technische universität
dortmund

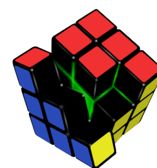
Bachelorarbeit

Systematisierung und Kombination von Kontrollfluss-Überwachungsverfahren

Henri Gründer
16. Juli 2020

Betreuer:
Dr.-Ing. Horst Schirmeier
M.Sc. Alexander Lochmann

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



Abstract

Fault tolerance beschäftigt sich mit dem Härten von Programmen gegen das Auftreten von Fehlern, die durch das Einwirken von Umwelteinflüssen, wie kosmischer Strahlung, entstehen können. In der *Software-implemented Hardware Fault tolerance* gibt es den Bereich der Kontrollflussüberwachung. Diese Verfahren überwachen bei der Ausführung eines Programms, ob es unerlaubte Sprünge im Kontrollfluss gibt.

In den vergangenen Jahren wurden verschiedene Verfahren zur Kontrollflussüberwachung vorgestellt. In dieser Arbeit werden ein paar dieser Verfahren betrachtet und kategorisiert. Aus ausgewählten wesentlichen Eigenschaften wird ein neues konfigurierbares Kontrollfluss-Überwachungsverfahren abgeleitet, welches verschiedene Vorgehen zur Überwachung vereint.

Mithilfe der LLVM-Compilerbibliotheken [9] erfolgte dann eine Implementierung als konfigurierbarer *Optimizer-Pass*. Die Leistungsfähigkeit der möglichen Methodenkombinationen wird durch Nutzung verschiedener Benchmark-Programme und dem Fehlerinjektionswerkzeug FAIL* [14] ermittelt. Dazu wird auch untersucht, wie Kontrollflussfehler mit FAIL* getestet werden können.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation und Zielsetzung 1
1.2 Vorgehen und Struktur der Arbeit 3

2 Grundlagen **5**
2.1 Kontrollfluss 5
2.1.1 Sprünge 5
2.1.2 Basisblock 5
2.2 Hardwarefehler 6
2.3 Kontrollflussfehler 6
2.4 Kontrollflussüberwachung 7
2.5 Fehlerinjektion 7
2.6 LLVM 8
2.7 Fazit 9

3 Verwandte Arbeiten **11**
3.1 Notation 11
3.2 Control-Flow Checking by Software Signatures 11
3.3 Block Signature Self Checking 12
3.4 Enhanced Control-flow Checking Using Assertions 13
3.5 Yet Another Control-flow Checking Using Assertions 14
3.6 Relationship Signatures for Control Flow Checking 15
3.7 Assertions for Control Flow Checking 16
3.8 Control Flow Checking via Regular Expressions 17
3.9 Fazit 18

4 Problemanalyse und Entwurf **19**
4.1 Verallgemeinerung der Verfahren 19
4.2 Kategorisierung 19
4.2.1 Erweiterung des Modells 20
4.2.2 Weitere Merkmale 22
4.3 Entwurf 23
4.3.1 No-Jump 23
4.3.2 Shared 23
4.3.3 Anti-Alias 24

4.3.4 Kombinierbarkeit der Methoden . . . 24
4.3.5 Mögliche Erweiterungen . . . 24
4.4 Fazit . . . 25

5 Implementierung 27
5.1 LLVM Passes . . . 27
5.2 Struktur . . . 27
5.3 Zwischensprache . . . 28
5.4 Implementierung . . . 29
5.5 Benutzung . . . 30
5.6 Einschränkungen . . . 31
5.7 Beachtenswertes . . . 31
5.8 Fazit . . . 31

6 Evaluation 33
6.1 Fehlerfreie Ausführung . . . 33
6.2 Fehlerinjektion . . . 34
6.2.1 Benchmarks . . . 34
6.2.2 Reduktion des Versuchsraums . . . 34
6.2.3 Sprunginjektion . . . 35
6.2.4 Registerinjektion . . . 36
6.3 Fazit . . . 37

7 Zusammenfassung 41
7.1 Resümee . . . 41
7.2 Ausblick . . . 42

Literaturverzeichnis 43

Abbildungsverzeichnis 45

Listingverzeichnis 47

1 Einleitung

Eine häufig angewandte Methode um die Leistungsfähigkeit von Computerhardware zu verbessern ist die Verringerung von Strukturgröße und Betriebsspannung. Das führt zu einer hohen Anfälligkeit heutiger Komponenten gegenüber Störeinflüssen, was besonders im Bereich von sicherheitskritischen oder eingebetteten Systemen beachtet werden muss.

Umwelteinflüsse wie Strahlung können zu Fehlern der Hardware, wie dem Kippen von Bits führen. Das kann neben der Verfälschung von Daten auch zu Fehlern im Kontrollfluss führen. Spezielle Hardware kann gegen solche Fehler zur Härtung eingesetzt werden. Um aber Standardhardware benutzen zu können, gibt es im Bereich der *Software-implemented Hardware Fault tolerance* Kontrollfluss-Überwachungsverfahren, die in dieser Arbeit betrachtet werden.

1.1 Motivation und Zielsetzung

In der Literatur wurden in den letzten Jahren einige Kontrollfluss-Überwachungsverfahren vorgestellt. Diese zeichnen sich teilweise durch unterschiedliche Herangehensweisen aus. Das führt dazu, dass verschiedene Verfahren auch unterschiedliche Eigenschaften in Bezug auf Fehlererkennung und benötigtem Rechenaufwand haben.

Zur Verdeutlichung wird hier eine Beispielanwendung des Verfahrens BSSC (siehe 3.3) betrachtet. Dazu werden in ein Programm Prüfinstruktionen geschrieben, die fehlerhafte Sprünge in der Ausführung (Kontrollflussfehler) erkennen sollen.

In Listing 1.1 ist eine Version des Bubblesort-Algorithmus in C dargestellt. Ein Ausschnitt (Zeile 6–15) des Programms wird in Abbildung 1.1 als Kontrollflussgraph dargestellt. Auf der rechten Seite dieser Abbildung sind dann grau unterlegt die Instruktionen von BSSC hinzugefügt.

Im rechten Graph sind auch zwei Pfeile ergänzt, die Kontrollflussfehler darstellen sollen. Hier kann man sehen, dass der Sprung des grünen Pfeils (linke Seite) erkannt würde. Die Variable S wird erst auf 2 gesetzt und die nächste Instruktion von BSSC, die ausgeführt wird, meldet einen Fehler, weil S nicht den Wert 3 hat.

Der rote Pfeil (rechte Seite) stellt einen nicht von BSSC erkannten Fehler dar. Dieser Sprung könnte durch eine verfälschte Sprunganweisung entstanden sein. Nach dem Sprung wird die Variable S mit dem neuen Wert überschrieben und der Fehler bleibt unentdeckt.

```
 1  void bubblesort(int *nums, int len) {
 2      int i, j, tmp, b1, b2, b3;
 3      i = 1;
 4      do {
 5          j = 0;
 6          do {
 7              b1 = nums[j] > nums[j + 1];
 8              if (b1) {
 9                  tmp = nums[j];
10                  nums[j] = nums[j + 1];
11                  nums[j + 1] = tmp;
12              }
13              j++;
14              b2 = j < len - i;
15          } while (b2);
16          i++;
17          b3 = i < len;
18      } while (b3);
19  }
```

Listing 1.1: Implementierung von Bubblesort in C

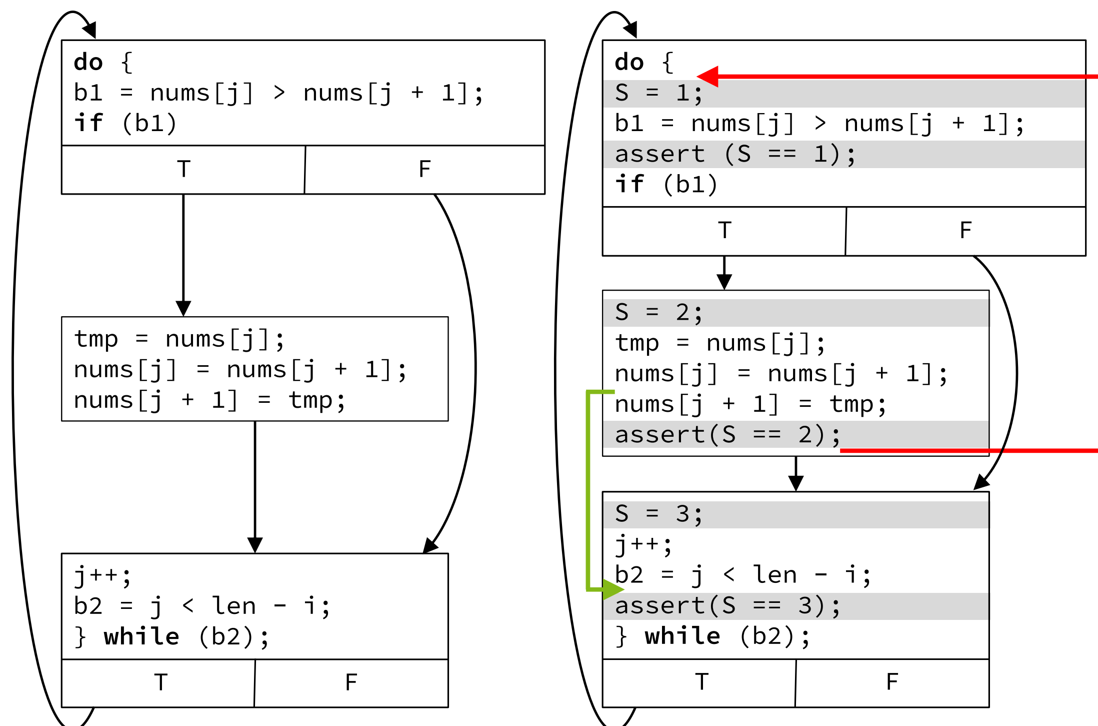


Abbildung 1.1: Kontrollflussgraph zu Abb. 1.1 und Anwendung von BSSC

Andere Verfahren würden auch oder nur den roten Kontrollflussfehler entdecken. Da bisher in der Literatur keine ausreichende Systematisierung solcher Verfahren existiert, sollen in dieser Arbeit vorhandene mit neuen Metriken kombiniert und untersucht werden. Außerdem bietet sich die Gelegenheit einer Kombination von Aspekten verschiedener Verfahren zu einem konfigurierbaren neuen Verfahren.

1.2 Vorgehen und Struktur der Arbeit

Nachdem übliche Definitionen und erforderliche Grundlagen für dieses Thema in Kapitel 2 erläutert wurden, werden in Kapitel 3 zuerst eine Reihe von Kontrollfluss-Überwachungsverfahren aus der Literatur vorgestellt. Diese Verfahren werden, unter Betrachtung von Kategorisierungen aus [5] und vorgestellten Verfahren, untersucht und strukturiert. Außerdem werden ergänzende Metriken vorgestellt, um eine bessere Kategorisierung zu ermöglichen.

Die Verfahren werden ebenfalls generalisiert betrachtet, um im nächsten Schritt ein konfigurierbares Verfahren zu entwerfen, das auf unterschiedlichen Aspekten einzelner vorgestellter Verfahren beruht. Dazu wurden vier Aspekte herausgegriffen, die relativ unabhängig voneinander und dem Umfang der Arbeit angemessen kombinierbar sind (Kapitel 4).

Der Entwurf wird anschließend in Kapitel 5 mithilfe des LLVM Framework als *Optimizer-Pass* implementiert und kann so auf verschiedenste Programme angewendet werden.

Zur Evaluation (Kapitel 6) werden die Verfahren mit Benchmarkprogrammen auf fehlerfreie Anwendbarkeit geprüft. Abschließend wird untersucht, wie mit dem Fehlerinjektionswerkzeug *FAIL** [14] Verfahren solcher Art getestet werden können und dies auf das vorgestellte Verfahren angewandt.

2 Grundlagen

In diesem Kapitel werden die für diese Arbeit benötigten Begriffe und Konzepte definiert. Es wird auf übliche Definitionen der Literatur zurückgegriffen und erläutert, welche Rolle sie für diese Arbeit spielen.

2.1 Kontrollfluss

Der Kontrollfluss eines Programms bezeichnet den Verlauf, den die Ausführung der Instruktionen nimmt.

2.1.1 Sprünge

Die Reihenfolge, in der die Instruktionen eines Programms ausgeführt werden verläuft in der Regel nicht linear, sondern es gibt an bestimmten Stellen auch Sprünge. In höheren, imperativen Programmiersprachen (bspw. C) wird der Kontrollfluss durch Sprachkonstrukte wie Konditionen (`if`), Sprünge (`goto`) oder Schleifen (`for`) beeinflusst.

Maschinenbefehlssätze enthalten Befehle wie `jmp` oder `jne` (x86 Assembler), die den Kontrollfluss beeinflussen. Diese Instruktionen beinhalten eine Information darüber, wo die Ausführung fortgeführt werden soll und optional eine Bedingung für die Abzweigung im Verlauf.

2.1.2 Basisblock

Um darzustellen, an welchen Stellen Sprünge in der Programmausführung stattfinden und wo diese hinführen, kann man ein Programm in Basisblöcke (basic block, BB) unterteilen. Dabei wird auf der Ebene von Funktionen gearbeitet.

BB fassen eine Menge an aufeinanderfolgenden Instruktionen zusammen. Sie werden so gewählt, dass sie die maximale Menge an Instruktionen enthalten, aber trotzdem die Bedingung erfüllen, dass ein Sprung nur an den Anfang eines BB und nur vom Ende eines BB aus erfolgt.

Um die möglichen Verläufe einer Programmausführung darzustellen, kann man BB zu einem Kontrollflussgraphen (control-flow graph, CFG) kombinieren. Ein CFG $G = (V, E)$ ist ein gerichteter Graph, der aus den Knoten V, die jeweils einen BB darstellen und den Kanten E, die alle gültigen Sprünge markieren, besteht.

Ein Beispiel für einen CFG wurde in Abbildung 1.1 gegeben, die Zeile 6 bis 15 aus Listing 1.1 umsetzt.

2.2 Hardwarefehler

In den vergangenen Jahren wurden viele Computer-Bauteile in immer kleineren Maßstäben hergestellt und mit immer kleineren Spannungen betrieben. Das führt aber auch dazu, dass die Bauteile elektronisch empfindlicher gegenüber Störeinflüssen sind. Falls ein Teilchen die Ladung eines Transistors beeinflusst, kann es zu einem Fehler kommen. Die Wahrscheinlichkeit dafür ist zwar gering, es kann aber z. B. durch Wirken kosmischer Strahlung oder elektromagnetischer Einflüsse ein Bitfehler erzeugt werden. Bitfehler beschreiben die Invertierung eines Bits das bspw. im Speicher oder einem Register liegt.

Manche Fehler haben keinen Einfluss auf die Ausführung von Programmen, weil sie Bereiche beeinflussen, die nicht benutzt werden oder anderweitig unerheblich für die weiterführende Ausführung sind. Es kann aber auch zu transienten Fehlern kommen, die Fehler im Programm verursachen.

Das kann die Ursache für eine Verfälschung von Daten sein. Falls diese Verfälschung unbemerkt bleibt, spricht man von *silent data corruption*.

Durch einen Bitfehler kann es aber auch zu einem Kontrollflussfehler kommen, was im nächsten Abschnitt betrachtet wird.

2.3 Kontrollflussfehler

Bei einem Kontrollflussfehler (control-flow error, CFE) wird der Kontrollfluss eines Programms beeinflusst. Dies äußert sich durch einen Sprung an eine falsche Adresse. Ursache dafür kann ein Bitfehler im *Instruction Pointer*, in einer Adresse in einem Befehl oder die falsche Auswertung in einer Sprunginstruktion sein.

In *Software-Implemented Hardware Fault Tolerance* [5] wurde für ausgewählte Verfahren bereits ein Fehlermodell entworfen und ausgewertet. Von diesen Verfahren werden in dieser Arbeit auch einige vorgestellt.

In Abbildung 2.1 sind die betrachteten fehlerhaften Sprünge durch Pfeile dargestellt. Die Typen 1, 2 und 3 beziehen sich auf einen Sprung von der letzten Instruktion eines BB zu einem anderen BB. Die Typen 1 und 2 bedingen einen Sprung an den Anfang eines BB und Typ 3 den Sprung an eine beliebige andere Stelle. Typ 2 unterscheidet sich von Typ 1 dahingehend, dass der Sprung dort (im Gegensatz zu allen anderen Typen) im CFG vorgesehen ist, aber es zur Laufzeit nicht der Richtige ist. Das wird als *wrong* bezeichnet. Die anderen Sprünge fallen unter die Bezeichnung *illegal*. Typ 4 beinhaltet Sprünge, die von einer anderen Stelle aus zu einem anderen BB führen.

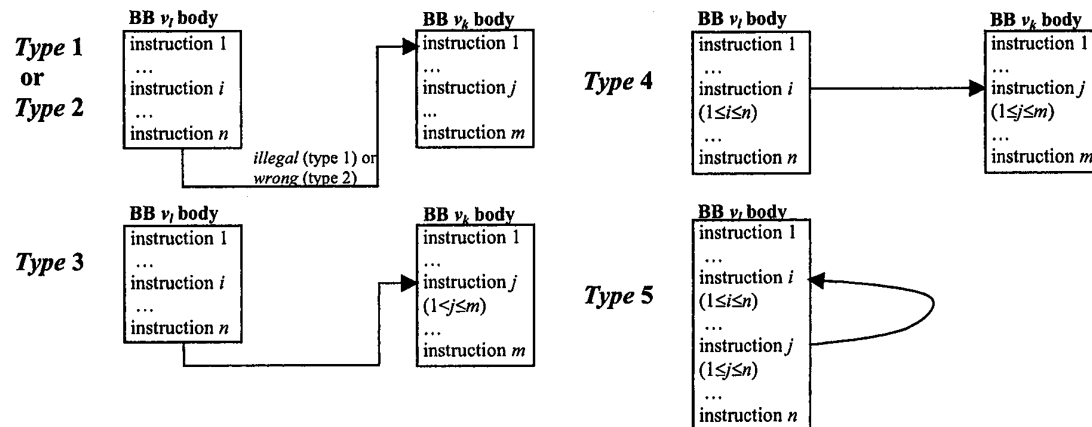


Abbildung 2.1: Fehlermodell aus SIHFT [5]

Typ 5 sind alle Sprünge, die wieder im Ausgangs-BB landen. Dieser Typ bezeichnet inter-BB Sprünge. Die anderen beziehen sich auf intra-BB Sprünge.

2.4 Kontrollflussüberwachung

Im Bereich der *Software-implemented Hardware Fault tolerance* gibt es Veröffentlichungen, die sich mit dem Erkennen von Kontrollflussfehlern beschäftigen. Diesen Bereich nennt man Kontrollflussüberwachung (control-flow monitoring, CFM).

In Kapitel 3 werden eine Menge an Verfahren aus der Literatur vorgestellt, die auf der Ebene von BB agieren und dort Instruktionen zur Erkennung von CFE einfügen. Im Allgemeinen wird hier mit einem Zustand gearbeitet, der sich eine Information wie den aktuellen BB in der Ausführung merkt. Falls durch einen Fehler in einen anderen BB gesprungen wird, kann das mit einer Prüfinstruktion festgestellt werden.

Ein Beispiel dafür wurde bereits in Abschnitt 1.1 gezeigt, wo auf den Algorithmus Bubblesort das Verfahren BSSC (Abschnitt 3.3) angewendet wird.

2.5 Fehlerinjektion

Das Auftreten von Hardwarefehlern wird häufig, wenn überhaupt, nur durch Zufall entdeckt und tritt nur sehr selten auf. Das Testen der Fehlertoleranz von Programmen muss daher mit absichtlich erzeugten Fehlern passieren.

In [8] und anderen Arbeiten wurde beschrieben, wie man mit dem Einsatz von Strahlung vorsätzlich Hardwarefehler erzeugen kann. Das Testen mit dieser Methode hat jedoch den Nachteil, dass man keine wirkliche Kontrolle über die Stelle oder das Ausmaß einer Fehlerinjektion hat.

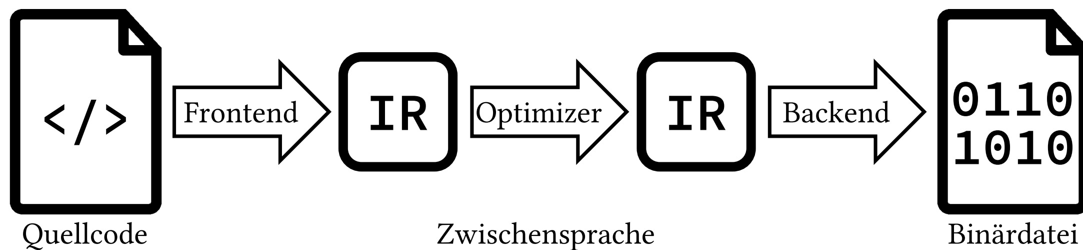


Abbildung 2.2: Architektur von LLVM

In dieser Arbeit wird daher auf das Tool FAIL* [14] zurückgegriffen, mit dem die Ausführung von Software simuliert wird, in die dann kontrolliert Bitfehler injiziert werden können.

Zur Benutzung wird zuerst ein *Golden Run* durchgeführt, der einen unbeeinflussten Durchlauf des Programms aufzeichnet. Die Daten dieser Ausführung werden zusammen mit möglichen Experimenten in eine Datenbank importiert, von der aus dann Experimente geplant und durchgeführt werden können. Dafür können dann verschiedene Formen von Fehlerinjektion geplant werden. Für diese Arbeit wurde die Injektion von zufälligen Sprüngen im Programm (also CFE) und die Injektion von Bitfehlern in Register genutzt.

Für die Fehlerinjektion ist neben dem Abdecken aller möglichen Stellen im Fehlerraum auch das stichprobenartige Untersuchen möglich, bei dem versucht wird eine möglichst gute Abdeckung des Fehlerraums unter einer geringen Stichprobengröße zu bieten.

2.6 LLVM

LLVM ist ein Compilerframework, dessen Entwicklung 2000 an der University of Illinois at Urbana–Champaign unter Vikram Adve und Chris Lattner gestartet wurde [9].

Die herausragende Eigenschaft dieses Frameworks ist, dass Frontend und Backend klar getrennt und durch eine Zwischensprache (intermediate representation, IR) verbunden sind (siehe Abbildung 2.2). Es gibt heute für viele Programmiersprachen ein Frontend zur Übersetzung in die Zwischensprache. Außerdem existieren auch für viele Plattformen Backends, welche die Zwischensprache in die entsprechende Architektur übersetzen. Somit ist eine Vielzahl an Kombinationen aus Programmiersprache und Zielplattform möglich.

In dieser Arbeit geht es insbesondere um die LLVM-Zwischensprache, auf der von dem LLVM-Optimizer *opt* viele Optimierungen durchgeführt werden. Als Beispiel ist in Listing 2.1 die Funktion aus Listing 1.1 in IR dargestellt. Man sieht, dass die IR nach Funktionen die BB als nächste Ordnungsstruktur benutzt, was für die Anwendung von CFM-Verfahren von Vorteil ist.

```
1  define void @bubblesort(i32* nocapture %0, i32 %1) {
2    %3 = sext i32 %1 to i64
3    %4 = sext i32 %1 to i64
4    br label %5
5  5:
6    %6 = phi i64 [ %20, %19 ], [ 1, %2 ]
7    %7 = sub nsw i64 %3, %6
8    br label %8
9  8:
10   %9 = phi i64 [ %12, %17 ], [ 0, %5 ]
11   %10 = getelementptr inbounds i32, i32* %0, i64 %9
12   %11 = load i32, i32* %10, align 4, !tbaa !4
13   %12 = add nuw nsw i64 %9, 1
14   %13 = getelementptr inbounds i32, i32* %0, i64 %12
15   %14 = load i32, i32* %13, align 4, !tbaa !4
16   %15 = icmp sgt i32 %11, %14
17   br i1 %15, label %16, label %17
18 16:
19   store i32 %14, i32* %10, align 4, !tbaa !4
20   store i32 %11, i32* %13, align 4, !tbaa !4
21   br label %17
22 17:
23   %18 = icmp slt i64 %12, %7
24   br i1 %18, label %8, label %19
25 19:
26   %20 = add nuw nsw i64 %6, 1
27   %21 = icmp slt i64 %20, %4
28   br i1 %21, label %5, label %22
29 22:
30   ret void
31 }
```

Listing 2.1: Bubblesort in IR

LLVM stellt eine relativ einfache Schnittstelle zur Verfügung, mit der *Optimizer Passes* ergänzt und dynamisch eingebunden werden können. Zusätzlich wird in dieser Arbeit das Frontend Clang betrachtet, welches ein Frontend für die Sprachen C und C++ darstellt. Das Programm *llc* kann IR in Assembler übersetzen.

2.7 Fazit

Die grundlegenden Themen und Begriffe für diese Arbeit aus den Bereichen Kontrollfluss, Hardwarefehler und Kontrollflussfehler im speziellen, sowie LLVM hat dieses Kapitel erläutert. Insbesondere wurde das Fehlermodell aus [5] vorgestellt. Im Folgenden werden einige CFM-Verfahren aus der Literatur betrachtet.

3 Verwandte Arbeiten

Hier werden ausgewählte CFM-Verfahren vorgestellt, die verschiedene Aspekte der Kategorisierung verdeutlichen sollen und für die spätere Implementierung Vorlagen geliefert haben. Neben einer Beschreibung des jeweiligen Verfahrens gibt es ein Pseudocodebeispiel dazu, wie ein BB modifiziert wird und eine beispielhafte Anwendung auf einen Ausschnitt des Bubblesort Codes aus Abschnitt 2.6. Die Verfahren wurden so ausgewählt, dass sie möglichst viele Aspekte der Verfahren, die in der Literatur vorgestellt wurden, abdecken.

3.1 Notation

Um ein einheitliches Bild bei der Vorstellung der Verfahren zu liefern und die Vergleichbarkeit zu verbessern, werden gemeinsame Definitionen der Konzepte verwendet. Diese sind in der folgenden Übersicht dargestellt:

$V = \{b_1, b_2, \dots\}$	Liste aller Basisblöcke einer Funktion
$E = \{br_{1,2}, br_{2,3}, \dots\}$	Liste aller gültigen Sprünge zwischen Basisblöcken
s_i	Eine Signatur, die zu b_i gehört
S	Variable, die den aktuellen Zustand speichert
$\mathrm{suc}(b_x)$	Nachfolgerfunktion: alle b_i, für die gilt $br_{x,i} \in E$
$\mathrm{pred}(b_x)$	Vorgängerfunktion: alle b_i, für die gilt $br_{i,x} \in E$
$-x$	Numerische Negation (von x)
$\overline{x}$	Logische Negation (von x)
$\&$	Bitweises UND
$\vert$	Bitweises ODER
$\oplus$	Bitweises exklusives ODER
`[BB]`	Instruktionen des ursprünglichen BB

Alle vorgestellten Verfahren ordnen jedem BB einen Wert zu, der diesen identifiziert. Dafür wird hier der Name s_i verwendet. Außerdem wird fast immer ein Zustand vorgehalten, was hier in der Variable S geschieht.

3.2 Control-Flow Checking by Software Signatures

Control-Flow Checking by Software Signatures (CFCSS) wird in [12] beschrieben. In dem Verfahren wird jedem b_i neben s_i ein weiterer Wert d_i zugeordnet. Der Wert von

d_i wird so gewählt, dass $s_j \oplus d_i = s_i$ für alle $b_j \in \text{pred}(b_i)$ gilt. Falls $|\text{pred}(b_i)| > 1$ gilt, muss für jeden $b_j \in \text{pred}(b_i)$ ein zusätzliches D_j erzeugt werden, sodass S auf denselben Wert bei allen Vorgängern von s_i gesetzt wird.

Die Prüfung $S = s_i$ wird am Anfang jedes BB durchgeführt und danach wird S auf den Wert des nächsten BB gesetzt. Das führt zu einem BB-Aufbau wie unten dargestellt.

In dem Paper wird auch auf das Problem des *Aliasing* eingegangen. Es tritt hier auf, wenn ein bestimmtes Konstrukt im CFG vorkommt. *Aliasing* wird im Abschnitt 4.2.1 genauer erläutert.

```
1  assert S = s_i
2  S ← S ⊕ D_i ⊕ d_i
3  [BB]
```

Listing 3.1: BB ergänzt durch CFCSS als Pseudocode

```
1  S = 0b0010;
2  do {
3  B3: // s = 0b0011
4      S = S ^ 0b0001 ^ 0
5      assert (S == 0b0011);
6      if (nums[j] > nums[j + 1]) {
7      B4: // s = 0b0100
8          S = S ^ 0b0111 ^ 0b0111;
9          assert (S == 0b0100);
10         tmp = nums[j];
11         nums[j] = nums[j + 1];
12         nums[j + 1] = tmp;
13     }
14 B5: // s = 0b0101
15     S = S ^ 0b0110 ^ 0b0111;
16     assert (S == 0b0101);
17     j++;
18 } while (j < len - i);
```

Listing 3.2: Bubblesort ergänzt durch CFCSS

3.3 Block Signature Self Checking

Block Signature Self Checking (BSSC) wird in [11] beschrieben. Dort wird neben BSSC ein weiteres Verfahren „Error Capturing Instructions“ beschrieben, das Instruktionen einfügt, die Sprünge in Bereiche des Programms feststellen sollen, die bei einer normalen Ausführung nicht erreicht werden.

BSSC fügt in jeden BB am Anfang und Ende je eine Instruktion ein. Zu Beginn wird S auf s_i gesetzt. Am Ende wird geprüft, ob S immer noch den Wert s_i enthält. Der Aufbau ist unten dargestellt.

```
1  S ← s_i
2  [BB]
3  assert S = s_i
```

Listing 3.3: BB ergänzt durch BSSC als Pseudocode

```
1  do {
2  B2:  // next: B3, B4; prev: B1
3      S = 2;
4      b1 = nums[j] > nums[j + 1];
5      assert(S == 2);
6      if (b1) {
7      B3:  // next: B4; prev: B2
8          S = 3;
9          tmp = nums[j];
10         nums[j] = nums[j + 1];
11         nums[j + 1] = tmp;
12         assert(S == 3);
13     }
14 B4:  // next: B2, B5; prev: B2, B3
15     S = 4;
16     j++;
17     b2 = j < len - i;
18     assert(S == 4);
19 } while (b2);
```

Listing 3.4: Bubblesort ergänzt durch BSSC

3.4 Enhanced Control-flow Checking Using Assertions

Enhanced Control-flow Checking Using Assertions (ECCA) wird in [1] beschrieben. Es soll eine verbesserte Version von CCA [7] sein. Bei ECCA sind die s_i ausschließlich Primzahlen und es wird mit dem Produkt der Nachfolgeblöcke gearbeitet. Am Anfang jedes Blocks wird S geprüft und auf s_i des aktuellen Blocks gesetzt. Am Ende jedes Blocks wird S ebenfalls geprüft und auf einen Wert gesetzt, der den folgenden Sprung widerspiegelt.

Die Division am Anfang des BB erzeugt eine Division durch Null im Fehlerfall. Ein Fehler am Ende des BB wird erst an einem nächsten Blockanfang erkannt. Durch die Division durch Null im Fehlerfall ist kein weiterer Behandlungscode nötig.

Ebenfalls wird die Option beschrieben, die Auflösung des Verfahrens zu verringern und als kleinste Einheit mehrere BB zusammenzufassen.

1 $S \leftarrow s_i/(\overline{(S \mod s_i)}(S \mod 2))$
2 [**BB**]
3 $S \leftarrow \prod(s_x : b_x \in \mathrm{suc}(b_i)) + \overline{\overline{(S - s_i)}}$

Listing 3.5: BB ergänzt durch ECCA als Pseudocode

```
1  S = 5;
2  do {
3  B2:  // next: B3, B4; prev: B1
4      S = 7 / ((!(S % 7)) * (S % 2));
5      b1 = nums[j] > nums[j + 1];
6      S = 143 + !!(S - 7);
7      if (b1) {
8      B3:  // next: B4; prev: B2
9          S = 11 / ((!(S % 11)) * (S % 2));
10         tmp = nums[j];
11         nums[j] = nums[j + 1];
12         nums[j + 1] = tmp;
13         S = 13 + !!(S - 11);
14     }
15 B4:  // next: B2, B5; prev: B2, B3
16     S = 13 / ((!(S % 13)) * (S % 2));
17     j++;
18     b2 = j < len - i;
19     S = 119 + !!(S - 13);
20 } while (b2);
```

Listing 3.6: Bubblesort ergänzt durch ECCA

3.5 Yet Another Control-flow Checking Using Assertions

Yet Another Control-flow Checking Using Assertions (YACCA) wird in [4] beschrieben und in [3] überarbeitet. Jedem BB werden zwei Signaturen $s1_i$ und $s2_i$ zugeordnet. Am Anfang und Ende eines BB findet je ein Setzen und Prüfen von S statt.

Der Zustand S wird am Anfang auf den Wert von $s1_i$ und am Ende auf den Wert von $s2_i$ gesetzt. Statt direkt auf einen Fehler zu prüfen, wird lediglich eine Variable ERR_CODE gesetzt, die seltener geprüft werden kann. Statt S direkt zu setzen, wird es in Abhängigkeit des vorigen Wertes gesetzt.

Im einfachsten Fall ist $M1 = 1$ und $M2 =$ <alter Wert> $\oplus$ <neuer Wert>. Für mehrere mögliche alte Werte, muss $M1$ eingesetzt werden, um alle unterschiedlichen Bits der möglichen alten Werte auf 0 zu setzen.

1 $ERR_CODE \leftarrow ERR_CODE \mid ((S \neq s2_{j1}) \,\&\, (S \neq s2_{j2}) \,\&\, \dots)$
2 $S \leftarrow (S \,\&\, M1) \oplus M2$ `// setze auf` $s1_i$
3 `[BB]`
4 $ERR_CODE \leftarrow ERR_CODE \mid (S \neq s1_i)$
5 $S \leftarrow (S \,\&\, M1) \oplus M2$ `// setze auf` $s2_i$

Listing 3.7: BB ergänzt durch YACCA als Pseudocode

```
1  S = 1;
2  do {
3  B2:  // next: B3, B4; prev: B1
4      ERR_CODE |= S != 1;
5      assert(!ERR_CODE);
6      b1 = nums[j] > nums[j + 1];
7      S ^= 3;  // = B1 ^ B2
8      if (b1) {
9      B3:  // next: B4; prev: B2
10         ERR_CODE |= S != 2;
11         assert(!ERR_CODE);
12         tmp = nums[j];
13         nums[j] = nums[j + 1];
14         nums[j + 1] = tmp;
15         S ^= 1;  // = B2 ^ B3
16     }
17 B4:  // next: B2, B5; prev: B2, B3
18     ERR_CODE |= S != 2 && S != 3;
19     assert(!ERR_CODE);
20     j++;
21     b2 = j < len - i;
22 //    = (S & ~(B2 ^ B3)) ^ (B2 & ~(B2 ^ B3) ^ B3)
23     S = (S & 0xfffffffe) ^ 1;
24 } while (b2);
```

Listing 3.8: Bubblesort ergänzt durch YACCA

3.6 Relationship Signatures for Control Flow Checking

Relationship Signatures for Control Flow Checking (RSCFC) wird in [10] beschrieben. Hier wird S als Bitvektor betrachtet. Jedem b_i ist die Bitposition i zugeordnet, ablesbar an $L_i = 0 \cdots 010 \cdots 0$, welches nur an Stelle i den Wert 1 hat. Die s_i stellen für jeden BB alle möglichen Nachfolger dar und sind für jedes $b_x \in \mathrm{suc}(b_i)$ an der Stelle x 1. Die Berechnung von $S = s_i \& (-\overline{(S \oplus L_i)})$ aktualisiert den Zustand. Ein Fehler liegt vor, wenn $S(= L_i) \neq S\&L_i$.

```
1  assert S = S & L_i
2  [BB]
3  S ← s_i & (−!(S ⊕ L_i))
```

Listing 3.9: BB ergänzt durch RSCFC als Pseudocode

```
1  do {
2  B2:  // next: B3, B4; prev: B1
3      assert(S = S & 3);
4      b1 = nums[j] > nums[j + 1];
5      S = 0b10011000 & (-!(S ^ 3));
6      if (b1) {
7      B3:  // next: B4; prev: B2
8          assert(S = S & 4);
9          tmp = nums[j];
10         nums[j] = nums[j + 1];
11         nums[j + 1] = tmp;
12         S = 0b10010000 & (-!(S ^ 4));
13     }
14 B4:  // next: B2, B5; prev: B2, B3
15     assert(S = S & 5);
16     j++;
17     b2 = j < len - i;
18     S = 0b10100100 & (-!(S ^ 5));
19 } while (b2);
```

Listing 3.10: Bubblesort ergänzt durch RSCFC

3.7 Assertions for Control Flow Checking

Assertions for Control Flow Checking (ACFC) wird in [15] beschrieben. Es benutzt wie RSCFC ebenfalls Bitvektoren zur Zustandsspeicherung. Es wird in die meisten BB nur eine Instruktion eingefügt ($S \leftarrow S \oplus s_i$), die das Bit, das für den aktuellen BB steht, invertiert. Dabei werden alternative BB (z. B. ein if und ein else BB) mit demselben Wert für s_i versehen, um am Ende bei beiden Pfaden den selben Wert für S zu erhalten.

Die Prüfung auf Kontrollflussfehler findet nur am Ende einer Funktion und bei Schleifen statt, weil dort S zurückgesetzt werden muss.

```
1  S ← S ⊕ s_i
2  [BB]
3  assert S = 011...1 // nur, falls noetig
```

Listing 3.11: BB ergänzt durch ACFC als Pseudocode

```
1  do {
2  B2:  // next: B3, B4; prev: B1
3      ES = ES ^ 0b100;
4      b1 = nums[j] > nums[j + 1];
5      if (b1) {
6      B3a:  // next: B4; prev: B2
7          ES = ES ^ 0b1000;
8          tmp = nums[j];
9          nums[j] = nums[j + 1];
10         nums[j + 1] = tmp;
11     } else {
12     B3b:
13         ES = ES ^ 0b1000;
14     }
15 B4:  // next: B2, B5; prev: B2, B3
16     ES = ES ^ 0b10000;
17     assert(ES == 0b11111);
18     ES = 0b11;
19     j++;
20     b2 = j < len - i;
21 } while (b2);
```

Listing 3.12: Bubblesort ergänzt durch ACFC

3.8 Control Flow Checking via Regular Expressions

Control Flow Checking via Regular Expressions (CFCRE) wird in [2] beschrieben. Die Signatur s_i wird für jeden Block eindeutig gewählt. Es wird eine reguläre Sprache erzeugt, für die die Menge aller s_i das Eingabealphabet bildet. Aus dem CFG kann ein regulärer Ausdruck erzeugt werden, der alle gültigen Pfade abbildet. Ein separater Prozess kann dann z. B. mithilfe von Interprozesskommunikation die Ausführung prüfen. Dazu sendet das Programm am Ende jedes b_i das zugehörige s_i an den Überwachungsprozess. Dieser verwendet einen endlichen Automaten, um die Folge der Signaturen zu prüfen.

Im Beispiel wurde der Automat nicht mit angegeben. Dieser muss aber lediglich folgenden regulären Ausdruck akzeptieren: $(2(3)?4)^+$.

```
1  [BB]
2  send s_i
```

Listing 3.13: BB ergänzt durch CFCRE als Pseudocode

```
1  do {
2  B2:  // next: B3, B4; prev: B1
3      b1 = nums[j] > nums[j + 1];
4      send(2);
5      if (b1) {
6      B3:  // next: B4; prev: B2
7          tmp = nums[j];
8          nums[j] = nums[j + 1];
9          nums[j + 1] = tmp;
10         send(3);
11     }
12 B4:  // next: B2, B5; prev: B2, B3
13     j++;
14     b2 = j < len - i;
15     send(4);
16 } while (b2);
```

Listing 3.14: Bubblesort ergänzt durch CFCRE

3.9 Fazit

In diesem Kapitel gab es eine kurze Einführung in ausgewählte Verfahren, die in vergangenen Jahren vorgestellt wurden. Neben einer Beschreibung als Pseudocode gab es ein Codebeispiel, das die Anwendung an einem Ausschnitt des Bubblesort-Code aus der Einleitung zeigt. Das nächste Kapitel beschreibt Gemeinsamkeiten und Unterschiede dieser Verfahren und eine Erweiterung des vorher gezeigten Fehlermodells.

4 Problemanalyse und Entwurf

In diesem Kapitel geht es darum, die vorgestellten Verfahren zueinander in Bezug zu setzen, sowie zu Kategorisieren. Außerdem soll ein konfigurierbares Verfahren erarbeitet werden, das verschiedene Elemente der Verfahren aus Kapitel 3 vereint.

4.1 Verallgemeinerung der Verfahren

Abstrahiert von den Optimierungen der einzelnen Verfahren ergibt sich ein einheitliches Bild der Methodik. Die vorgestellten Verfahren haben auf einer abstrakten Ebene betrachtet einige Merkmale gemein.

Da der Kontrollfluss auf der Ebene von BB definiert ist, ist es naheliegend, dass die Verfahren auch auf dieser Ebene operieren. Die meisten Verfahren fügen Instruktionen an Anfang oder Ende der BB ein. Manche Verfahren gruppieren aber auch BB zu größeren Einheiten, um weniger Overhead zu erzeugen.

Den BB werden IDs zugeordnet. Dies kann meist beliebig geschehen. Manchmal müssen dabei aber auch bestimmte Regeln beachtet werden. Diese sind dann aber für jedes Verfahren unterschiedlich.

Jedes Verfahren benutzt eine Variable um den Zustand zu speichern. Im Allgemeinen ist damit eine Form von Identifikation des/der akuellen BB während der Ausführung gemeint. Beim Setzen des Zustandes wird entweder direkt eine ID gesetzt oder zusätzlich eine Rechung durchgeführt, die zum Ziel hat, bei einer möglichen übersprungenen Anweisung des Verfahrens den Fehler trotzdem noch etwas später zu erkennen.

4.2 Kategorisierung

Durch die unterschiedlichen Eigenschaften der Implementierung der Verfahren haben diese auch unterschiedliche Eigenschaften im Bezug auf Fehlererkennung. Eine Möglichkeit zur Kategorisierung der Verfahren bietet ein Fehlermodell. Danach kann bestimmt werden, welches Verfahren welche Fehlertypen abdeckt.

In Abschnitt 2.3 wurde bereits ein Fehlertypen-Modell vorgestellt. Dieses ordnet alle betrachteten Verfahren mit Ausnahme von YACCA und BSSC in die Fehlertypen 1, 3 und 4 ein. Bei den meisten Verfahren werden diese Fehlerklassen aber nur unter Einschränkungen abgedeckt. Als Beispiel sei hier CFCSS mit *Aliasing* genannt. BSSC deckt lediglich Typ 3 und 4 ab, weil es nur Sprünge bemerkt, die nicht an den Anfang eines

BB springen. YACCA deckt alle Fehlertypen ab, da in dem Paper auf zwei Techniken verwiesen wird, die Typ 2 und 5 abdecken und ergänzend verwendet werden sollen.

4.2.1 Erweiterung des Modells

Anhand der betrachteten Verfahren in Kapitel 3 und des Modells aus dem vorigen Abschnitt, wird im Folgenden eine Verallgemeinerung des Fehlertypen-Modells vorgestellt, um die Fähigkeiten der Verfahren besser zu unterscheiden.

In einem ersten Schritt soll das Modell erweitert werden, um alle möglichen Fälle unterscheiden zu können. Wenn man das Modell abstrakter betrachtet, kann man ein Muster für die dargestellten Fehlerarten erkennen. Es lassen sich verschiedene Dimensionen zur Unterscheidung benennen. Aus dem Kreuzprodukt aller Dimensionen lassen sich dann alle möglichen Fälle ablesen.

Position im BB Die Trennung von Quell- und Zielposition im BB stellt die ersten beiden Dimensionen dar. Hierbei kann man von den BB der modifizierten oder der ursprünglichen Funktion sprechen. Wenn man von der modifizierten Version ausgeht, kann man den neu entstandenen CFG betrachten oder den alten CFG behalten und die Ergänzungen als zugehörig zu den alten BB betrachten. Hier wird die letzte Variante gewählt.

Die Betrachtung der Prüfinstruktionen als Teil der Menge möglicher Sprünge führt dazu, dass Verfahren wie BSSC, die den neuen Wert des Zustandes nicht von dem alten Wert abhängig machen, schlechter abschneiden, wenn das Ziel eines Sprungs zwischen der Prüf- und Setz-Instruktion eines Verfahrens liegt. Bei dem Verfahren CFCSS wird gut gezeigt, wie ein Fehler trotz zweimaliger Ausführung der Setz-Instruktion erkannt werden kann. Dadurch, dass S wieder den Wert von S bekommt und nur die passenden Bits invertiert werden, um den neuen Wert zu erhalten, wird der Fehler dort trotzdem an der nächsten Prüfinstruktion erkannt.

Für Quelle und Ziel des Sprungs kann also zwischen Anfang des Blocks, in dem Prüfcode am Anfang, Mitte des Blocks, in dem Prüfcode am Ende und Ende des Blocks unterschieden werden.

Relation der BB Durch Verfahren wie ACFC und ECCA ist es ebenfalls sinnvoll, Relationen der BB im Bezug auf den CFG zu betrachten. Das macht dann Sinn, wenn weniger häufig als einmal pro BB der Zustand geprüft wird. Es ist dann von Bedeutung, ob durch einen CFE der Zustand, der sich in so einem Fall einen Verlauf mehrerer BB merkt, eventuell trotzdem gültig bleibt. Hier lassen sich drei Relationen von BB unterscheiden. Wenn zwei BB einen alternativen Pfad zum selben Ziel bilden (z. B. durch if-else), wird das in ACFC *Multi-Path* genannt. Außerdem gibt es noch den Fall eines Rück- und Vorwärtssprungs, was einer Wiederausführung bzw. dem Überspringen eines BB entspricht.

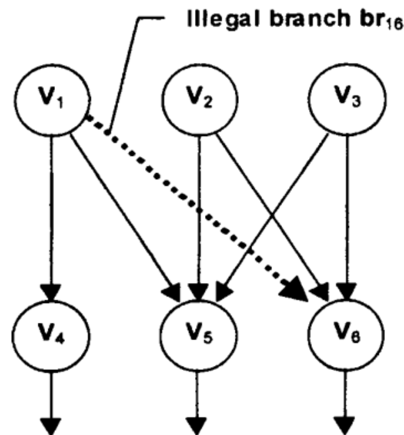


Abbildung 4.1: CFG, bei dem *Aliasing* entstehen kann aus CFCSS [12]

Eine weitere Beziehung der BB, die bereits genannt wurde, ist die Unterscheidung zwischen *illegal* und *wrong* und ob Quelle und Ziel derselbe BB ist.

Die letzte Konstellation im CFG, die noch interessant ist, ist die bei der Aliasing auftreten kann. Davon ist z. B. CFCSS betroffen. Es kann entstehen, wenn für einen Nachfolger alle Vorgänger den Status auf denselben Wert setzen und es mehrere Nachfolger gibt. In Abbildung 4.1 wird die nötige Konstellation dafür gezeigt. Hier setzen die BB V_1 bis V_3 alle den Zustand auf denselben Wert, damit die möglichen folgenden BB V_4 bis V_6 sie als Vorgänger akzeptieren. Dadurch wird der Sprung von V_1 nach V_6 ebenfalls als gültig angesehen.

Interfunktionssprünge Eine weitere naheliegende Dimension wurde bereits in der Literatur benannt [5]. Dort wurden die Begriffe Intra- und Interfunktionssprung benutzt, um zu beschreiben, ob ein Sprung über die Grenzen einer Funktion hinaus führt. Dazu sind bisher keine Ergebnisse bekannt. In dem Kontext wäre auch ein Fall für Speicher, der nicht zum Programm gehört bzw. gar nicht in einem Textsegment liegt, denkbar. Diese Verfahren (siehe z. B. ECI [11]) sind aber im Umfang dieser Arbeit nicht enthalten.

Zusammenfassung Die vorgestellten Dimensionen ergeben zusammengefasst folgende Liste:

- Position im BB für Sprungquelle und Sprungziel
  - Anfang
  - Prüfbereich am Anfang
  - Mitte
  - Prüfbereich am Ende
  - Ende

- Relation der BB
  - Multi-Path
  - Rückwärtssprung
  - Vorwärtssprung
  - Illegal
  - Wrong
  - Alias
- Interfunktionssprünge
  - Andere Funktion
  - Anderes Programm
  - Außerhalb des Textsegments

Man muss beachten, dass das hier genannte Modell für Verfahren, die auf der Ebene von BB arbeiten gut geeignet ist. Durch den hohen Bezug zu BB sind andere Verfahren, die nicht diesem Schema entsprechen, hier aber nicht so gut erfasst.

4.2.2 Weitere Merkmale

Neben den abgedeckten Fehlerarten gibt es natürlich noch andere Eigenschaften, die von Interesse für die Bewertung der Verfahren sind.

Eine Eigenschaft ist die Latenz bis zur Erkennung eines Fehlers. Diese Eigenschaft ist bei den meisten Verfahren aus dieser Arbeit sehr ähnlich, weil die Verfahren in jeden Basisblock eine Prüfung einfügen, die ermittelt, ob ein Fehler erkannt wurde. Es gibt jedoch auch Verfahren, die nur seltener auf Fehler prüfen, wie ECCA oder ACFC. Auch bei CFCRE kann es zu Verzögerungen kommen, abhängig davon, wie schnell der prüfende Prozess arbeitet.

Eine weitere wichtige Komponente ist die Menge an Instruktionen, die dem Programm hinzugefügt werden. Hier ist ein niedriger Wert natürlich erstrebenswerter, weil durch die Verlängerung der Laufzeit auch die Wahrscheinlichkeit steigt, dass überhaupt ein Fehler eintritt. Außerdem spielt bei manchen (embedded) Anwendungen auch die Programmgröße eine Rolle. Beispiele für wenige benötigte Instruktionen sind hier ACFC, CFCSS und BSSC. ECCA und YACCA benötigen hingegen relativ viele Instruktionen.

Alle Verfahren haben gemeinsam, dass der ergänzte Code am Anfang oder Ende eines BB eingefügt wird. Je nach Art der Kontrollflussfehler lässt sich hier eine unterschiedliche Rate vermuten, je nachdem, an welche Stelle im BB der Code eingefügt wird (insb. die Fehlerprüfung).

4.3 Entwurf

Eine Auswahl der betrachteten Eigenschaften wird im folgenden Kapitel implementiert. Hier wird beschrieben, welche Elemente in der Implementierung umgesetzt werden und in welcher Form.

Da alle Verfahren einen Zustand speichern, ist es naheliegend, ebenfalls auf diese Weise vorzugehen. In den hier implementierten Verfahren soll jedem BB eine ID zugeordnet werden. Es wird der Unterschied zwischen der Prüfung zu Beginn und am Ende des BB untersucht. Außerdem wird auf das Problem des *Aliasing* eingegangen und Methoden mit unterschiedlich großer Anzahl hinzuzufügender Instruktionen benutzt. Dazu wurden vier Methoden entworfen, die die unterschiedlichen Punkte abbilden.

4.3.1 No-Jump

Die Methode No-Jump entspricht BSSC. Hier wird lediglich am Anfang jedes Blocks der Zustand auf die ID des Blocks gesetzt. Am Ende wird geprüft, ob der Zustand immer noch der aktuellen ID entspricht.

Dieses Verfahren bietet wahrscheinlich eine relativ schlechte Fehlerabdeckung, da keine falschen Sprünge von der Sprunginstruktion aus erkannt werden. Im Gegenzug dafür ist der Zuwachs an Instruktionen für das Programm aber sehr gering.

```
1 S ← s_i
2 [BB]
3 assert S = s_i
```

Listing 4.1: BB ergänzt durch No-Jump in Pseudocode

4.3.2 Shared

Die Methode Shared ist an einige Verfahren angelehnt (z. B. CFCSS), die einen gemeinsamen Wert der Statusvariablen für alle Vorgänger haben. Sie ist in die Varianten Shared-Begin und Shared-End unterteilt. Die Methode setzt am Ende des Blocks eine *shared ID*. Diese wird im nächsten Block geprüft.

Die Varianten Begin und End unterscheiden sich durch die Stelle, an der die Prüfung stattfindet. Bei Begin ist das der Blockanfang und bei End das Ende. Damit kann untersucht werden, wie sich in dieser Art CFM die Positionierung der Prüfinstruktion auswirkt.

Die *shared ID* bildet die Funktionsweise aus CFCSS nach, die das Problem des *Aliasing* hat. Sie wird gebildet, indem für jeden BB alle Vorgänger-BB dieselbe ID zugewiesen bekommen. So muss die Prüfung nur für die eine *shared ID* stattfinden und nicht alle möglichen Vorgänger unterscheiden.

Für den BB b_i ist d_i im Listing 4.2 die *shared ID* der Vorgänger und s_i die eigene.

```
1 assert S = d_i // Shared-Begin
2 [BB]
3 assert S = d_i // Shared-End
4 S ← s_i
```

Listing 4.2: BB ergänzt durch Shared in Pseudocode

4.3.3 Anti-Alias

Die Methode Anti-Alias ist an ECCA und YACCA angelehnt. Sie geht wie Shared-End vor, unterbindet jedoch das Problem des *Aliasing* durch Benutzung eindeutiger IDs. Dadurch ist die Prüfung aber auch aufwendiger und es werden im Gegensatz zu den anderen drei Methoden zusätzliche Instruktionen zum Vergleich benötigt.

Es wird so vorgegangen, dass die IDs ausschließlich als Primzahlen gewählt werden. Zur Prüfung wird dann das Produkt aller Vorgänger-IDs d_i benutzt. Dazu wird $d_i \bmod S = 0$ überprüft, da durch die Berechnung der d_i ausschließlich die Primfaktoren restlos dividiert werden können.

```
1 assert d_i mod S = 0
2 [BB]
3 S ← s_i
```

Listing 4.3: BB ergänzt durch Anti-Alias in Pseudocode

4.3.4 Kombinierbarkeit der Methoden

Die Methode No-Jump ist mit allen anderen Methoden kombinierbar, da der Zustand S am Anfang des BB gesetzt wird und die anderen Methoden dies jeweils am Ende tun.

Die Methoden Shared-Begin und Shared-End lassen sich untereinander ebenfalls kombinieren, weil das lediglich zu zwei statt einer Prüfinstruktion führt. In dem Fall wird S nur einmal am Ende des BB gesetzt und sowohl am Anfang als auch Ende der Wert geprüft.

Die Kombination aus Shared-Begin und Anti-Alias hat einen ähnlichen Effekt, nur dass zwei verschiedene IDs gesetzt werden, von denen eine am Anfang und eine am Ende geprüft wird. Die Kombination aus Shared-End und Anti-Alias ist wahrscheinlich nicht sinnvoll, weil das auf dieselbe Positionierung der Instruktionen hinausläuft. Anti-Alias bildet die gleiche Positionierung mit einem komplexeren Prüfen von S ab.

4.3.5 Mögliche Erweiterungen

In YACCA wird ebenfalls eine Methode vorgestellt, die beschreibt, wie man Kontrollflussfehler, die sich im selben BB auf Instruktionsebene auswirken (also zwischen

Instruktionen im selben BB), feststellen kann. Das wäre eine mögliche Erweiterung, die unabhängig von den obigen Methoden implementiert werden könnte.

Ebenfalls wäre eine Ergänzung denkbar, die am Beginn eines BB erneut die Sprungbedingung prüft, um festzustellen, ob evtl. die vorige Auswertung der Sprungbedingung fehlerhaft war. Diese Erweiterung würde Fehler von Sprüngen abdecken, die im CFG *legal* sind, aber zur Laufzeit nicht korrekt, weil von zwei möglichen Zielen durch die fehlerhafte Auswertung das Falsche gewählt wird.

Um die Menge an hinzugefügten Instruktionen zu verringern, kann man die Methoden anpassen, sodass eine Prüfung nur noch in größeren Abständen vorgenommen wird, was bereits in der Metrik erwähnt wurde.

Die beiden erstgenannten Erweiterungen passen nicht in das Schema der hier betrachteten Verfahren, wären aber eine Vervollständigung der Methoden. Der dritte Vorschlag würde die vorhandenen Methoden etwas abwandeln, denn man müsste das Format des Zustandes anpassen, um dort mehrere Sprünge bzw. Verläufe speichern zu können.

4.4 Fazit

Neben der allgemeinen Betrachtung der Verfahren aus Kapitel 3, wurde das in Abschnitt 2.3 vorgestellte Fehlermodell um Aspekte bzgl. Positionierung im BB, Relation von BB und Interfunktionssprünge erweitert. Darauf baute der Entwurf des konfigurierbaren Verfahrens auf, das aus den vier Methoden No-Jump, Shared-Begin, Shared-End und Anti-Alias besteht. Im nächsten Kapitel wird beschrieben, wie dieses Verfahren implementiert wurde.

5 Implementierung

In diesem Kapitel wird beschrieben, wie der Entwurf aus Kapitel 4 als LLVM Pass umgesetzt wurde.

5.1 LLVM Passes

Nachdem das LLVM Frontend den Quelltext in die IR übersetzt hat, wird das Tool *opt* aufgerufen, welches für die Ausführung der Passes verantwortlich ist. Der *Pass-Manager* im Speziellen sorgt für die korrekte Reihenfolge und Anwendung aller Passes bei der Ausführung. Neben den Passes die LLVM mitbringt, ist es möglich, zur Laufzeit eine dynamische Bibliothek einzubinden, die zusätzliche Passes enthält. Auf diese Weise wurde der Pass entwickelt.

Um das Testen und Anwenden der entworfenen Methoden mit C/C++-Programmen zu erleichtern, wurde außerdem ein *Clang*-Wrapper erstellt, der als drop-in Ersatz für *Clang* benutzt werden kann. Dieser sorgt dafür, dass beim Übersetzen die angegebenen Methoden angewendet werden. Die Funktion, die im Fehlerfall aufgerufen wird, fügt der Wrapper dann beim Linken hinzu.

Die Entwicklung fand unter LLVM Version 9 statt. Es wurde der neue Pass-Manager verwendet und ein Wrapper als Interface für den alten PM bereitgestellt. Das war erforderlich, um den Pass direkt mit *Clang* aufrufen zu können.

5.2 Struktur

Das Programm analysiert in einem ersten Schritt alle Informationen zu den BB, die für die ausgewählten Methoden benötigt werden. Die vier Methoden werden dann gemeinsam nach Platzierungsreihenfolge im BB auf die Blöcke angewendet.

In Abbildung 5.1 ist dargestellt, in welcher Reihenfolge die Instruktionen der Methoden eingefügt werden. Da sich die Funktionalität der Methoden gegenseitig nicht beeinflusst, ist für die Abfolge nur entscheidend, dass das Setzen und Prüfen in der vorgegebenen Reihenfolge stattfindet. Die Instruktionen zur Prüfung des Zustandes und damit der Fehlererkennung sind durch `check` dargestellt. In der Implementierung entspricht das mehreren Instruktionen und einer *branch*-Instruktion für den Fehlerfall. Mit `store` ist das Speichern der ID im Zustand gemeint.

```
[phi instructions]
 check Shared-Begin
 store No-Jump
[original instructions]
 check No-Jump
 check Shared-End
 store Shared
 check Anti-Alias
 store Anti-Alias
```

Abbildung 5.1: Struktur eines Basisblocks nach Anwendung aller Methoden

```
1 int Increment(int x) {
2   int y = x+1;
3   return y;
4 }
```

Listing 5.1: Inkrement-Funktion in C

5.3 Zwischensprache

Die IR teilt Funktionen ebenfalls in BB ein. Die Verfahren gehen so vor, dass sie für jede mögliche Feststellung eines Kontrollflussfehlers einen neuen BB einführen, der die Sprunginstruktion (Prüfung am Ende) oder den BB bis auf die Prüfinstruktionen (Prüfung am Anfang) enthält. Wie in Abbildung 5.1 dargestellt, muss beim Einfügen neuer Instruktionen beachtet werden, dass *PHI*-Instruktionen am Anfang eines BB erhalten bleiben. Diese formale Voraussetzung hängt mit der *static single assignment form* der IR zusammen.

Falls ein Fehler auftritt, wird an einen neuen BB am Ende der Funktion gesprungen, der eine Fehlerbehandlungsfunktion aufruft. Diese Funktion beendet die Ausführung und kann zusätzlich andere Maßnahmen ergreifen.

Listing 5.1 zeigt eine sehr kurze C-Funktion, die nur aus einem BB (`entry`) besteht und den übergebenen Wert inkrementiert. Auf diese Funktion soll auf IR-Ebene die Methode No-Jump angewandt werden, um zu zeigen, welche Schritte für die Implementierung notwendig sind. Dazu ist in Listing 5.2 eine Umsetzung der Funktion in der IR gezeigt.

In Listing 5.3 ist das Ergebnis nach Anwendung der Methode No-Jump dargestellt. Es wurde der Block `cfe` hinzugefügt, der die Funktion `ErrorHandler` aufruft. Pro Funktion wird dieser Block einmal hinzugefügt und bei einem erkannten Fehler dorthin gesprungen. Die letzte Instruktion eines BB wird in einen eigenen BB verschoben (Zeile 10), da sie selbst eine Sprunginstruktion oder terminierende Instruktion ist und so erhalten bleiben kann. In dem Eingangsblock jeder Funktion wird auf dem Stack eine Zustandsvariable für jede Methode angelegt (Zeile 3). Für die Speicherung der Signatur

```
1 define i32 @Increment(i32 %0) {
2 entry:
3   %1 = add nsw i32 %0, 1
4   ret i32 %1
5 }
```

Listing 5.2: Inkrement-Funktion in der LLVM Zwischensprache

```
1  define i32 @Increment(i32 %0) {
2  entry:
3    %S = alloca i32
4    store volatile i32 1, i32* %S
5    %1 = add nsw i32 %0, 1
6    %2 = load volatile i32, i32* %S
7    %3 = icmp ne i32 %2, 1
8    br i1 %3, label %cfe, label %4
9  4:
10   ret i32 %1
11 cfe:
12   call void @ErrorHandler()
13   unreachable
14 }
```

Listing 5.3: Inkrement-Funktion nach Anwendung der Methode No-Jump

wird nur eine `store` Anweisung benötigt (Zeile 4). Zur Prüfung der Zustandsvariablen wird der Wert vom Stack geladen (Zeile 6), dieser mit dem Sollwert verglichen (Zeile 7) und im Fehlerfall nach `cfe` gesprungen (Zeile 8).

Die Implementierung der Methode Shared benötigt keine neuen Anweisungen. Für die Methode Anti-Alias wird zusätzlich eine Instruktion für die Division benötigt. Hier wird dafür der Befehl `urem` verwendet.

5.4 Implementierung

Die Implementierung erfolgte im Wesentlichen in C++ und der Kern umfasst ca. 500 Quellcodezeilen.

Der neue *Pass Manager* bietet ein Interface für einen *Function Pass*, dessen Methode für jede Funktion des Programms aufgerufen wird. Zusätzlich zu diesem Interface wurde für den *Legacy Pass Manager*, der für Clang benötigt wird, die Klasse *Module Pass* gewählt. Das ist nötig, da ein *Function Pass* dort nicht das Modul modifizieren darf, was für die hinzugefügte Funktion `ErrorHandler` nötig ist.

Das Anwenden der Methoden findet in zwei Schritten statt. Zuerst werden die nötigen Informationen zu den BB ermittelt und IDs zugewiesen. In Listing 5.4 ist dargestellt, welche Informationen benötigt werden.

```
1  // properties of a basic block for cfc
2  struct BBInfo {
3    // is it the entry block for the function?
4    bool IsEntry = false;
5    // is it a block which returns from the function?
6    bool IsExit = false;
7    // does the block consist of only one instruction?
8    bool IsSingleInstruction = false;
9    // unique for every block (per function)
10   uint32_t UniqueID = 0;
11   // product of the predecessor's UniqueID
12   uint32_t UniqueIDProduct = 0;
13   // shared between predecessors
14   uint32_t SharedID = 0;
15   // ID of the predecessors
16   uint32_t PreviousSharedID = 0;
17 };
18 // holds necessary information on all blocks
19 typedef ValueMap<BasicBlock *, BBInfo> BBProperties;
```

Listing 5.4: Datenstruktur, die Informationen über die BB einer Funktion enthält

Im zweiten Schritt werden mithilfe des *IR-Builder* die nötigen Instruktionen hinzugefügt. Dabei werden Hilfsfunktionen von LLVM benutzt, die die Integrität der vorhandenen Instruktionen erhalten.

5.5 Benutzung

Um Projekte, statt mit *Clang*[1], mit dem *Clang*-Wrapper zu übersetzen, muss lediglich der Aufruf **clang/clang++** durch den Aufruf eines symbolischen Links (**clang-cfc** / **clang-cfc++**), der auf das Skript verweist, ersetzt werden. Durch Setzen der Umgebungsvariablen **CFC_METHODS=<Methoden>** auf die anzuwendenden Methoden, kann dann der *Pass* aktiviert werden. Sonst wird der Aufruf unverändert an *Clang* weitergegeben.

Die implementierten Methoden können natürlich auch mit *opt* auf beliebige IR-Dateien angewendet werden. Dazu muss *opt* nur mit den Parametern **-load-pass-plugin=<LIB>** und **-passes=cfc-<Methoden>** aufgerufen werden. Anschließend kann (z. B. mit *llc*) Assembler-Code für die Zielplattform erzeugt werden.

[1] Aufgrund der Kommandozeilenkompatibilität zwischen *Clang* und *gcc*, ist hier auch die Ersetzung des *gcc* denkbar.

5.6 Einschränkungen

Die Benutzung von *Exceptions* in der IR ist nicht trivial und würde den Implementierungsaufwand erheblich erhöhen. Deshalb wurde dieser Aspekt hier außen vor gelassen und bei Ausführung des Pass werden alle Funktionen die *Exceptions* enthalten übersprungen.

Viele Verfahren sehen ein eigenes Register für die Speicherung der Zustandsvariablen vor. Das kann auf IR-Ebene nicht umgesetzt werden, weil diese in *static single assignment form* definiert ist und kein Wissen über die tatsächlichen Register der Zielmaschine enthält. Somit wird bei dieser Implementierung eine Lösung auf dem Stack gewählt.

Hinzu kommt, dass bei einer Übersetzung der IR nach Assembler mit *llc* die neu eingefügten Instruktionen entfernt werden, weil sie aus Sicht der fehlerfreien Ausführung keinen Effekt haben. Da es naheliegend ist, dass die Instruktionen durch automatisierte Optimierungen nicht in die Zielsprache übersetzt werden, wurden die Variablen als `volatile` gekennzeichnet und so eine mögliche Werteänderung außerhalb des Programmeinflusses vorgesehen.

5.7 Beachtenswertes

Beim Debugging von Programmen ist es üblich, Debugsymbole in die Datei einzufügen. Beim Anwenden der Verfahren bewirkt das, dass das Programm auf Quelltextebene wie gewohnt debuggt werden kann. Um die CFM Teile des Programms zu debuggen, wurde in dieser Arbeit das Programm erst nach Assembler (inkl. Anwendung der Verfahren) übersetzt und anschließend bei der Übersetzung zur Binärdatei die Debuginformationen hinzugefügt. Dadurch kann das Debugging mit den von *llc* erstellten Assembler-Dateien durchgeführt werden, die zusätzliche Annotationen der IR beinhalten.

5.8 Fazit

Der LLVM Pass wurde so implementiert, dass er auf BB-Ebene über jede Funktion iteriert. Nach einem Analyseschritt werden jedem BB die nötigen Instruktionen hinzugefügt. Durch die Architektur der IR musste eine Variable für den Zustand auf dem Stack angelegt werden, statt dafür Register vorzusehen. Im nächsten Kapitel wird nun die Funktionalität und Leistungsfähigkeit unter Anwendung des Clang-Wrappers getestet.

6 Evaluation

In diesem Kapitel geht es darum, das Verhalten des implementierten Pass zu untersuchen. Zuerst wird durch Anwendung auf etablierte Software ohne die Injektion von Hardwarefehlern das Verhalten unter normalen Umständen getestet. Anschließend wird das Verhalten unter Fehlerinjektion betrachtet.

6.1 Fehlerfreie Ausführung

Um sicher zu stellen, dass durch die Anwendung der Methoden die Funktionalität von Software nicht beeinträchtigt ist, wurden diese mithilfe des beschriebenen Clang-Wrappers auf verschiedene C/C++ Programme angewendet.

Die *GNU core utilities*[1] sind eine Sammlung von Programmen, die grundlegende Funktionen von Unixoiden Betriebssystemen bereitstellen. Darunter fallen Programme zum Arbeiten mit Dateien, der Shell und Textmanipulation. Die Programme sind in C geschrieben und sind mit ausführlichen Tests ausgestattet. Die Programme wurden mit verschiedenen Methodenkombinationen kompiliert und getestet. Das Ausführen der Tests und die stichprobenartigen Funktionsprüfungen der Programme ergaben keine Auffälligkeiten.

Die Benchmark Suite *MiBench* [6] beinhaltet einige Benchmarks in C. Diese decken verschiedene Einsatzgebiete ab und wurden für eingebettete Systeme entwickelt. Beim Anwenden des Pass und Ausführen der Benchmarks konnten ebenfalls keine Auffälligkeiten festgestellt werden.

Die grafische Textverarbeitung *LyX*[2] ist in C++ programmiert. Leider werden hier *Exceptions* verwendet, weshalb nicht jede Funktion getestet werden konnte (siehe Abschnitt 5.6). Die mitgelieferten Tests ergaben keine Auffälligkeiten für die ersten drei Methoden (No-Jump, Shared-Begin und Shared-End). Einzelne Tests haben eine fälschliche Auslösung von Anti-Alias gezeigt, dessen Grund noch nicht abschließend ermittelt wurde. Es scheint sich hier aber um ein seltenes Programmiermuster zu handeln, da es an anderen Stellen keine Auffälligkeiten gab. Eine oberflächliche Benutzung der Software (ausgenommen Anti-Alias) ergab keine Probleme.

[1] `https://www.gnu.org/software/coreutils/`
[2] `https://www.lyx.org/`

6.2 Fehlerinjektion

Das Fehlerinjektionswerkzeug FAIL* bietet verschiedene Möglichkeiten, gezielt simulierte Hardwarefehler in Programme zu injizieren. Ein Experiment stellt die Simulation der Ausführung des Programms dar und beinhaltet die Injektion eines Fehlers an einer Stelle. Das kann theoretisch für alle möglichen Fehlerinjektionen wiederholt werden oder eine geringere Zahl an Experimenten wird über den Versuchsraum verteilt.

In den folgenden Untersuchungen wurde hauptsächlich die Möglichkeit zur Injektion von CFE genutzt. Das bedeutet, dass während der Ausführung einmalig der Wert des *Instruction Pointers* manipuliert wird, um die Ausführung an anderer Stelle fortzusetzen. Dazu geschieht das Importieren in die Datenbank mit dem *RandomJumpImporter*, der prinzipiell alle möglichen Injektionen von CFE ab dem Start des Benchmarks betrachtet.

Desweiteren wurde die Möglichkeit der Injektion von Bitfehlern in die Register während der Ausführung unter Verwendung des *RegisterImporter* genutzt.

6.2.1 Benchmarks

Als Testsoftware für FAIL* wurde *eCos*[3] mit den Benchmarks *bitcount*, *dijkstra* und *qsort* aus dem oben genannten *MiBench* benutzt. Das *eCos*-Betriebssystem ist ein sehr kleines eingebettetes System. Es wird benutzt, um eine Umgebung zur Ausführung der Benchmarks in FAIL* zu schaffen. Die Benchmarks wurden mit dem angepassten Clang-Wrapper übersetzt und daraufhin mit *eCos* gelinkt. Die Benchmarks wurden ausgewählt, weil sie im Vergleich zu den anderen eine relativ geringe Laufzeit aufweisen. Weiterhin wurde die Eingabemenge der Benchmarks im Vergleich zur ursprünglichen Version ungefähr um den Faktor 50 reduziert, um die Datenmenge möglichst gering zu halten.

6.2.2 Reduktion des Versuchsraums

Beim Importieren der Daten des *Golden Run* mit dem *RandomJumpImporter* kann dort die Zahl der Versuche bereits eingeschränkt werden. Das ist sinnvoll, da sonst die Menge der Experimente aus dem Produkt der dynamischen Instruktionen (ausgeführte Instruktion zu einem Zeitpunkt) und den statischen Instruktionen des Programms besteht.

Mithilfe einer *MemoryMap* können die betrachteten Instruktionen auf bestimmte Adressräume eingeschränkt werden. Diese Möglichkeit wurde hier genutzt, um den Bereich von Sprungziel und -quelle auf die ursprünglichen Funktionen des Benchmarks einzuschränken.

Bei vielen Programmen werden einige Bereiche des Codes nur einmalig oder wenige Male ausgeführt, bestimmte Schleifen o. ä. dagegen sehr häufig. Um den Versuchsraum

[3] `https://ecos.sourceware.org/`

etwas zu Gunsten weniger häufig ausgeführter Instruktionen auszulegen und die Gesamtmenge an Experimenten weiter zu reduzieren, wurde in FAIL* die Option *jump-max* eingebaut. Damit kann die maximale Anzahl an Ausführungen einer Instruktion angegeben werden, bis zu der diese als Quelle für einen Sprung benutzt wird. Für die folgenden Tests wurde dieser Wert auf 10 festgelegt.

6.2.3 Sprunginjektion

In einem ersten Experiment wurden bei den drei Benchmarks je 1000 Kontrollflussfehler-Injektionen verteilt über die Programmausführung durchgeführt. Das wude für alle 16 möglichen Kombinationen der vier vorgestellten Methoden vorgenommen.

Metrik Um die Vergleichbarkeit zwischen den verschiedenen Methodenkombinationen sicherzustellen, werden zur Extrapolation der Ergebnisse die Anzahl der Vorkommen mit der Gesamtdauer der Ausführung multipliziert. Das liegt daran, dass alle möglichen Vorkommen eines CFE an einer bestimmten dynamischen Instruktion liegen müssen und diese Berechnungsvorschrift so den gesamten Fehlerraum beachtet.

Es ist zusätzlich zu beachten, dass hier davon ausgegangen wird, dass an jeder Stelle mit gleicher Wahrscheinlichkeit ein CFE auftritt. In der Realität könnte diese Vermutung nicht ganz korrekt sein. Es wäre vorstellbar, dass, wenn die Ausführung auf eine Sprunginstruktion trifft, dort zusätzlich durch Veränderung des Sprungziels ein CFE auftreten kann, was die Wahrscheinlichkeit ungleichmäßig über die möglichen Injektionspunkte verteilen würde. Aufgrund mangelnder Datengrundlage, lassen sich hier aber keine weiteren Schlüsse ableiten.

Ergebnis In den Abbildungen 6.1 bis 6.6 sind die berechneten Ergebnisse der drei Benchmarks dargestellt. Dabei haben die Ausgänge einer Injektion folgende Bedeutung:

OK_MARKER Das Ergebnis des Programms wurde nicht beeinträchtigt

TIMEOUT Die Ausführung wurde nach der dreifachen Programmdauer abgebrochen

TRAP Durch einen unerwarteten Fehler wurde eine Unterbrechung ausgelöst

SDC Die Ausgabe des Programms ist inkorrekt, ist aber normal beendet

DETECTED_MARKER Die Funktion als Signal für die Erkennung eines CFE wurde aufgerufen

Anhand der Abbildung mit allen Ergebnistypen kann man die Laufzeit der Methoden vergleichen. Durch die verwendete Metrik werden die 1000 Ergebnisse mit der Laufzeit multipliziert. Bei Betrachtung der Gesamthöhe der Balken, kann somit die Laufzeit abgelesen werden. Die beiden Benchmarks *bitcount* und *qsort* liegen beide mit der

Laufzeit in derselben Größenordnung. Hingegen hat *bitcount* ungefähr eine um Faktor 100 kürzere Laufzeit.

Die Variation zwischen den Methoden ist bei *dijkstra* sehr gering, was auf eine große Zahl an Instruktionen pro BB schließen lässt, da sich durch die Anwendung der Methoden die Laufzeit kaum verändert. Das gegenteilige Bild liefert *bitcount*, wo sich bei Anwendung aller Methoden die Laufzeit mehr als verzehnfacht. Dort kann man gut sehen, wie in ungünstigen Konstellationen Verfahren mit vielen Instruktionen eine wesentliche Auswirkung auf die Laufzeit haben können.

Bei der Kombination aller Methoden lässt sich auch im Diagramm der SDC bei *qsort* und *bitcount* eine deutliche Spitze ausmachen. Durch den hohen Anteil an Prüfinstruktionen überwiegen beim Hinzufügen weiterer Methoden also hauptsächlich die Effekte längerer Laufzeit, im Gegensatz zu erhöhter Fehlerabdeckung. Das wird insbesondere an den Methoden Shared-End und Anti-Aliasing deutlich. Eine der beiden Methoden in Kombination mit Shared-Begin und No-Jump verbessert die Fehlerrate. Eine gemeinsame Verwendung führt aber zu der beschriebenen Verschlechterung, was die schlechte Kombinierbarkeit (siehe Abschnitt 4.3.4) bestätigt.

Abgesehen von einer Ausprägung bei einer Kombination von vielen Methoden bei *qsort*, stimmen die relativen Anteile von SDC gut mit *dijkstra* überein. Danach schneidet eine Kombination aus Shared-Begin und Anti-Alias mit evtl. No-Jump am besten ab. Das geht eigentlich gegen die Intuition, dass die Kombination von Shared-Begin und Anti-Alias keine Vorteile gegenüber den einzelnen Methoden bietet, weil Anti-Alias lediglich leicht von Shared-Begin abgewandelt ist. Die niedrige Fehlerrate könnte sich aber durch eine hohe Zahl an Interfunktionssprüngen in der Fehlerinjektion erklären lassen, wo durch die unterschiedlichen IDs die Wahrscheinlichkeit steigt, dass zumindest eines der beiden Verfahren einen Fehler erkennt.

Allgemein schneidet bei den Benchmarks *dijkstra* und *qsort* die Kombination aus No-Jump, Shared-Begin und Anti-Alias mit einer verbesserten SDC-Rate von 65% (*qsort*) und 89% (*dijkstra*) am besten ab. Bei *bitcount* ist fast nie eine Verbesserung gegenüber der Version ohne CFM zu sehen. Hier schneidet die Kombination von Shared-Begin und Shared-End mit 8% weniger SDC am besten ab.

6.2.4 Registerinjektion

Um ein allgemeineres Fehlermodell zu testen, wurde für *bitcount* ebenfalls eine Fehlerinjektion im Register getestet. Es wurden wieder 1000 Injektionen für jede Kombination durchgeführt. Die Auswertung ist ähnlich, mit der einzigen Änderung, dass die Berechnung der Skalierung des Ergebnisses zusätzlich mit der Menge an möglichen Injektionspunkten in den Registern multipliziert [13] wird.

Bei *bitcount* schneiden die meisten Verfahren schlechter ab, was zu der Vermutung passt, dass hier auch andere Fehlerarten als CFE eine Rolle spielen. Dennoch schaffen es die besten Kombinationen, eine niedrigere Anzahl an SDC als im Fall ohne Anwendung

der Methoden zu erreichen. Hier liegt die Kombination Shared-Begin mit Anti-Alias und 34% weniger SDC vorne.

Bei *dijkstra* ergibt sich das eigenartige Bild, dass alle Methoden bis auf Shared-Begin viel besser als in der CFE-Injektion abschneiden. Das passt zu der Vermutung, dass Shared-Begin aufgrund der Distanz zwischen Prüf- und Setzinstruktion schlecht für lange BB geeignet ist. Hier schneiden alle anderen Methoden mit nahezu keinen SDC Vorkommen sehr gut ab.

Bei *qsort* bietet sich ein ähnliches Bild wie bei *bitcount*. Hier sind allerdings die Variationen zwischen den Methoden geringer. Am besten schneidet die Kombination aus allen Methoden mit 29% weniger SDC ab.

6.3 Fazit

In diesem Kapitel wurde der modulare Pass auf Eignung zum Einsatz als CFM-Verfahren getestet. Die verschiedenen Methoden konnten anhand von Fehlerinjektionsexperimenten verglichen werden, was ebenfalls einige Eigenschaften im Bezug auf die Kategorisierung verdeutlicht hat.

Im nächsten Kapitel wird ein abschließender Überblick zu der gesamten Arbeit gegeben.

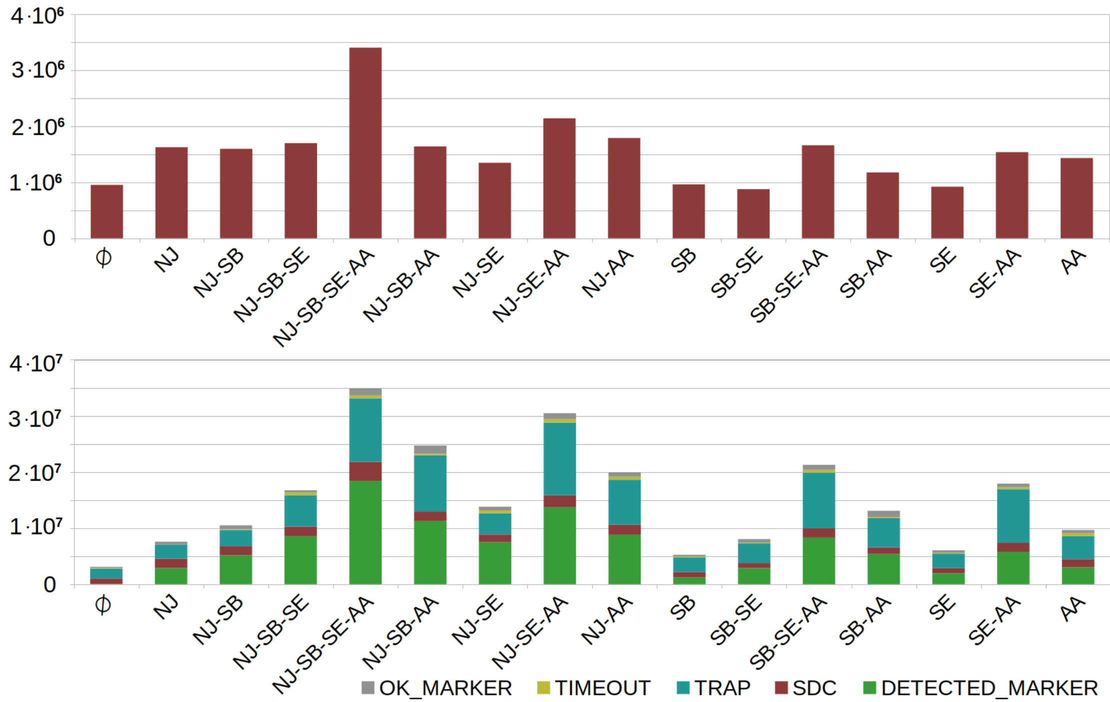


Abbildung 6.1: Ergebnisse der Sprunginjektion mit *bitcount*

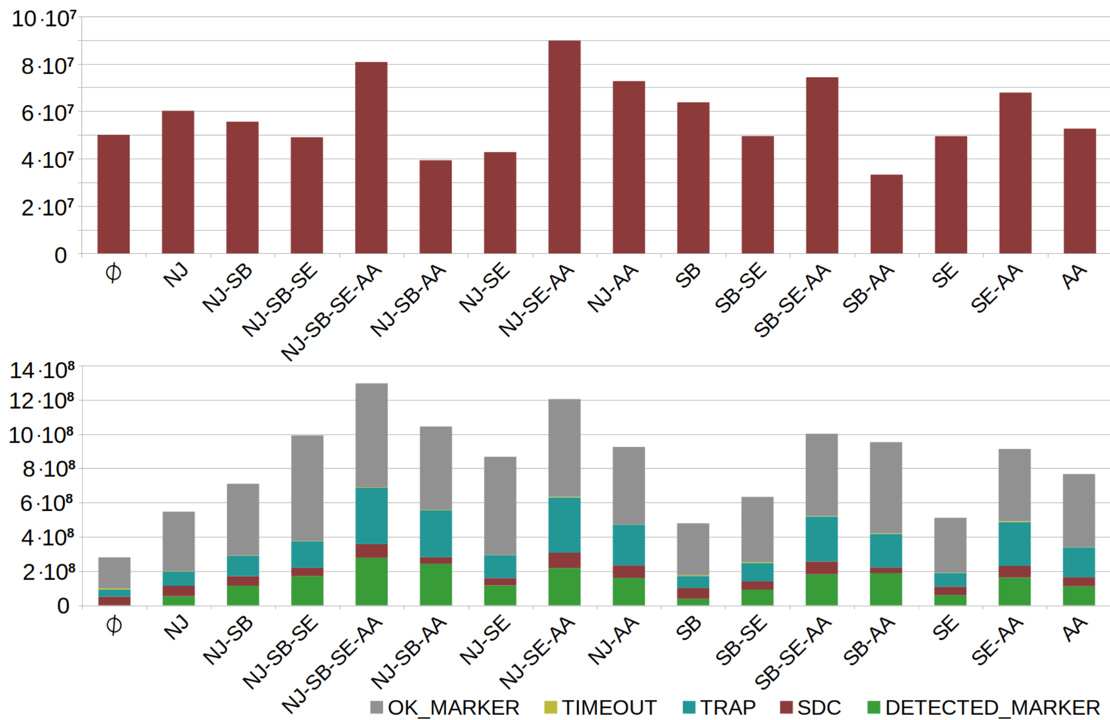


Abbildung 6.2: Ergebnisse der Registerinjektion mit *bitcount*

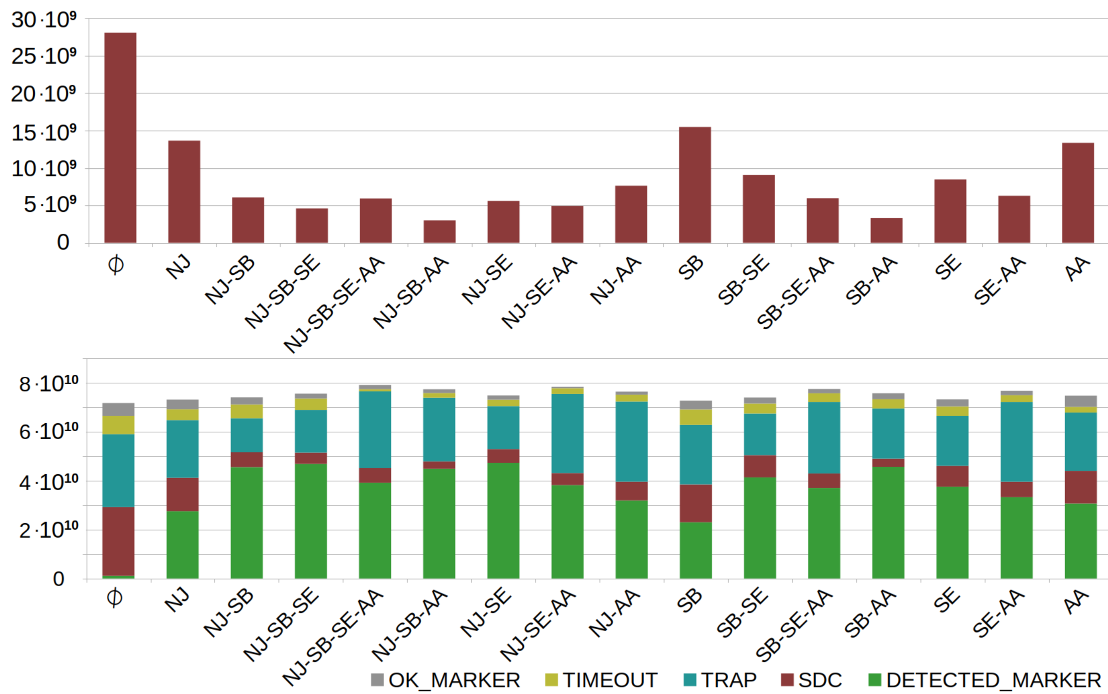


Abbildung 6.3: Ergebnisse der Sprunginjektion mit *dijkstra*

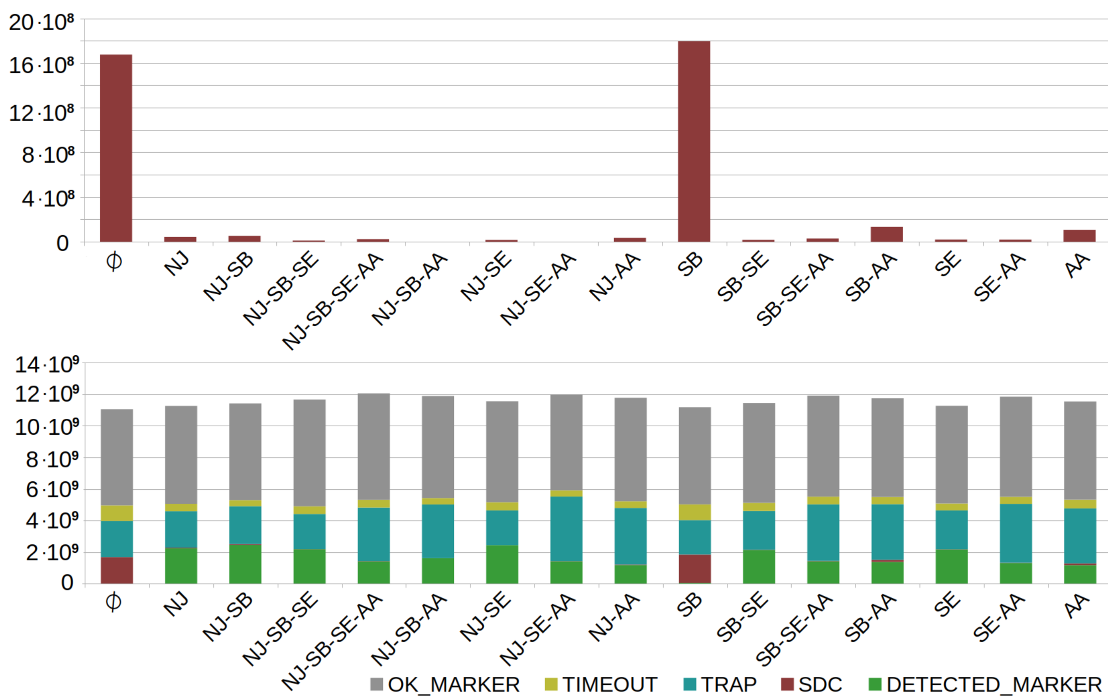


Abbildung 6.4: Ergebnisse der Registerinjektion mit *dijkstra*

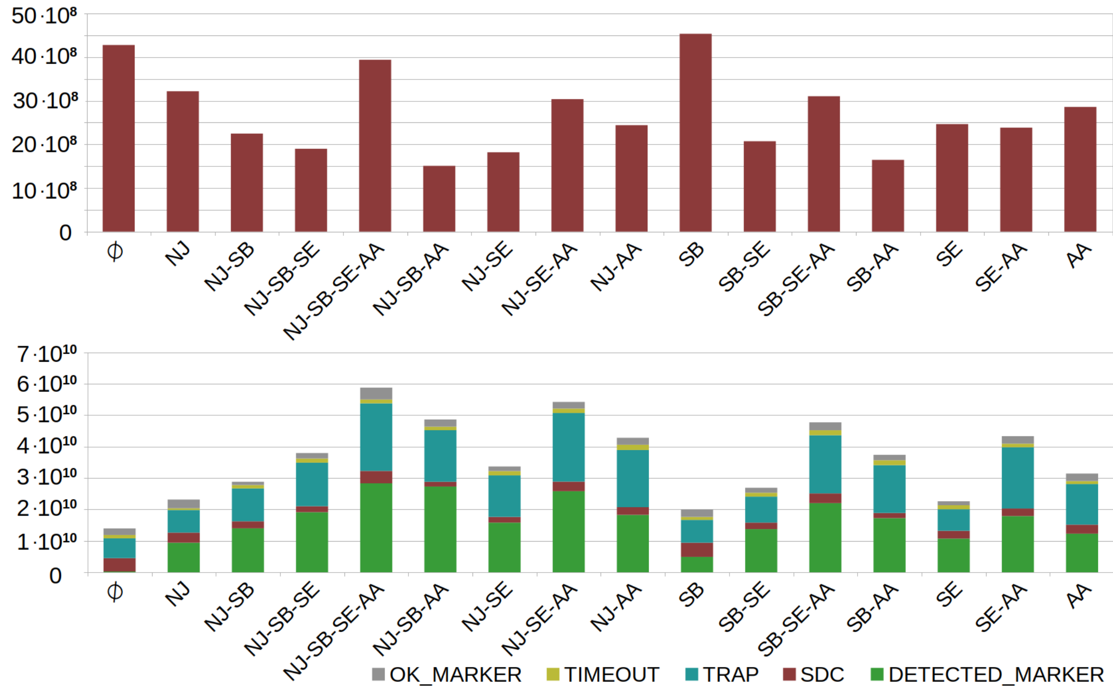


Abbildung 6.5: Ergebnisse der Sprunginjektion mit *qsort*

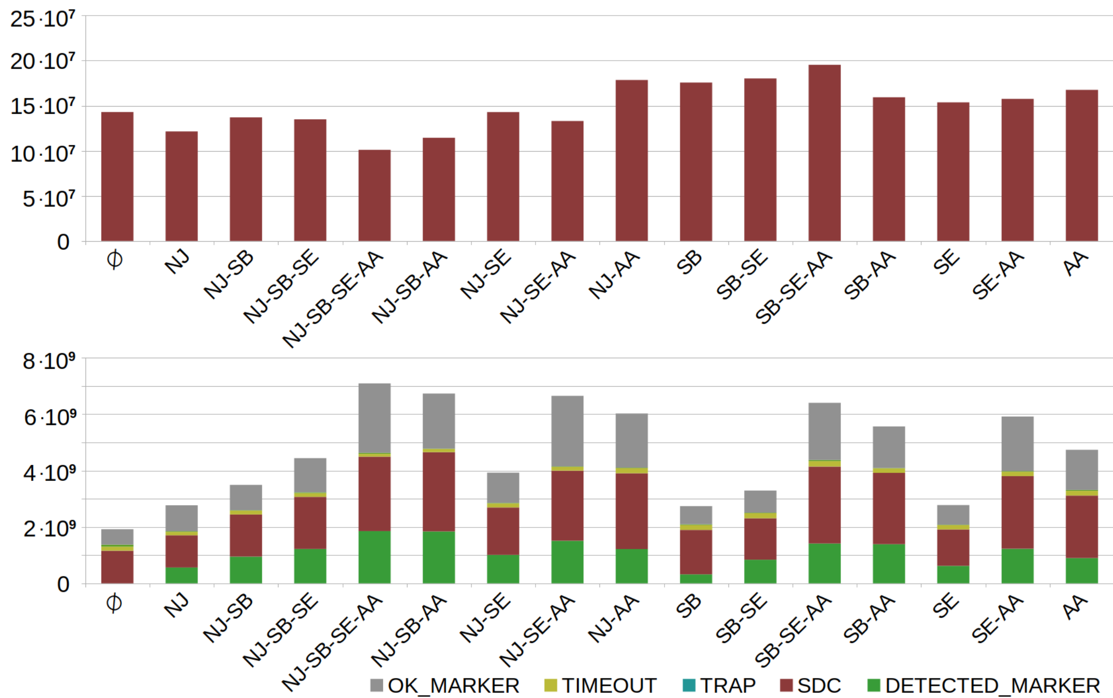


Abbildung 6.6: Ergebnisse der Registerinjektion mit *qsort*

7 Zusammenfassung

Dieses Kapitel schließt die Arbeit ab. Der Inhalt wird zusammengefasst und es werden mögliche Fortsetzungen betrachtet.

7.1 Resümee

Es wurden zunächst eine Reihe von CFM-Verfahren aus der Literatur betrachtet. Daran konnte die unter Kapitel 4 dargestellte Verfeinerung und Erweiterung des Fehlermodells von Goloubeva et al. [5] erklärt werden. Es wurden Dimensionen zur Position im BB, Relation der BB und Sprünge über Funktionsgrenzen hinweg betrachtet.

Aus der Kategorisierung und den Verfahren wurde ein modulares Verfahren entworfen, das aus den drei Komponenten No-Jump, Shared und Anti-Alias besteht. Zur Anwendung wurde ein LLVM Pass, der diese Methoden umsetzt, sowie ein Clang-Wrapper für C/C++ Programme entworfen. Das hat auch Schwächen der Nutzung von LLVM für CFM aufgezeigt. So ist es nicht möglich, ein eigenes Register für die Zustandsvariable vorzusehen und die Verfahren müssen mit dem Stack interagieren, was die Laufzeit in der Realität erheblich beeinflusst.

Durch die Implementierung konnten einfache Anwendungen auf Programme die Beeinträchtigung der Software nahezu ausschließen. Schließlich haben Fehlerinjektionsexprimente mit FAIL* und drei Benchmarks aus *MiBench* die Leistungsfähigkeit und Eigenschaften der Methoden gezeigt. Das Testen aller möglichen 16 Kombinationen ergab für alle Testfälle mindestens eine Kombination, die eine Verbesserung der CFE-Anfälligkeit erreicht hat. Besonders gut hat bei der CFE-Injektion die Kombination von No-Jump, Shared-Begin und Anti-Alias abgeschnitten, bei der mit dem *dijkstra* Benchmark eine Verbesserung von 89% weniger SDC Fällen zu beobachten war. Die Register-Injektion ergab ein gemischtes Bild, hat aber generell auch gute Methodenkombinationen hervorgebracht.

Insgesamt sieht man an den Ergebnissen der Evaluation, dass sich ein konfigurierbares Verfahren durchaus für verschiedene Programme mit unterschiedlichen Eigenschaften eignet, um eine gute Leistungsfähigkeit durch die richtige Wahl der Methodenkombination zu erreichen.

7.2 Ausblick

Konzeptuell konnte die Idee eines konfigurierbaren Pass gut umgesetzt werden. Die genannten Nachteile der Nutzung von LLVM lassen sich nicht leicht beheben. Hier stellt sich also auch die Frage nach möglichen Alternativen zur Implementierung.

Mit den in Abschnitt 4.3.5 beschriebenen Ergänzungen kann man den Pass leicht zur Abdeckung weiterer Fälle ergänzen und so den Umfang der Möglichkeiten erhöhen. Auch die schon verwendeten Methoden benutzen zur simpleren Betrachtung nicht alle Optimierungen, die bei den in Kapitel 3 vorgestellten Verfahren angeblich für eine Abdeckung von mehr Fehlern führen. Eine Anpassung der Methoden wäre also denkbar.

Schließlich lassen weitere Untersuchungen im Bezug auf das gezeigte Fehlermodell, wie eine Unterscheidung nach Inter- und Intra-Funktionssprüngen und eventuelle tiefere Analysen der Ergebnisse der Benchmarks, die sich auf die recht verschiedenen Ergebnisse beziehen, sowie das Testen von mehr Programmen auf weitere Erkenntnisse zur Erstellung von CFM-Verfahren hoffen.

Literaturverzeichnis

[1] Z. Alkhalifa u. a. „Design and evaluation of system-level checks for on-line control flow error detection“. In: *IEEE Transactions on Parallel and Distributed Systems* 10.6 (Juni 1999), S. 627–641. DOI: 10.1109/71.774911.

[2] A. Benso u. a. „Control-flow checking via regular expressions“. In: *Proceedings 10th Asian Test Symposium.* Nov. 2001, S. 299–303. DOI: 10.1109/ATS.2001.990300.

[3] O. Goloubeva u. a. „Improved software-based processor control-flow errors detection technique“. In: *Annual Reliability and Maintainability Symposium, 2005. Proceedings.* Jan. 2005, S. 583–589. DOI: 10.1109/RAMS.2005.1408426.

[4] O. Goloubeva u. a. „Soft-error detection using control flow assertions“. In: *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems.* Nov. 2003, S. 581–588. DOI: 10.1109/DFTVS.2003.1250158.

[5] O. Goloubeva u. a. *Software-Implemented Hardware Fault Tolerance.* Springer-Verlag, 2006. ISBN: 0387260609. DOI: 10.1007/0-387-32937-4.

[6] M. R. Guthaus u. a. „MiBench: A free, commercially representative embedded benchmark suite“. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538).* 2001, S. 3–14. DOI: 10.1109/WWC.2001.990739.

[7] G. A. Kanawati u. a. „Evaluation of integrated system-level checks for on-line error detection“. In: *Proceedings of IEEE International Computer Performance and Dependability Symposium.* Sep. 1996, S. 292–301. DOI: 10.1109/IPDS.1996.540230.

[8] J. Karlsson, U. Gunneflo und J. Torin. „Use of Heavy-Ion Radiation from 252Californium for Fault Injection Experiments“. In: *Dependable Computing for Critical Applications.* Vienna: Springer Vienna, 1991, S. 197–212. ISBN: 978-3-7091-9123-1. DOI: 10.1007/978-3-7091-9123-1_9.

[9] C. Lattner und V. Adve. „LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation“. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization.* CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, S. 75. ISBN: 0769521029. DOI: 10.1109/CGO.2004.1281665.

[10] A. Li und B. Hong. „Software implemented transient fault detection in space computer“. In: *Aerospace Science and Technology* 11 (März 2007), S. 245–252. DOI: 10.1016/j.ast.2006.06.006.

[11] G. Miremadi u. a. „Two software techniques for on-line error detection“. In: *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing.* Juli 1992, S. 328–335. DOI: 10.1109/FTCS.1992.243568.

[12] N. Oh, P. P. Shirvani und E. J. McCluskey. „Control-flow checking by software signatures“. In: *IEEE Transactions on Reliability* 51.1 (März 2002), S. 111–122. DOI: 10.1109/24.994926.

[13] H. Schirmeier, C. Borchert und O. Spinczyk. „Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors“. In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* 2015, S. 319–330. DOI: 10.1109/DSN.2015.44.

[14] H. Schirmeier u. a. „FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance“. In: *2015 11th European Dependable Computing Conference (EDCC).* Sep. 2015, S. 245–255. DOI: 10.1109/EDCC.2015.28.

[15] R. Venkatasubramanian, J. Hayes und B. Murray. „Low-cost on-line fault detection using control flow assertions“. In: Aug. 2003, S. 137–143. DOI: 10.1109/OLT.2003.1214380.

Abbildungsverzeichnis

1.1 Kontrollflussgraph zu Abb. 1.1 und Anwendung von BSSC 2

2.1 Fehlermodell aus SIHFT [5] 7
2.2 Architektur von LLVM 8

4.1 CFG, bei dem *Aliasing* entstehen kann aus CFCSS [12] 21

5.1 Struktur eines Basisblocks nach Anwendung aller Methoden 28

6.1 Ergebnisse der Sprunginjektion mit *bitcount* 38
6.2 Ergebnisse der Registerinjektion mit *bitcount* 38
6.3 Ergebnisse der Sprunginjektion mit *dijkstra* 39
6.4 Ergebnisse der Registerinjektion mit *dijkstra* 39
6.5 Ergebnisse der Sprunginjektion mit *qsort* 40
6.6 Ergebnisse der Registerinjektion mit *qsort* 40

Listingverzeichnis

1.1 Implementierung von Bubblesort in C 2

2.1 Bubblesort in IR 9

3.1 BB ergänzt durch CFCSS als Pseudocode 12
3.2 Bubblesort ergänzt durch CFCSS 12
3.3 BB ergänzt durch BSSC als Pseudocode 13
3.4 Bubblesort ergänzt durch BSSC 13
3.5 BB ergänzt durch ECCA als Pseudocode 14
3.6 Bubblesort ergänzt durch ECCA 14
3.7 BB ergänzt durch YACCA als Pseudocode 15
3.8 Bubblesort ergänzt durch YACCA 15
3.9 BB ergänzt durch RSCFC als Pseudocode 16
3.10 Bubblesort ergänzt durch RSCFC 16
3.11 BB ergänzt durch ACFC als Pseudocode 16
3.12 Bubblesort ergänzt durch ACFC 17
3.13 BB ergänzt durch CFCRE als Pseudocode 17
3.14 Bubblesort ergänzt durch CFCRE 18

4.1 BB ergänzt durch No-Jump in Pseudocode 23
4.2 BB ergänzt durch Shared in Pseudocode 24
4.3 BB ergänzt durch Anti-Alias in Pseudocode 24

5.1 Inkrement-Funktion in C 28
5.2 Inkrement-Funktion in der LLVM Zwischensprache 29
5.3 Inkrement-Funktion nach Anwendung der Methode No-Jump 29
5.4 Datenstruktur, die Informationen über die BB einer Funktion enthält . 30