwicklungen relevant sein.

Literaturverzeichnis

[1] Borkar, Shekhar: Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. In: *IEEE Micro* 25 (2005), November, Nr. 6, 10–16. http://dx.doi.org/10.1109/MM.2005.110. – DOI 10.1109/MM.2005.110. – ISSN 0272–1732

[2] Avizienis, A. ; Laprie, J.-C. ; Randell, B. ; Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. In: *Dependable and Secure Computing, IEEE Transactions on* 1 (2004), Jan, Nr. 1, S. 11–33. http://dx.doi.org/10.1109/TDSC.2004.2. – DOI 10.1109/TDSC.2004.2. – ISSN 1545–5971

[3] *The Home of AspectC++*. http://www.aspectc.org/. Version: 02 2014

[4] Schirmeier, H. ; Hoffmann, M. ; Kapitza, R. ; Lohmann, D. ; Spinczyk, O.: Fail*: Towards a versatile fault-injection experiment framework. In: *ARCS Workshops (ARCS), 2012*, 2012, S. 1–5

[5] Borchert, Christoph ; Schirmeier, Horst ; Spinczyk, Olaf: Generative Software-based Memory Error Detection and Correction for Operating System Data Structures. In: *Proceedings of the 43nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, IEEE Computer Society Press, Juni 2013. – ISBN 978–1–4673–6471–3

[6] Sridharan, Vilas ; Liberty, Dean: A Study of DRAM Failures in the Field. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2012 (SC '12). – ISBN 978–1–4673–0804–5, 76:1–76:11

[7] Koren, Israel ; Krishna, C. M.: *Fault-Tolerant Systems*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2007. – ISBN 0120885255, 9780080492681

[8] Mastipuram, Wee E. C. R.: Soft errors impact on system reliability. In: *design feature* (2004), September

[9] Dupont, E. ; Nicolaidis, M. ; Rohr, P.: Embedded robustness IPs for transient-error-free ICs. In: *Design Test of Computers, IEEE* 19 (2002), May, Nr. 3, S. 54–68. http://dx.doi.org/10.1109/MDT.2002.1003798. – DOI 10.1109/MDT.2002.1003798. – ISSN 0740–7475

[10] Benso, A. ; Rebaudengo, M. ; Impagliazzo, L. ; Marmo, P.: Fault-list collapsing for fault-injection experiments. In: *Reliability and Maintainability Symposium, 1998. Proceedings., Annual*, 1998. – ISSN 0149–144X, S. 383–388

[11] Hsueh, Mei-Chen ; Tsai, T.K. ; Iyer, R.K.: Fault injection techniques and tools. In: *Computer* 30 (1997), Apr, Nr. 4, S. 75–82. http://dx.doi.org/10.1109/2.585157. – DOI 10.1109/2.585157. – ISSN 0018–9162

[12] Benso, A. ; Di Carlo, S. ; Di Natale, G. ; Prinetto, P. ; Taghaferri, L.: Data criticality estimation in software applications. In: *Test Conference, 2003. Proceedings. ITC 2003. International* Bd. 1, 2003. – ISSN 1089–3539, S. 802–810

[13] Aidemark, Joakim ; Vinter, Jonny ; Folkesson, Peter ; Karlsson, Johan: *GOOFI: Generic Object-Oriented Fault Injection Tool*. 2001

[14] Skarin, D. ; Barbosa, R. ; Karlsson, J.: GOOFI-2: A tool for experimental dependability assessment. In: *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 2010, S. 557–562

[15] Leveugle, R. ; Calvez, A. ; Maistri, P. ; Vanhauwaert, P.: Statistical fault injection: Quantified error and confidence. In: *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 2009. – ISSN 1530–1591, S. 502–506

[16] Rademacher, Lars: *FailPanda: Fehlerinjektionsexperimente auf einer eingebetteten ARM-Plattform*, Technische Universität Dortmund, Diplomarbeit, 2013

[17] Spinczyk, Olaf ; Lohmann, Daniel ; Urban, Matthias: Advances in AOP with AspectC++. In: *Proceedings of the 2005 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the Fourth SoMeTW05*. Amsterdam, The Netherlands, The Netherlands : IOS Press, 2005. – ISBN 1–58603–556–8, 33–53

[18] *The official DanceOS project page*. http://www.danceos.org. Version: 02 2014

[19] Mihocka, Darek: *Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure Abstract*

[20] Binkert, Nathan ; Beckmann, Bradford ; Black, Gabriel ; Reinhardt, Steven K. ; Saidi, Ali ; Basu, Arkaprava ; Hestness, Joel ; Hower, Derek R. ; Krishna, Tushar ; Sardashti, Somayeh ; Sen, Rathijit ; Sewell, Korey ; Shoaib, Muhammad ; Vaish, Nilay ; Hill, Mark D. ; Wood, David A.: The Gem5 Simulator. In: *SIGARCH Comput. Archit. News* 39 (2011), aug, Nr. 2, 1–7. http://dx.doi.org/10.1145/2024716.2024718. – DOI 10.1145/2024716.2024718. – ISSN 0163–5964

[21] Mauerer, Wolfgang: *Linux Kernel Architektur, Anhang E: Das ELF-Binärformat*. 2003

[22] Srinivasaraghavan, Rajalakshmi: Exploring the DWARF debug format information. (2013), Aug. https://www.ibm.com/developerworks/aix/library/au-dwarf-debug-format/au-dwarf-debug-format-pdf.pdf

[23] Eager, Michael J. ; Consulting, Eager: Introduction to the DWARF Debugging Format. In: *Group* (2007)

[24] DWARF Standards Committee (Hrsg.): *DWARF Debugging Information Format Version 4*. DWARF Standards Committee, June 2010. http://www.dwarfstd.org/doc/DWARF4.pdf

[25] Bendersky, Eli: *How debuggers work: Part 3 Debugging information*. http://eli.thegreenplace.net/2011/02/07/how-debuggers-work-part-3-debugging-information/. Version: February 2011

[26] Tice, Caroline ; Graham, Susan L. ; Tice, Caroline ; Susan, Prof ; Susan, Prof ; Graham, L. ; Graham, L.: *Key Instructions: Solving the Code Location Problem for Optimized Code*. 2000

[27] Copperman, Max: *Debugging Optimized Code Without Being Misled*. 1993

[28] Venturini, Hugo ; Riss, Frédéric ; Fernandez, Jean-Claude ; Santana, Miguel: A Fully-non-transparent Approach to the Code Location Problem. In: *Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems*. New York, NY, USA : ACM, 2008 (SCOPES '08), 61–68

[29] Wu, Le chun ; Mirani, Rajiv ; Patil, Harish ; Olsen, Bruce ; W. Hwu, Wen mei: *A New Framework for Debugging Globally Optimized Code*. 1999

[30] Hennessy, John: Symbolic Debugging of Optimized Code. In: *ACM Trans. Program. Lang. Syst.* 4 (1982), Juli, Nr. 3, 323–344. http://dx.doi.org/10.1145/357172.357173. – DOI 10.1145/357172.357173. – ISSN 0164–0925

[31] Coutant, D. S. ; Meloy, S. ; Ruscetta, M.: DOC: A Practical Approach to Source-level Debugging of Globally Optimized Code. In: *SIGPLAN Not.* 23 (1988), Juni, Nr. 7, 125–134. http://dx.doi.org/10.1145/960116.54003. – DOI 10.1145/960116.54003. – ISSN 0362–1340

[32] Hölzle, Urs ; Chambers, Craig ; Ungar, David: Debugging Optimized Code with Dynamic Deoptimization. In: *SIGPLAN Not.* 27 (1992), Juli, Nr. 7, 32–43. http://dx.doi.org/10.1145/143103.143114. – DOI 10.1145/143103.143114. – ISSN 0362–1340

[33] Gough, John ; Ledermann, Jeff ; Elms, Kim: Interpretive Debugging of Optimised Code. In: *Proceedings of ACSC-17, Christchurch*, 1994

[34] Borchert, Christoph ; Schirmeier, Horst ; Spinczyk, Olaf: Protecting the Dynamic Dispatch in C++ by Dependability Aspects. In: *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, German Society of Informatics, September 2012 (Lecture Notes in Informatics), 521–535

[35] Borchert, Christoph ; Schirmeier, Horst ; Spinczyk, Olaf: Return-Address Protection in C/C++ Code by Dependability Aspects. In: *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, German Society of Informatics, September 2013 (Lecture Notes in Informatics)

[36] Pattabiraman, K. ; Kalbarczyk, Z. ; Iyer, R.K.: Application-based metrics for strategic placement of detectors. In: *Dependable Computing, 2005. Proceedings. 11th Pacific Rim International Symposium on*, 2005, S. 8 pp.–

[37] Sridharan, V. ; Kaeli, D.R.: Eliminating microarchitectural dependency from Architectural Vulnerability. In: *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, 2009. – ISSN 1530–0897, S. 117–128

[38] Hiller, M. ; Jhumka, A. ; Suri, N.: EPIC: profiling the propagation and effect of data errors in software. In: *Computers, IEEE Transactions on* 53 (2004), May, Nr. 5, S. 512–530. http://dx.doi.org/10.1109/TC.2004.1275294. – DOI 10.1109/TC.2004.1275294. – ISSN 0018–9340

[39] Mukherjee, Shubhendu S. ; Weaver, Christopher ; Emer, Joel ; Reinhardt, Steven K. ; Austin, Todd: A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2003 (MICRO 36). – ISBN 0–7695–2043–X, 29–

[40] *libdwarf - elftoolchain*. http://sourceforge.net/apps/trac/elftoolchain/wiki/libdwarf. Version: 04 2014

[41] *Bootstrap Framework*. http://getbootstrap.com/. Version: 04 2014

[42] *Jquery*. http://jquery.com/. Version: 04 2014

Abbildungsverzeichnis

A. Anhang

A.1. Komplette Sektion .debug_info

```
$ objdump --dwarf=info beispiel

beispiel:     file format elf32-i386

Contents of the .debug_info section:

  Compilation Unit @ offset 0x0:
   Length:        0x18b (32-bit)
   Version:       2
   Abbrev Offset: 0
   Pointer Size:  4
 <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <c>   DW_AT_producer    : (indirect string, offset: 0x1d): GNU C++ 4.7.2
    <10>   DW_AT_language    : 4 (C++)
    <11>   DW_AT_name        : (indirect string, offset: 0x9b): beispiel.cc
    <15>   DW_AT_comp_dir    : (indirect string, offset: 0x62): /fs/students/hell
    <19>   DW_AT_ranges      : 0x0
    <1d>   DW_AT_low_pc      : 0x0
    <21>   DW_AT_entry_pc    : 0x0
    <25>   DW_AT_stmt_list   : 0x0
 <1><29>: Abbrev Number: 2 (DW_TAG_class_type)
    <2a>   DW_AT_name        : (indirect string, offset: 0x56): Person
    <2e>   DW_AT_byte_size   : 4
    <2f>   DW_AT_decl_file   : 1
    <30>   DW_AT_decl_line   : 9
    <31>   DW_AT_sibling     : <0x82>
 <2><35>: Abbrev Number: 3 (DW_TAG_member)
    <36>   DW_AT_name        : (indirect string, offset: 0x93): m_alter
    <3a>   DW_AT_decl_file   : 1
    <3b>   DW_AT_decl_line   : 12
    <3c>   DW_AT_type        : <0x82>
    <40>   DW_AT_data_member_location: 2 byte block: 23 0  (DW_OP_plus_uconst: 0)
    <43>   DW_AT_accessibility: 3 (private)
 <2><44>: Abbrev Number: 4 (DW_TAG_subprogram)
```

```
   <45>   DW_AT_external    : 1
   <46>   DW_AT_name        : (indirect string, offset: 0x14): setAlter
   <4a>   DW_AT_decl_file   : 1
   <4b>   DW_AT_decl_line   : 15
   <4c>   DW_AT_MIPS_linkage_name: (indirect string, offset: 0x39): _ZN6Person8setAlterE
   <50>   DW_AT_declaration : 1
   <51>   DW_AT_object_pointer: <0x59>
   <55>   DW_AT_sibling     : <0x65>
 <3><59>: Abbrev Number: 5 (DW_TAG_formal_parameter)
   <5a>   DW_AT_type        : <0x89>
   <5e>   DW_AT_artificial  : 1
 <3><5f>: Abbrev Number: 6 (DW_TAG_formal_parameter)
   <60>   DW_AT_type        : <0x82>
 <2><65>: Abbrev Number: 7 (DW_TAG_subprogram)
   <66>   DW_AT_external    : 1
   <67>   DW_AT_name        : (indirect string, offset: 0x30): getAlter
   <6b>   DW_AT_decl_file   : 1
   <6c>   DW_AT_decl_line   : 16
   <6d>   DW_AT_MIPS_linkage_name: (indirect string, offset: 0xc2): _ZN6Person8getAlterE
   <71>   DW_AT_type        : <0x82>
   <75>   DW_AT_declaration : 1
   <76>   DW_AT_object_pointer: <0x7a>
 <3><7a>: Abbrev Number: 5 (DW_TAG_formal_parameter)
   <7b>   DW_AT_type        : <0x89>
   <7f>   DW_AT_artificial  : 1
 <1><82>: Abbrev Number: 8 (DW_TAG_base_type)
   <83>   DW_AT_byte_size   : 4
   <84>   DW_AT_encoding    : 5 (signed)
   <85>   DW_AT_name        : int
 <1><89>: Abbrev Number: 9 (DW_TAG_pointer_type)
   <8a>   DW_AT_byte_size   : 4
   <8b>   DW_AT_type        : <0x29>
 <1><8f>: Abbrev Number: 10 (DW_TAG_subprogram)
   <90>   DW_AT_external    : 1
   <91>   DW_AT_name        : (indirect string, offset: 0xb7): checkAlter
   <95>   DW_AT_decl_file   : 1
   <96>   DW_AT_decl_line   : 1
   <97>   DW_AT_MIPS_linkage_name: (indirect string, offset: 0xa7): _Z10checkAlteri
   <9b>   DW_AT_type        : <0xbf>
   <9f>   DW_AT_low_pc      : 0x80484b7
   <a3>   DW_AT_high_pc     : 0x80484d7
   <a7>   DW_AT_frame_base  : 0x0 (location list)
   <ab>   DW_AT_GNU_all_call_sites: 1
   <ac>   DW_AT_sibling     : <0xbf>
```

```
<2><b0>: Abbrev Number: 11 (DW_TAG_formal_parameter)
   <b1>   DW_AT_name        : (indirect string, offset: 0x95): alter
   <b5>   DW_AT_decl_file   : 1
   <b6>   DW_AT_decl_line   : 1
   <b7>   DW_AT_type        : <0x82>
   <bb>   DW_AT_location    : 2 byte block: 91 0  (DW_OP_fbreg: 0)
<1><bf>: Abbrev Number: 12 (DW_TAG_base_type)
   <c0>   DW_AT_byte_size   : 1
   <c1>   DW_AT_encoding    : 2 (boolean)
   <c2>   DW_AT_name        : (indirect string, offset: 0x4f): bool
<1><c6>: Abbrev Number: 13 (DW_TAG_subprogram)
   <c7>   DW_AT_specification: <0x44>
   <cb>   DW_AT_low_pc      : 0x80484d8
   <cf>   DW_AT_high_pc     : 0x80484f7
   <d3>   DW_AT_frame_base  : 0x38 (location list)
   <d7>   DW_AT_object_pointer: <0xe0>
   <db>   DW_AT_GNU_all_tail_call_sites: 1
   <dc>   DW_AT_sibling     : <0xfc>
<2><e0>: Abbrev Number: 14 (DW_TAG_formal_parameter)
   <e1>   DW_AT_name        : (indirect string, offset: 0x5d): this
   <e5>   DW_AT_type        : <0xfc>
   <e9>   DW_AT_artificial  : 1
   <ea>   DW_AT_location    : 2 byte block: 91 0  (DW_OP_fbreg: 0)
<2><ed>: Abbrev Number: 11 (DW_TAG_formal_parameter)
   <ee>   DW_AT_name        : (indirect string, offset: 0x5): neu_alter
   <f2>   DW_AT_decl_file   : 1
   <f3>   DW_AT_decl_line   : 15
   <f4>   DW_AT_type        : <0x82>
   <f8>   DW_AT_location    : 2 byte block: 91 4  (DW_OP_fbreg: 4)
<1><fc>: Abbrev Number: 15 (DW_TAG_const_type)
   <fd>   DW_AT_type        : <0x89>
<1><101>: Abbrev Number: 16 (DW_TAG_subprogram)
   <102>   DW_AT_specification: <0x65>
   <106>   DW_AT_low_pc      : 0x80484f8
   <10a>   DW_AT_high_pc     : 0x8048502
   <10e>   DW_AT_frame_base  : 0x70 (location list)
   <112>   DW_AT_object_pointer: <0x11b>
   <116>   DW_AT_GNU_all_call_sites: 1
   <117>   DW_AT_sibling     : <0x129>
<2><11b>: Abbrev Number: 14 (DW_TAG_formal_parameter)
   <11c>   DW_AT_name        : (indirect string, offset: 0x5d): this
   <120>   DW_AT_type        : <0xfc>
   <124>   DW_AT_artificial  : 1
   <125>   DW_AT_location    : 2 byte block: 91 0  (DW_OP_fbreg: 0)
```

```
<1><129>: Abbrev Number: 17 (DW_TAG_subprogram)
   <12a>   DW_AT_external    : 1
   <12b>   DW_AT_name        : (indirect string, offset: 0x8e): main
   <12f>   DW_AT_decl_file   : 1
   <130>   DW_AT_decl_line   : 19
   <131>   DW_AT_type        : <0x82>
   <135>   DW_AT_low_pc      : 0x804848c
   <139>   DW_AT_high_pc     : 0x80484b7
   <13d>   DW_AT_frame_base  : 0xa8 (location list)
   <141>   DW_AT_GNU_all_tail_call_sites: 1
   <142>   DW_AT_sibling     : <0x17b>
<2><146>: Abbrev Number: 11 (DW_TAG_formal_parameter)
   <147>   DW_AT_name        : (indirect string, offset: 0x0): argc
   <14b>   DW_AT_decl_file   : 1
   <14c>   DW_AT_decl_line   : 19
   <14d>   DW_AT_type        : <0x82>
   <151>   DW_AT_location    : 2 byte block: 91 0  (DW_OP_fbreg: 0)
<2><154>: Abbrev Number: 11 (DW_TAG_formal_parameter)
   <155>   DW_AT_name        : (indirect string, offset: 0xf): argv
   <159>   DW_AT_decl_file   : 1
   <15a>   DW_AT_decl_line   : 19
   <15b>   DW_AT_type        : <0x17b>
   <15f>   DW_AT_location    : 2 byte block: 91 4  (DW_OP_fbreg: 4)
<2><162>: Abbrev Number: 18 (DW_TAG_lexical_block)
   <163>   DW_AT_low_pc      : 0x8048495
   <167>   DW_AT_high_pc     : 0x80484b5
<3><16b>: Abbrev Number: 19 (DW_TAG_variable)
   <16c>   DW_AT_name        : (indirect string, offset: 0x54): myPerson
   <170>   DW_AT_decl_file   : 1
   <171>   DW_AT_decl_line   : 21
   <172>   DW_AT_type        : <0x29>
   <176>   DW_AT_location    : 2 byte block: 74 1c  (DW_OP_breg4 (esp): 28)
<1><17b>: Abbrev Number: 9 (DW_TAG_pointer_type)
   <17c>   DW_AT_byte_size   : 4
   <17d>   DW_AT_type        : <0x181>
<1><181>: Abbrev Number: 9 (DW_TAG_pointer_type)
   <182>   DW_AT_byte_size   : 4
   <183>   DW_AT_type        : <0x187>
<1><187>: Abbrev Number: 12 (DW_TAG_base_type)
   <188>   DW_AT_byte_size   : 1
   <189>   DW_AT_encoding    : 6 (signed char)
   <18a>   DW_AT_name        : (indirect string, offset: 0x2b): char
```