technische universität
dortmund

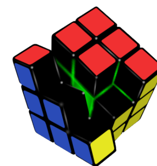
Bachelorarbeit

Entwicklung eines Werkzeugs zur Injektion von Softwarefehlern basierend auf Clang

Daniel Ferdinand Siegert
28. Mai 2018

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
M.Sc. Ulrich Thomas Gabor

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



Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Software Fault Injection . . . 3
2.2 Repräsentative Fehler . . . 3
2.3 SAFE . . . 4
2.4 Clang . . . 5
2.4.1 Clang AST . . . 6
2.5 Zusammenfassung . . . 7

3 Fehlermodell **9**
3.1 Zuweisungsfehler . . . 9
3.1.1 MVIV . . . 9
3.1.2 MVAV . . . 10
3.1.3 MVAE . . . 10
3.1.4 WVAV . . . 11
3.2 Algorithmusfehler . . . 11
3.2.1 MFC . . . 11
3.2.2 MIFS . . . 11
3.2.3 MIEB . . . 12
3.2.4 MLPA . . . 12
3.3 Überprüfungsfehler . . . 13
3.3.1 MIA . . . 13
3.3.2 MLAC . . . 13
3.3.3 MLOC . . . 14
3.4 Interfacefehler . . . 14
3.4.1 WPFV . . . 14
3.4.2 WAEP . . . 14
3.5 Zusammenfassung . . . 15

4 Entwurf **17**
4.1 Anforderungen . . . 17
4.2 Entwurfsentscheidungen . . . 17
4.2.1 Clang-Werkzeug . . . 19
4.2.2 FaultInjector . . . 20

4.2.3 Speicherung der Fehler 22
4.2.4 Konfiguration 22
4.3 Zusammenfassung 23

5 Implementierung 25
5.1 Hauptprogramm 25
5.2 Clang-Werkzeug 28
5.2.1 SFIAction 28
5.2.2 SFIASTConsumer 29
5.3 FaultInjector 29
5.3.1 Finden von Injektionsstellen 30
5.3.2 Injektion und Speichern der Patch-Files 31
5.3.3 Subklassen für das Fehlermodell 34
5.3.4 MIAInjector 34
5.3.5 MIFSInjector 35
5.3.6 MIEBInjector 36
5.3.7 MFCInjector 37
5.3.8 MVIVInjector 38
5.3.9 MVAVInjector 39
5.3.10 MVAEInjector 40
5.3.11 WVAVInjector 41
5.3.12 MLOCInjector 41
5.3.13 MLACInjector 42
5.3.14 WAEPInjector 42
5.3.15 WPFVInjector 43
5.3.16 MLPAInjector 44
5.4 Zusammenfassung 45

6 Evaluation 47
6.1 Evaluation der Ergebnisse der beiden Werkzeuge 48
6.1.1 Vergleich der Injektionsstellen 48
6.1.2 Vergleich der Injektionen 52
6.2 Fazit 55

7 Gesamtfazit und Zusammenfassung 57
7.1 Zusammenfassung 57
7.2 Fazit und Ausblick 58

8 Anhang 61
8.1 test.cpp 61

Literaturverzeichnis 65

Abbildungsverzeichnis **67**

1 Einleitung

Software wird heutzutage immer komplexer. Jedoch wird eine schnellst mögliche Implementierung von teilweise vielen komplexen Funktionen innerhalb eines Softwareprojekts bei zu beachtendem Budget gefordert. Gleichzeitig soll die zu entwickelnde Software auch fehlerfrei sein. Es ist an sich schon schwer bis unmöglich die Fehlerfreiheit eines Produkts bei der Entwicklung durch Testen zu garantieren und diese genannten Anforderungen machen dieses noch schwieriger [5, 8].

Nun kann es passieren, dass es während der Ausführung der Software zu fehlerhaften Zuständen kommt, welche beim Testen während der Entwicklung nicht betrachtet wurden. Um diesem Problem entgegenzuwirken sollte man bei der Entwicklung Fehlertoleranzmechanismen implementieren, um zur Laufzeit einen fehlerhaften Zustand zu erkennen und somit auf diesen reagieren zu können. Damit man zu diesen Aussagen über die Effektivität machen kann, muss das Verhalten der Software unter nicht vorhergesehenen fehlerhaften Bedingungen getestet werden, indem man z.B. eine fehlerhafte Bedingung injiziert. Dies nennt sich Fehlerinjektion (engl. fault injection) und ist von manchen Sicherheitsstandards, wie z.B. ISO 26262 für Sicherheit in Autos, empfohlen [8].

Für Fehlerinjektion gibt es verschiedene Ansätze: Angefangen beim Injizieren von physischen Fehlern, um Hardwarefehler zu emulieren, über die Emulation von Hardwarefehlern durch Software zur Laufzeit, bis hin zu komplexeren Fehlern durch Injektion von Fehlern in den Quelltext [8]. Für letzteres existieren die `G-SWFIT`-Technik [5], mit welcher in den binären bereits kompilierten Programmcode injiziert wird, und das Werkzeug `SAFE` [8], welches in den Quelltext eines Programmes Fehler injizieren kann. Durch die Injektion in den binären Code können zwar Fehler injiziert werden, ohne den Quelltext des Programmes zu haben, es kommt aber häufig zu Ungenauigkeiten im Gegensatz zur Injektion in den Quelltext [3].

Bisher gibt es für diese Injektion in den Quelltext so gut wie keine Werkzeuge. `SAFE` [8] ist ein Werkzeug, welches Softwarefehler in Quelltext injiziert und dabei dasselbe Fehlermodell wie in `G-SWFIT` [5] verwendet. Allerdings ist `SAFE` nicht frei verfügbar, es muss eine alte Standardbibliothek verwendet werden und es kann nicht in aktuellen C++-Code injizieren[1]. Außerdem ist der Quelltext nicht öffentlich und es ist nicht möglich das Fehlermodell gegebenenfalls anzupassen.

Dementsprechend ist die Zielsetzung dieser Arbeit der Entwurf und die Im-

[1] Auf `SAFE` wird in den Grundlagen und bei der Evaluation nochmals genauer eingegangen.

plementierung eines Werkzeugs für `SFI`. Dieses soll die folgenden Anforderungen erfüllen:

- Das Werkzeug soll quelloffen sein.
- Es soll in aktuellen C++-Code mit einer aktuellen Standardbibliothek injiziert werden können.
- Das Fehlermodell soll erweiterbar und konfigurierbar sein.
- Das Werkzeug soll Patch-Files für die spätere Anwendung der injizierten Fehler erstellen.
- Es soll eine Möglichkeit geben, diese Patch-Files anzuwenden, das Projekt, in das injiziert wird, zu kompilieren und Tests auszuführen.

Aus dieser Zielsetzung folgt der weitere Aufbau dieser Arbeit in

1. die Grundlagen, die für das weitere Verständnis dieser Arbeit benötigt werden,
2. das Fehlermodell, welches dem zu entwerfenden Werkzeug zugrunde liegen soll,
3. den Entwurf des Werkzeugs `clang-sfi`,
4. dessen Implementierung und
5. einer Evaluation des Werkzeugs bei dem es unter anderem mit `SAFE` verglichen wird.

2 Grundlagen

In diesem Kapitel werden die Grundlagen für die weitere Arbeit erläutert. Darunter vor allem das Werkzeug SAFE, welches im Bereich der Softwarefehlerinjektion bereits existiert und verwendet werden kann und das Frontend Clang, welches in dem in dieser Ausarbeitung zu entwerfenden Werkzeug verwendet wird.

2.1 Software Fault Injection

Bei `Software Fault Injection (SFI)` werden kleine Veränderungen im Quelltext von Software durchgeführt um damit die Effekte von Bugs zu emulieren [8]. Die injizierten Fehler sollen dabei im Idealfall Bugs entsprechen, welche vorher beim Testen nicht aufgefallen wären [5]. Deswegen ist es hier wichtig ein dafür repräsentatives Fehlermodell zu verwenden [5]. Im Gegensatz zur `Software Implemented Fault Injection (SWIFI)` bei der mit Hilfe von Werkzeugen zur Laufzeit einer Software einfache Hardwarefehler, wie z.B. Bitflips, emuliert werden, können bei `SFI` komplexere Fehler emuliert werden [8].

`SFI` wird eingesetzt um die folgenden Ziele zu erreichen [5]:

- Bewertung von Fehlertoleranzmechanismen.
- Prognose von worst-case Szenarien und experimentelle Risikobewertung.
- Dependability benchmarking. Dabei werden unter Anderem Maße bezüglich der Zuverlässigkeit eines Computersystems in Anwesenheit von Fehlern bewertet. Durch `SFI` werden für diese Benchmarks die Fehler injiziert.

2.2 Repräsentative Fehler

Wie in dem vorherigen Abschnitt beschrieben ist es für `SFI` wichtig repräsentative Fehler zu injizieren. In dieser Ausarbeitung wird das Fehlermodell aus `G-SWFIT` [5] mit kleinen Abweichungen verwendet, da bei dem in dieser Ausarbeitung vorgestellten Werkzeug in den Quelltext und nicht in den binären Code injiziert wird. Auf dieses Fehlermodell wird in Abschnitt 3 näher eingegangen.

In Duares Felddatenstudie [5] wurden zum Finden eines repräsentativen Fehlermodells 12 Open-Source Projekte bezüglich behobener Fehler analysiert, indem

eine relativ große Menge an patch- und diff-Files bezüglich vorgekommener Fehler analysiert und diese Fehler klassifiziert wurden. Damit eine Fehlerklasse nun repräsentativ ist, wurden zwei Bedingungen definiert [5]:

- Die Anzahl an auftretenden Fehlern muss mindestens so hoch sein wie im Durchschnitt.
- Ein solcher Fehler muss in mehr als zwei der analysierten Programmen aufgetreten sein.

2.3 SAFE

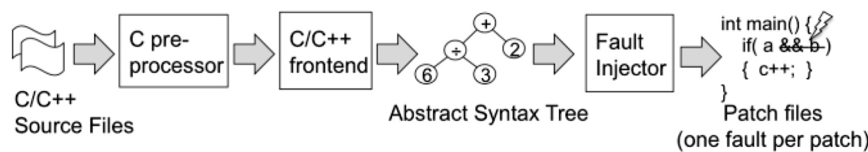


Abbildung 2.1: Vorgehen von SAFE bei der Fehlerinjektion [3]

`SoftwAre Fault Emulator` (`SAFE`) [8, 3] ist ein Werkzeug für Softwarefehlerinjektion, welches in C/C++-Projekte Fehler injiziert und für jeden injizierten Fehler ein Patch-File erstellt. Es verwendet das Fehlermodell aus `G-SWFIT` [5], auf welches im Abschnitt 3 näher eingegangen wird, mit dem Unterschied, dass in `G-SWFIT` die Injektionen in binären Code und nicht in Quelltext durchgeführt werden. Außerdem besteht mit dem Werkzeug die Möglichkeit, die erstellten Patch-Files auf die Software, in die injiziert wird, anzuwenden und mit vom Benutzer erstellten Skripten ist es dann möglich Tests auszuführen und die Ergebnisse dieser zu evaluieren. `SAFE` kann Fehler immer nur in jeweils eine C/C++-Datei injizieren, welche eine eigenständige Kompiliereinheit sein muss. Das bedeutet, dass vor dem Anwenden des Werkzeugs alle Präprozessor-Makros angewendet werden müssen, sodass die Datei nur noch C/C++-Code enthält, den der Compiler übersetzen kann. Dazu zählen:

- Include-Direktiven [9]: Bei einer Include-Direktive kopiert der Präprozessor die entsprechend angegebene Quelldatei an die Position der Direktive.
- Direktiven für bedingte Kompilierung [9] (if, elif, endif, ifdef, ifndef): Mit diesen Direktiven kann man mit einer angegebenen Bedingung entscheiden welcher Teil einer Datei später vom Compiler übersetzt werden soll bzw. welcher nicht.
- Define-Direktiven [9]: Mit diesen Makro-Direktiven werden Makros definiert, welche dann in der Quelltextdatei durch ihre Definition ersetzt werden.

Andere Direktiven wie z.B. Pragma-Direktiven beinhalten Informationen für den Compiler und werden von dem Präprozessor nicht aufgelöst, da der Compiler diese benötigt.

Bei der Injektion geht das Werkzeug wie in Abbildung 2.1 vor [3]:

1. Zunächst verwendet `SAFE` ein C/C++-Frontend, um einen `AST` zu erstellen.
2. In diesem sucht `SAFE` nach möglichen Injektionsstellen für den jeweiligen Fehler aus dem Fehlermodell.
3. Dann wird der entsprechende Fehler injiziert und ein Patch-File für diesen Fehler erzeugt.

Bei der Verwendung von `SAFE` hat sich gezeigt, dass `SAFE`

- nicht mit allen eingebundenen Dateien funktioniert hat,
- nicht in den Code neuerer C++-Versionen injizieren kann und
- es müssen sowohl ein relativ alter Präprozessor (`MCPP`), als auch eine alte Versionen der Standardbibliothek verwendet werden.

Außerdem verändert `SAFE` manchmal bei der Injektion gleichbleibende Stellen im Code so, dass sie semantisch äquivalent sind, aber nicht syntaktisch übereinstimmen. Das folgende Beispiel[1] zeigt dieses Verhalten bei einer Injektion eines Fehlers der Fehlerklasse MLAC[2]. Bei diesem wird aus dem folgenden `if`-Konstrukt das `true` entfernt.

```
1 if(argv[1][0] == '3' && true)
```

Dabei wird aber auch der andere Teil der Konjunktion, wie oben beschrieben, in ein semantisches Äquivalent umgeformt:

```
1 if( ( (* ( (* ( (argv)+ (1)))+ (0)))== ('3')))
```

2.4 Clang

Clang [2] ist ein Frontend für C-basierte Sprachen wie C, C++ oder Objective-C für den LLVM-Compiler, mit welchem schnelles Kompilieren bei wenig Speicherverbrauch angestrebt wird. Im Gegensatz zu `SAFE` können auch Dateien mit aktuellem C++-Code geparst und verarbeitet werden.

Um Features von Clang in eigenen Werkzeugen verwenden zu können, werden verschiedene Interfaces angeboten[3]:

[1] Dieses Beispiel ist entstanden, indem das `SAFE`-Werkzeug auf die Datei `test.cpp` aus dem Anhang angewendet wurde.

[2] Siehe Abschnitt 3.3.2.

[3] `https://clang.llvm.org/docs/Tooling.html` (abgerufen am 29.04.2018)

- `LibClang`[4] bietet ein C-Interface mit entsprechenden Python Bindings. Hier wird eine ziemlich hohe Abstraktion verwendet, wodurch man zwar durch den `AST` iterieren kann, aber keine vollständige Kontrolle über den `AST` hat.

- `ClangPlugins`[5] ist ein Interface, mit welchem eine weitere Aktion während eines Kompiliervorgangs auf dem `AST` ausgeführt werden kann. Dieses Interface kann aber nur als Teil eines Kompilierungsvorgangs eingesetzt werden.

- `LibTooling`[6] ist ein C++-Interface, welches volle Kontrolle über den Clang `AST` bietet und die Möglichkeit bietet Stand-Alone Werkzeuge zu erstellen.

2.4.1 Clang AST

Ein `Abstract Syntax Tree` (`AST`) ist eine Baum-Datenstruktur, welche die Syntax einer Programmiersprache abbildet und es somit wesentlich leichter macht bestimmte Stellen innerhalb des Quelltextes eines Programms zu identifizieren.

Alle Knoten eines `Clang ASTs` gehören jeweils einer der folgenden drei Oberklassen an [1]:

Decl Dieser Klasse gehören Deklarationen an.

Stmt Dieser Klasse gehören Anweisungen an.

Type Dieser Klasse gehören Typen an.

Zum Finden von `AST`-Knoten bietet Clang zwei verschiedene Mechanismen [1]:

ASTMatcher realisieren in Clang eine `DSL` (`Domain Specific Language`) mit der gesuchte Codestellen beschrieben werden können. Dann kann mit den gefundenen `AST`-Knoten direkt weitergearbeitet werden. Es ist außerdem möglich eigene Matcher zu entwerfen. Eine Liste aller von Clang bereitgestellten Matcher kann man in der `ASTMatcher`-Dokumentation[7] finden.

RecursiveASTVisitor ist eine abstrakte Klasse in der man für jede Art von `AST`-Knoten eine Funktion überschreiben kann, welche für jeden entsprechenden Knoten im `AST` einmal ausgeführt wird. Es ist also eine Möglichkeit durch alle Knoten des `AST` zu iterieren.

[4] `http://clang.llvm.org/doxygen/group__CINDEX.html` (abgerufen am 29.04.2018)
[5] `https://clang.llvm.org/docs/ClangPlugins.html` (abgerufen am 29.04.2018)
[6] `https://clang.llvm.org/docs/LibTooling.html` (abgerufen am 29.04.2018)
[7] `http://clang.llvm.org/docs/LibASTMatchersReference.html` (abgerufen am 29.04.2018)

2.5 Zusammenfassung

In diesem Kapitel wurden die nötigen Grundlagen für die nächsten Kapitel beschrieben. Dazu gehören der Begriff der `Software Fault Injection`, welches die Methodik des Werkzeugs dieser Arbeit ist, welches in den folgenden Kapiteln entworfen und implementiert wird. Außerdem wird erklärt, dass es wichtig ist, ein Fehlermodell aus repräsentativen Fehlern zu verwenden, um `SFI` sinnvoll umzusetzen. Das Fehlermodell für das Werkzeug dieser Arbeit wird im nächsten Kapitel erläutert.

Neben den Grundlagen für die Methodik des zu erstellenden Werkzeugs wurde auch ein bereits im Bereich `SFI` existierendes Werkzeug `SAFE` vorgestellt, wobei auf die Funktionsweise und auf Probleme, die bei der Anwendung aufgetreten sind, eingegangen wurde. Zuletzt ist noch das Frontend `Clang` für C-basierte Sprachen vorgestellt worden, welches für die Entwurfsentscheidungen in Kapitel 4 benötigt wird.

3 Fehlermodell

In diesem Kapitel wird das Fehlermodell, welches für die Fehlerinjektion im Rahmen dieser Ausarbeitung zugrunde gelegt wird, beschrieben. Wie bei `SAFE` [3] wird das Fehlermodell von `G-SWFIT` [5] verwendet, welches von Duares [5, 6] beschrieben wird und aus den in der Realität in Software am häufigsten vorkommenden Fehlerklassen besteht. Da unabhängig von Duares Studie auch andere Studien eine ähnliche Verteilung von Fehlern hervorgebracht haben, wird vermutet, dass dieses Fehlermodell repräsentativ für allgemeine Fehler in Programmen ist [5]. Es ist möglich, dass eine Analyse von mehr Programmen noch weitere Fehler hervorbringen kann, diese würden dann aber eher selten auftreten, da sie sonst in Duares Studie schon aufgetaucht wären [5].

Wichtig ist noch zu erwähnen, dass bei `G-SWFIT` Fehler nicht in den Quelltext, sondern in den kompilierten binären Code des Programms injiziert werden [5]. Bei der Injektion in den binären Code können Ungenauigkeiten auftreten, wodurch das Injizierte nicht der entsprechenden Änderung im Quelltext entspricht [3]. Dieses Problem tritt bei der Injektion in den Quelltext nicht auf und die injizierten Fehler entsprechen also immer der entsprechenden Fehlerklasse.

Im Folgenden wird nun auf die einzelnen Fehlerklassen eingegangen, die sich in Zuweisungs-, Algorithmus-, Überprüfungs- und Interface-Fehler gruppieren lassen. Alle folgenden Fehlerklassen sind aus Duares Studie [5, 6].

3.1 Zuweisungsfehler

In dieser Gruppe befinden sich Fehler, bei denen Zuweisungen oder Initialisierungen von Variablen entweder fehlen oder fehlerhaft sind.

3.1.1 MVIV (Missing Variable Initialization using a Value)

Diese Fehlerklasse beinhaltet Fehler, bei denen die Initialisierung einer Variable mit einem Wert fehlt. Eine Initialisierung ist nach Duares [6] die erste Zuweisung eines Wertes oder Ausdrucks zu einer Variable. Um nach Duares [6] näher an realistischen Softwarefehlern zu sein, müssen hier noch folgende Einschränkungen gelten:

- Die Anweisung darf nicht die einzige Anweisung in dem umgebenden Block sein.

- Die betreffende Variable muss lokal sein.
- Die Zuweisung darf nicht innerhalb einer Schleife geschehen, da diese dann mehrfach ausgeführt wird.
- Der Ausdruck darf nicht Teil eines `for`-Konstrukts sein, da der Fehler bei Duares Studie [5] nie innerhalb eines `for`-Konstrukts vorkam.

Da in C++ eine Initialisierung einer Variable nur während der Deklaration geschehen kann [9], werden in dieser Ausarbeitung nur Deklarationen mit Initialisierer als Initialisierungen gesehen. Somit ist es möglich, dass es Zuweisungen zu Variablen gibt, obwohl sie nicht während der Deklaration initialisiert wurden. Die erste Zuweisung wird in dem Fall nicht als Initialisierung angesehen, was einen Unterschied zu `SAFE` darstellt, denn `SAFE` sieht genau wie `G-SWFIT` die erste Zuweisung als Initialisierung.

In dieser Arbeit werden genau wie bei `SAFE` auch Initialisierungen innerhalb einer Schleife betrachtet, da im Quelltext, im Gegensatz zum Binärcode, durchaus unterschieden werden kann, ob eine Variable innerhalb einer Schleife deklariert und initialisiert wurde, oder nur eine Zuweisung erfolgt.

3.1.2 MVAV (Missing Variable Assignement using a Value)

Diese Fehlerklasse beinhaltet Fehler, bei denen eine Zuweisung eines Wertes zu einer Variable fehlt, wobei die Zuweisung keine Initialisierung sein darf. Außerdem behauptet die Studie [6], dass dieser Fehler am realistischsten ist, wenn folgendes gilt:

- Die Anweisung darf nicht die einzige Anweisung in dem umgebenden Block sein.
- Die betreffende Variable muss lokal sein.
- Der Ausdruck darf nicht Teil eines `for`-Konstrukts sein.

An dieser Stelle ist zu beachten, dass in dieser Arbeit alle Zuweisungen, die nicht bei der Deklaration geschehen, keine Initialisierungen sind[1].

3.1.3 MVAE (Missing Variable Assignement using an Expression)

Diese Fehlerklasse beinhaltet Fehler, bei denen eine Zuweisung eines Ausdrucks zu einer Variable fehlt. Die Zuweisung darf keine Initialisierung sein, was wie in

[1] Siehe Abschnitt 3.1.1.

Abschnitt 3.1.1 umgesetzt wird. Außerdem behauptet die Studie [6], dass dieser Fehler am realistischsten ist, wenn folgendes gilt:

- Die Anweisung darf nicht die einzige Anweisung in dem umgebenden Block sein.
- Die betreffende Variable muss lokal sein.
- Der Ausdruck darf nicht Teil eines `for`-Konstrukts sein.

3.1.4 WVAV (Wrong Value Assigned to Variable)

Diese Fehlerklasse beinhaltet Fehler, bei denen eine Zuweisung eines Wertes zu einer Variable mit einem falschen Wert vorhanden ist. Die Anweisung darf keine Initialisierung sein, die Variable muss lokal sein und die Zuweisung darf nicht innerhalb eines `for`-Konstrukts sein [6].

Um bei der Injektion für diesen Fehler Vergleichbarkeit zu schaffen und keine zufälligen Werte zu verwenden, wird hier das letzte Byte des Wertes komplementiert [6], bzw. bei boolschen Werten der Wert negiert.

3.2 Algorithmusfehler

Zu dieser Gruppe zählen Fehler, die bei der Implementierung von Algorithmen auftreten.

3.2.1 MFC (Missing Function Call)

Bei dieser Fehlerklasse fehlt ein Funktionsaufruf und nach Duares [6] ist dieser Fehler am realistischsten, wenn folgendes gilt:

- Der Rückgabewert des Funktionsaufrufs darf nicht verwendet werden.
- Der Aufruf darf nicht die einzige Anweisung innerhalb des umgebenden Blocks sein.

Es ist also wichtig zu überprüfen, ob die Funktion überhaupt einen Rückgabewert besitzt. Wenn ja, dann muss noch überprüft werden, ob dieser Teil eines Ausdrucks, einer Zuweisung oder eines anderen Konstruktes ist.

3.2.2 MIFS (Missing IF construct plus Statements)

Bei dieser Fehlerklasse fehlt ein `if`-Konstrukt mitsamt Anweisungen innerhalb des Konstrukts und nach Duares [6] ist dieser Fehler am realistischsten, wenn folgendes gilt:

- Das Konstrukt darf nicht die einzige Anweisung innerhalb des umgebenden Blockes sein.
- Das `if`-Konstrukt darf keinen `else`-Block besitzen.
- Es dürfen maximal 5 Anweisungen innerhalb des Blocks sein und diese dürfen keine Sprünge enthalten, mit Ausnahme von Funktionsaufrufen.

3.2.3 MIEB (Missing If construct plus statements plus Else construct)

Bei dieser Fehlerklasse fehlt ein `if`-Konstrukt mitsamt Anweisungen und außerdem das `else`-Schlüsselwort vor dem entsprechenden `else`-Konstrukt. Die Injektion dieser Fehlerklasse ist in Abbildung 3.1 dargestellt.

Operator	Example	Example with fault

OMIEB	if (*expression*)
	{
	statements-IF
	}
	else
	{
	statements-ELSE
	}
	
	... *remaining code*
		~~if (*expression*)~~
		~~{~~
		~~*statements-IF*~~
		~~}~~
		~~else~~
		~~{~~
		statements-ELSE
		~~}~~
		
		... *remaining code*

Abbildung 3.1: Bei dem Operator für die Fehlerklasse MIEB werden aus einem `if-else`-Konstrukt das `if`-Konstrukt und das `else`-Schlüsselwort entfernt, sodass nur noch die Anweisungen des `else`-Blocks übrig bleiben [6].

Außerdem behauptet die Studie [6], dass dieser Fehler am realistischsten ist, wenn folgendes gilt:

- Das `if`-Konstrukt muss einen `else`-Block besitzen.
- Es dürfen maximal 5 Anweisungen innerhalb des Blocks sein und diese dürfen keine Sprünge enthalten, mit Ausnahme von Funktionsaufrufen.

3.2.4 MLPA (Missing small and Localized Part of Algorithm)

Bei dieser Fehlerklasse fehlen ein paar hintereinander stehende Anweisungen. Außerdem behauptet die Studie [6], dass dieser Fehler am realistischsten ist, wenn

folgendes gilt:

- Die entfernten Anweisungen dürfen nicht die einzigen Anweisungen innerhalb des umgebenden Blocks sein.
- Alle Anweisungen befinden sich im selben Block, es sind keine Sprünge, mit Ausnahme von Funktionsaufrufen, vorhanden und es sind insgesamt maximal 5 Anweisungen, die entfernt werden.

3.3 Überprüfungsfehler

Bei dieser Gruppe handelt es sich um Fehler bei der Überprüfung von Daten.

3.3.1 MIA (Missing If construct Around statements)

Bei dieser Fehlerklasse fehlt ein `if`-Konstrukt um Anweisungen herum. Die Injektion dieser Fehlerklasse ist in Abbildung 3.2 dargestellt.

Operator	Example	Example with fault

OMIA	if (expression)
	{
	statements
	}
		~~if (expression)~~
		~~{~~
		statements
		~~}~~

Abbildung 3.2: Bei dem Operator für die Fehlerklasse MIA wird bei einem `if`-Konstrukt das `if`-Schlüsselwort entfernt, sodass nur noch die Anweisungen innerhalb des Konstrukts übrig bleiben [6].

Außerdem behauptet die Studie [6], dass dieser Fehler am realistischsten ist, wenn folgendes gilt:

- Das `if`-Konstrukt darf keinen `else`-Block besitzen.
- Es dürfen maximal 5 Anweisungen innerhalb des Blocks sein und diese dürfen keine Sprünge mit Ausnahme von Funktionsaufrufen enthalten.

3.3.2 MLAC (Missing And Clause in branch condition)

Bei dieser Fehlerklasse fehlt eine Konjunktion innerhalb eines Ausdrucks, welcher als Sprungbedingung verwendet wird. Dieses gilt also z.B. für Konjunktionen innerhalb der Bedingung eines `if`-Konstrukts.

3.3.3 MLOC (Missing Or Clause in branch condition)

Bei dieser Fehlerklasse fehlt eine Disjunktion innerhalb eines Ausdrucks, welcher als Sprungbedingung verwendet wird. Dieses gilt also z.B. für Disjunktionen innerhalb der Bedingung eines `if`-Konstrukts.

3.4 Interfacefehler

Bei dieser Gruppe handelt es sich um Fehler im Umgang mit Interfaces.

3.4.1 WPFV (Wrong Variable used in Parameter of Function Call)

Diese Fehlerklasse beinhaltet Fehler, bei denen innerhalb eines Funktionsaufrufs eine falsche Variable als Argument verwendet wird und die Studie [6] behauptet, dass dieser Fehler am realistischsten ist, wenn folgendes gilt:

- Es müssen mindestens 2 Variablen, bzw. Parameter innerhalb des Moduls vorhanden sein.
- Diese Variablen müssen lokal sein.

Ein Modul entspricht hier einer Subroutine, also z.B. einer Funktion [6]. Im Gegensatz zu `G-SWFIT` wird in dieser Arbeit auch noch der Typ der Variable betrachtet, was innerhalb des Binär-Codes nicht möglich ist, aber im Quelltext. Außerdem würde es zu Syntaxfehlern führen, wenn die Typen der Variablen nicht übereinstimmen. Somit wäre dies kein realistischer Fehler mehr, da sich das Programm nicht kompilieren ließe und der Compiler einen entsprechenden Fehler ausgeben würde.

Wenn die zu ersetzende Variable ein Parameter ist und ein Parameter mit dem selben Typ vorhanden ist, dann wird dieser für die Ersetzung verwendet, ansonsten wird eine lokale Variable mit demselben Typ verwendet [6]. Wenn die zu ersetzende Variable eine lokale Variable ist und eine weitere lokale Variable mit dem selben Typ vorhanden ist, dann wird diese für die Ersetzung verwendet, ansonsten wird ein Parameter mit demselben Typ verwendet [6].

3.4.2 WAEP (Wrong Arithmetic Expression in function call Parameter)

Bei dieser Fehlerklasse ist ein arithmetischer Ausdruck innerhalb eines Funktionsaufrufs fehlerhaft.

Bei der Injektion dieses Fehlers wird zur Vergleichbarkeit immer der letzte Operand entfernt [6].

3.5 Zusammenfassung

Zusammenfassend ist das Fehlermodell ursprünglich aus G-SWFIT [5] und wird auch von dem `SAFE`-Werkzeug verwendet. Allerdings gibt es zwei Unterschiede:

- Wie in `SAFE` werden auch Initialisierungen innerhalb von Schleifen als Initialisierungen betrachtet, da dies zu einer normalen Zuweisung im Quelltext klar unterscheidbar ist, im Gegensatz zum binären Code.
- In dieser Ausarbeitung werden Zuweisungen nur bei der Deklaration als Initialisierung angesehen und jede andere Zuweisung ist somit eine normale Zuweisung, wie im C++-Standard [9].

Insgesamt kann es bei der Injektion in binären Code zu Abweichungen zu den, der Fehlerklasse entsprechenden Quelltextänderungen, kommen, da je nach Compiler und Optimierung Konstrukte nicht mehr immer klar erkennbar sind.

4 Entwurf

In diesem Kapitel geht es um den Entwurf des Werkzeugs `clang-sfi`. Dabei wird zuerst auf die Anforderungen an das Werkzeug eingegangen, um danach Entscheidungen darüber zu treffen, wie diese umzusetzen sind.

4.1 Anforderungen

Das hier zu entwerfende Werkzeug soll die folgenden Anforderungen erfüllen:

- Das Werkzeug soll aktuellen C++-Code parsen und abstrahieren können.
- Das Werkzeug soll ein Stand-Alone Werkzeug sein.
- Das Werkzeug soll die Fehler des Fehlermodells aus dem vorherigen Abschnitt injizieren können.
- Das Werkzeug soll konfigurierbar sein.
- Das Werkzeug soll die injizierten Fehler abspeichern, sodass diese im Nachhinein auf das entsprechende Projekt angewandt werden können.
- Das Werkzeug soll die Fehler, nachdem diese abgespeichert wurden, auf das Projekt anwenden und Tests ausführen können.

4.2 Entwurfsentscheidungen

`SAFE` erfüllt nicht alle oben definierten Anforderungen, da z.B. nicht in C++-Code eines aktuellen Standards injizieren werden kann. Somit ist es notwendig ein eigenes Werkzeug zu erstellen, um aktuellen C++-Code parsen und abstrahieren zu können. In dieser abstrahierten Darstellung können dann Fehlerinjektionsstellen, welche dem Fehlermodell aus dem letzten Kapitel entsprechen, gesucht werden. Dafür gibt es zwei Möglichkeiten:

- Erstellen eines eigenen Parsers.
- Verwenden bestehender Bibliotheken, die den C++-Code parsen und abstrahieren können.

Da es sehr aufwendig ist einen eigenen C++-Parser zu schreiben und es bereits häufig verwendete und somit gut getestete Bibliotheken gibt, die diese Aufgabe übernehmen können, soll hier eine solche verwendet werden. Dafür bietet sich vor allem `Clang`[1] an, welches ein Frontend für C-basierte Sprachen wie C++ für den `LLVM`-Compiler ist [2]. Zur Abstraktion von Quelltext bietet `Clang` einen `AST` [1], eine Baumstruktur, welche die syntaktische Struktur des Quelltextes abbildet.

Um `Clang` verwenden zu können, muss man sich für eines der folgenden drei Interfaces entscheiden:

- `LibClang`[2] ist ein C-Interface. Es bietet die Möglichkeit Quelltext in einen `AST` zu parsen, einen bereits geparsten `AST` zu laden, durch einen `AST` zu traversieren und eine Verbindung zwischen den `AST`-Knoten und den entsprechenden Quelltextstellen herzustellen. Allerdings bietet diese Schnittstelle nicht alle Informationen des `Clang-AST`, um zwischen den verschiedenen Releases relativ stabil zu bleiben.

- `ClangPlugins`[3] ist ein C++-Interface und bietet die Möglichkeit während eines Kompilierungsprozesses eine weitere Aktion auf dem entsprechenden `AST` auszuführen. Dabei ist eine vollständige Kontrolle über den `AST` möglich.

- `LibTooling`[4] ist ein C++-Interface, welches es ermöglicht Stand-Alone Werkzeuge zu bauen. Dabei ist eine vollständige Kontrolle über den `AST` gewährleistet und ein hiermit erstelltes Werkzeug kann auf einzelne Dateien, unabhängig des entsprechenden Build Systems, angewandt werden.

Da bei `LibClang` keine vollständige Kontrolle über den `AST` erreicht wird, können mit diesem Interface nicht alle Anforderungen an `clang-sfi` erfüllt werden. `ClangPlugins` bietet die vollständige Kointrolle über den `AST`, allerdings nur als Schritt während eines Kompilierungsvorgangs, wodurch es nicht möglich ist ein Stand-Alone Werkzeug zu realisieren. `LibTooling` hingegen bietet neben der vollständigen Kontrolle über den `AST` auch die Möglichkeit ein Stand-Alone Werkzeug zu erstellen. Somit wird für die Realisierung der Fehlerinjektion bei diesem Werkzeug die `LibTooling` von `Clang` verwendet. Der Entwurf zu diesem `Clang`-Werkzeug wird im nächsten Abschnitt beschrieben.

Aus der Entscheidung `Clangs` `LibTooling` zu verwenden ergibt sich der Name für das zu entwickelnde Werkzeug `clang-sfi`, sowie die weitere Aufteilung dieses Kapitels in

1. den Entwurf des `Clang`-Werkzeugs,

[1] Siehe Abschnitt 2.4.
[2] `http://clang.llvm.org/doxygen/group__CINDEX.html` (abgerufen am 29.04.2018)
[3] `https://clang.llvm.org/docs/ClangPlugins.html` (abgerufen am 29.04.2018)
[4] `https://clang.llvm.org/docs/LibTooling.html` (abgerufen am 29.04.2018)

2. den Entwurf der `FaultInjector`-Klasse, welche für das Finden und Injizieren der Fehler zuständig ist,

3. der Art der Speicherung der injizierten Fehler und

4. den Entwurf der Konfiguration.

4.2.1 Clang-Werkzeug

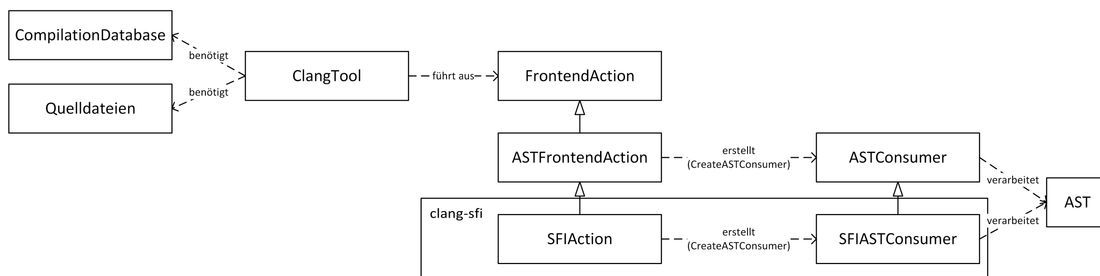


Abbildung 4.1: Darstellung der Zusammenhänge eines `Clang`-Werkzeuges und der Umsetzung in `clang-sfi`

Um ein `Clang`-Werkzeug mit der `LibTooling` zu erstellen benötigt man zum Parsen von Quelltextdateien eine `CompilationDatabase` [1], welche die Befehle, bzw. Argumente für das Kompilieren von Quelldateien beinhaltet. Diese kann durch Argumente in der Kommandozeile oder durch Lesen eines Bauverzeichnisses erstellt werden [1].

Dabei folgen die Kommandozeilenargumente, mit welchen die `CompilationDatabase` aufgebaut werden kann, der Syntax eines `Clang`-Aufrufs, vorausgesetzt man verwendet zum Parsen den `CommonOptionParser` [4]. Wenn diese `CompilationDatabase` aus einem Bauverzeichnis geladen wird, wird in diesem die `compile_commands.json`[5] ausgelesen und verarbeitet [1]. Um nun bei `clang-sfi` die selben Kommandozeilenargumente wie bei anderen `Clang`-Werkzeugen verwenden zu können, soll der `CommonOptionParser` verwendet werden.

Nun kann das Werkzeug mit Hilfe der zu erstellenden `CompilationDatabase` eine Aktion ausführen. Dafür stellt `LibTooling` die Klasse `FrontendAction` [1] bereit. Zum Finden von Injektionsstellen für das Fehlermodell aus Abschnitt 3 soll der `AST` von `Clang` verwendet werden. Dieser wird von einer `ASTFrontendAction` [4] für jede dem Werkzeug übergebene Quelldatei erstellt.

Zur Verwendung des `AST` muss ein sogenannter `ASTConsumer` innerhalb der `CreateASTConsumer`-Methode [4] der Aktion erstellt werden. Ein solcher hat dann

[5] Das Format dieser Datei wird auf `https://clang.llvm.org/docs/JSONCompilationDatabase.html` (abgerufen am 29.04.2018) beschrieben.

Zugriff auf den AST. Eine Übersicht über den Aufbau eines Clang-Werkzeugs und die Entsprechungen in clang-sfi sind in Abbildung 4.1 dargestellt. Bei dem Werkzeug clang-sfi heißt die Aktion SFIAction und der ASTConsumer SFIASTConsumer.

Das zu entwerfende Werkzeug muss den Fehlerklassen aus Abschnitt 3 entsprechende Injektionsstellen finden. Zum Suchen nach Codestellen innerhalb des AST bietet LibTooling zwei Möglichkeiten [1]:

- Ein RecursiveASTVisitor besitzt für jede Art von AST-Knoten eine Methode, die für jeden zutreffenden AST-Knoten einmal aufgerufen wird. Diese können überschrieben werden und erhalten als Argument den entsprechenden AST-Knoten.

- ASTMatcher realisieren eine Domain Specific Language, durch deren Hilfe es möglich ist AST-Stellen zu beschreiben und somit Codestellen zu finden. Dabei kann einem solchen Ausdruck ein String zugeordnet werden, mit welchem man später wieder auf den Ausdruck zurück schließen kann. Wenn nun AST-Knoten dem Ausdruck entsprechen, so wird ein Callback aufgerufen, welcher als Argument sowohl den AST-Knoten als auch den, dem Ausdruck zugeordneten String, erhält. Eine Übersicht über von Clang bereitgestellte ASTMatcher, aus denen man solche Ausdrücke erstellen kann, findet man in der Dokumentation[6].

ASTMatcher nehmen beim Identifizieren von Codestellen in einem AST einen großen Teil der Arbeit ab, die man bei der Verwendung eines RecursiveASTVisitor noch entwickeln müsste. Dementsprechend sollen bei clang-sfi für das Identifizieren von Codestellen ASTMatcher verwendet werden.

Für das Finden der Injektionsstellen und der Injektion von Fehlern wird im nächsten Abschnitt die Klasse FaultInjector entworfen. Subklassen dieser sollen jeweils Fehlerinjektionsstellen für eine Fehlerklasse aus dem Fehlermodell aus Abschnitt 3 finden, die Fehler an den gefundenen Stellen injizieren und den injizierten Fehler für das spätere Anwenden abspeichern.

4.2.2 FaultInjector

Die Klasse FaultInjector soll dafür zuständig sein Fehlerinjektionsstellen zu finden und Fehler zu injizieren. Um später das Fehlermodell ändern zu können, ohne den Quelltext anpassen zu müssen, soll für jede Fehlerklasse eine Subklasse erstellt werden. Nun sollen zuerst alle der Fehlerklasse entsprechenden Injektionsstellen gefunden werden und danach die Fehler injiziert und abgespeichert werden. Durch diese Aufteilung müssen die Injektionsstellen innerhalb der Klasse zwischengespeichert werden. Eine Übersicht über diese Klasse bietet Abbildung 4.2.

[6] http://clang.llvm.org/docs/LibASTMatchersReference.html (abgerufen am 29.04.2018)

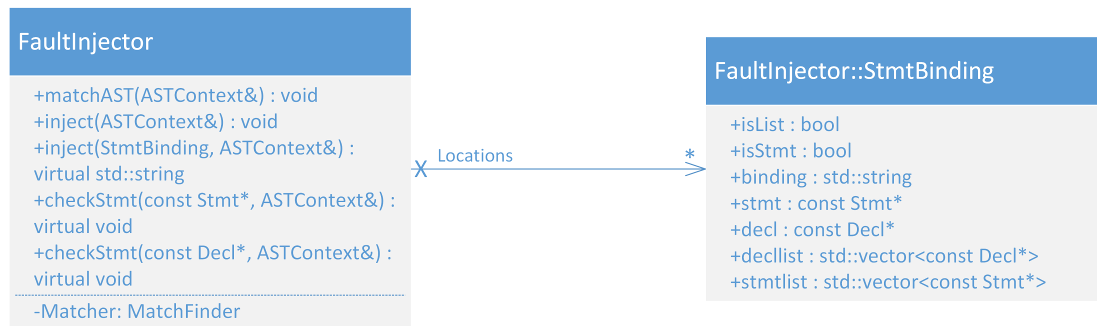


Abbildung 4.2: UML-Diagramm der `FaultInjector`-Klasse

Um Injektionsstellen für Fehler zu finden, soll die `Faultinjector`-Klasse `ASTMatcher` verwenden. Zur Verwendung von `ASTMatchern` gibt es `MatchFinder` [4], welche ein Hilfsmittel zum Finden von Codestellen auf dem `AST` sind. Diesen können `ASTMatcher` mit einem jeweiligen Callback hinzugefügt werden. Die `FaultInjector`-Klasse soll einen solchen `MatchFinder` erhalten und die für eine Fehlerklasse entsprechenden `ASTMatcher` und Callbacks müssen dann im Konstruktor der jeweiligen Subklasse hinzugefügt werden.

Ein `ASTMatcher`, welcher alle Variablendeklarationen, die gleichzeitig eine Initialisierung sind bei der nur eine Variable verwendet wird[7], würde z.B. so aussehen:

```
varDecl(
    hasInitializer(
        ignoringParenCasts(
            declRefExpr()
        )
    )
)
```

Hier wurde zusätzlich noch `ignoringParenCasts`[8] verwendet, was dafür sorgt, dass Klammern und Casts ignoriert werden. Dadurch kann die bei der Initialisierung verwendete Variable auch gecastet sein oder in Klammern stehen.

Nun soll in der `FaultInjector`-Klasse jeder Treffer der `ASTMatcher` auf weitere Eigenschaften geprüft werden können. Dafür soll die `FaultInjector`-Klasse eine `checkStmt`-Methode verwenden, welcher der Treffer, sowie der `AST` übergeben werden. Diese Methode soll darüber entscheiden, ob der Treffer eine Injektionsstelle ist oder nicht. Sollte es für eine Fehlerklasse nötig sein für einen Treffer noch andere Injektionsstellen hinzuzufügen, außer der durch den `ASTMatcher` gefundenen, so muss das an dieser Stelle getan werden.

Für die Zwischenspeicherung der Injektionsstellen muss nun eine Container-Klasse `StmtBinding` erstellt werden, welche eine Injektionsstelle, also einen `AST`-

[7] Also z.B. `int i = j;`
[8] `http://clang.llvm.org/docs/LibASTMatchersReference.html` (abgerufen am 29.04.2018)

Knoten oder eine Liste von `AST`-Knoten, enthält. Des Weiteren muss die Klasse den String, welcher dem `ASTMatcher` zugewiesen ist, sowie den Typ der Injektionsstelle enthalten. Von den drei möglichen Typen die in einem `Clang AST` vorkommen können (`Type`, `Decl`, `Stmt`[9]), müssen nur `Decl` und `Stmt` betrachtet werden, da alle Fehler aus dem Fehlermodell aus Abschnitt 3 nur Anweisungen oder Deklarationen betreffen. Für eine Übersicht über diese Klasse siehe Abbildung 4.2.

Für die Injektion der Fehler sind textuelle Veränderungen am Quelltext, in den injiziert wird, nötig. Dafür bietet `Clang` die `Rewriter` [4], welche ein Hilfsmittel zur textuellen Veränderung von Quelltext, für den ein `AST` erstellt worden ist, sind. Ein `Rewriter` entspricht einem Adapter für `RewriteBuffer` [4], welche Änderungen am Quelltext speichern können. Also ermöglichen `Rewriter` es, die Stellen im Quelltext, die einem `AST`-Knoten entsprechen, textuell zu verändern. Dabei wird der `AST` nicht verändert.

Die `FaultInjector`-Klasse soll nun eine `inject`-Methode besitzen, in welcher der Fehler injiziert werden soll. Dazu soll hier in den entsprechenden Subklassen ein `Rewriter` erstellt werden, welcher genutzt werden soll, um die der Fehlerklasse entsprechenden Quelltextänderungen auf diesem durchzuführen. Der geänderte Quelltext soll dann zurückgegeben werden, sodass die Änderungen abgespeichert werden können, um den Fehler im Nachhinein in dem Projekt, in welches injiziert werden soll, zu injizieren.

4.2.3 Speicherung der Fehler

Bei der Speicherung der Fehler ist es wichtig, dass genügend Informationen gespeichert werden, um den Fehler mit Hilfe der gespeicherten Datei in ein Projekt zu injizieren. Dafür bieten sich vor allem die `GNU Diffutils`[10] an, welche die Möglichkeit bieten, die Unterschiede zwischen zwei Dateien auszugeben. Diese Ausgabe kann dann wiederum in einer Datei gespeichert werden und später auf die Ursprungsdatei angewandt werden.

Dies bedeutet für die Speicherung von Fehlern, dass zuerst der vollständige modifizierte Quelltext, wie er in der `inject`-Methode zurückgegeben wird, gespeichert werden muss. Danach verwendet man die `GNU Diffutils`, um die Unterschiede, die durch die Fehlerinjektion entstehen, zu finden und zu speichern.

4.2.4 Konfiguration

In diesem Abschnitt geht es um die Konfiguration von `clang-sfi`. Für das Werkzeug müssen die folgenden Informationen per Konfiguration angegeben werden:

- Benötigte Befehle und Argumente für Kompilieren und Ausführung.

[9] Siehe Abschnitt 2.4.1.
[10] `https://www.gnu.org/software/diffutils/` (abgerufen am 29.04.2018)

- Die zu verwendenden `FaultInjectors` (Fehlermodell).
- Der Speicherort für die Fehlerinjektionsdateien.

Für die Konfiguration gibt es zwei Möglichkeiten:

- Verwendung von ausschließlich Kommandozeilenparametern.
- Verwendung einer Konfigurationsdatei.

Die Verwendung von Kommandozeilenparametern ist hier nicht sinnvoll, da der Befehl zum Ausführen von `clang-sfi`, je nach Konfiguration, extrem lang werden kann. Dementsprechend soll eine Konfigurationsdatei erstellt werden. Dabei würden sich vor allem zwei Formate anbieten:

- Das `JSON`-Format [10].
- Das `XML`-Format [7].

Das `XML`-Format ist schwerer zu parsen, für Zugriffe auf einzelne Daten muss man durch eine hierarchische Struktur iterieren und es ist größer. Das `JSON`-Format ist einfacher aufgebaut, bietet aber keine Erweiterungsmöglichkeit wie das `XML`-Format. Beide Formate sind menschenlesbar und können die für `clang-sfi` nötigen Key-Value Paare aus Tabelle 4.1 darstellen. Da das `JSON`-Format etwas kleiner und somit übersichtlicher ist, bietet es sich an dieses zu verwenden.

4.3 Zusammenfassung

Zusammenfassend ist in diesem Kapitel der Entwurf des in dieser Ausarbeitung zu erstellenden Werkzeugs `clang-sfi` und die dabei getroffenen Entscheidungen über die zu verwendenden Technologien beschrieben.

Dabei hat sich ergeben, dass für dieses Werkzeug zum Parsen und Abstrahieren von C++-Code die `LibTooling` von `Clang` verwenden soll, damit sowohl aktueller C++-Code geparst werden kann, als auch die Stand-Alone Eigenschaft von `clang-sfi` sichergestellt werden kann. Dabei soll der `Clang AST` zur Abstraktion verwendet werden und `ASTMatcher`, um Fehlerinjektionsstellen innerhalb des `AST` zu identifizieren.

Des Weiteren soll eine Klasse `FaultInjector` erstellt werden, welche für das Finden der Injektionsstellen, das Injizieren der Fehler und das Speichern dieser zuständig ist, wobei für jede Fehlerklasse aus dem Fehlermodell aus Abschnitt 3 eine Subklasse erstellt werden soll. Dies vereinfacht die Anpassung des Fehlermodells, indem nur noch entschieden werden muss, welche `FaultInjectors` auf den `AST` der Quelldatei angewandt werden sollen.

Für das Speichern der Fehler haben sich die `GNU DiffUtils` als nützlich erwiesen. Bei der Konfiguration von `clang-sfi`, die unter anderem das Fehlermodell, sowie Befehle für das Kompilieren und die Ausführung beinhalten soll, hat sich eine Datei des `JSON`-Formats als beste Option gezeigt.

Im folgenden Kapitel wird auf die Implementierung von `clang-sfi` und somit auf die Anwendung dieser Technologien, bzw. Entwurfsentscheidungen eingegangen.

Name	**Typ**	**Bedeutung**
destDirectory	String	Verzeichnis in dem alles was von dem Werkzeug erstellt wird gespeichert werden soll
verbose	Bool	Wenn `true` wird die Konsolenausgabe des Werkzeugs erweitert
injectors	[String]	Array von `Strings`, welche den Namen der zu injizierenden Fehlern entsprechen
compileCommand	String	Befehl welcher zum Kompilieren des Projektes ausgeführt werden soll (optional)
compileCommandArgs	[String]	Array von Argumenten, welche dem Befehl zum Kompilieren des Projektes übergeben werden sollen (optional)
fileToExec	String	Befehl welcher nach dem Kompilieren ausgeführt werden soll (optional)
fileToExecArgs	[String]	Argumente welche dem Befehl `fileToExec` übergeben werden sollen (optional)
timeout	Int	Zeit in Millisekunden nach welcher die Ausführung von `fileToExec` abgebrochen werden soll (optional)
multipleRuns	Int	Anzahl an Ausführungen von `fileToExec` pro injiziertem Fehler

Tabelle 4.1: Diese Tabelle stellt die für die Konfiguration von `clang-sfi` nötigen Optionen, deren Typ und Bedeutung dar.

5 Implementierung

Im folgenden Kapitel wird auf die Implementierung des Entwurfs aus Abschnitt 4 von `clang-sfi` eingegangen. Dabei wird auch auf die Implementierungen der einzelnen Unterklassen von `FaultInjector` für die jeweilige Fehlerklasse eingegangen. Daraus ergibt sich die Aufteilung dieses Abschnitts in die Implementierung des Hauptprogramms, des `Clang`-Werkzeugs, der `FaultInjector`-Klassen und einer Zusammenfassung.

Im Folgenden wird, wie in Abschnitt 4 beschrieben, die `LibTooling`[1] von `Clang` verwendet. Alle verwendeten Methoden und Konstrukte dieser sind in der entsprechenden Dokumentation [4] zu finden.

5.1 Hauptprogramm

Das Hauptprogramm muss drei Aufgaben erfüllen:

- Das Auslesen und Parsen der Konfiguration.
- Das Erstellen des `Clang`-Werkzeugs, welches Fehler injizieren und als Patchfiles abspeichern soll.
- Das Anwenden, Kompilieren des Projekts mit injizierten Fehlern und Ausführen von Tests für das Projekt.

Dafür wird zuerst ein `CommonOptionParser` [4] erstellt, der die Kommandozeilenargumente, die beim Aufruf angegeben werden, parst. Zuzüglich zu den standardmäßig unterstützten Argumenten eines `Clang`-Werkzeugs werden noch die in Tabelle 5.1 dargestellten Argumente beachtet. Die Argumente `verbose` und `dir` überschreiben dabei die Werte aus der Konfigurationsdatei.

Nach dem Parsen der Kommandozeilenargumente wird zunächst der Inhalt der Konfiguration mit Mitteln der Standardbibliothek ausgelesen und mit Hilfe eines `JSON`-Parsers geparst, um daraufhin die Werte aus der Konfiguration[2] in Variablen, die im Programm verwendet werden können, zu speichern. Als Parser wird dabei der `JSON`-Parser von Nlohmann verwendet, dessen Dokumentation[3] auf GitHub

[1] `https://clang.llvm.org/docs/LibTooling.html` (abgerufen am 29.04.2018)
[2] Für den Aufbau der Konfigurationsdatei siehe Abschnitt 4.2.4.
[3] `https://github.com/nlohmann/json` (abgerufen am 29.04.2018)

Argument	Typ	Bedeutung
verbose	Bool	Wenn gesetzt wird die Konsolenausgabe des Werkzeugs erweitert.
dir	String	Verzeichnis in dem alle von dem Werkzeug erzeugten Dateien gespeichert werden (Standard: aktuelles Verzeichnis).
config	String	Pfad zur Konfigurationsdatei (Standard: ./config.json).
no-inject	Bool	Wenn gesetzt wird keine Injektion von Fehlern durchgeführt.
compile	Bool	Wenn gesetzt werden, ggf. direkt nach der Injektion, die Fehler auf das Projekt angewandt, kompiliert und der angegebene Befehl ausgeführt. Für diese Option müssen in der Konfigurationsdatei `compileCommand` und `fileToExec` gesetzt sein und eine Fehlerinjektion muss vorher durchgeführt worden sein.

Tabelle 5.1: Diese Tabelle zeigt die von `clang-sfi` unterstützen Kommandozeilenargumente, deren Typ und Bedeutung.

zu finden ist. Dieser ermöglicht das Lesen und Schreiben von `JSON`-Strukturen mit Hilfe von Iteratoren oder dem überschriebenen Array-Index-Operator. Mit dieser Bibliothek kann man dann z.B. wie folgt ein `JSON`-Objekt erstellen und dem Schlüssel `name` den Wert `"Max Mustermann"` zuordnen:

```
1 nlohmann::json data;
2 data["name"] = "Max Mustermann";//Zugriff über Array-Index-Operator
3 std::string name = "";
4 if(data.find("name") != data.end())
5 //überprüfen ob Eintrag "name" vorhanden ist
6     name = data.find("name").get<std::string>();
7     //Zugriff über Iterator
```

Zum Auslesen aus z.B. einem Filestream kann der » -Operator verwendet werden.

```
1 nlohmann::json data;
2 std::ifstream file("config.json");
3 //Erstellen eines Lesestreams für die Datei config.json
4 file >> data;
5 //Inhalt des Streams mit Nlohmanns JSON-Parser parsen
```

Aus den in der Konfiguration angegebenen Fehlerklassen wird nun eine ihnen entsprechende Liste von `FaultInjectors` erstellt. Diese Liste wird dann an die Aktion des `Clang`-Werkzeugs weitergegeben, welches im nächsten Abschnitt erläutert wird. Informationen über Fehler, wie Anzahl und Aufteilung dieser auf die

Fehlerklassen, und die Angaben, die in der Konfigurationsdatei vorhanden sind, werden im Zielverzeichnis in einer `summary.json` gespeichert. Somit wird das spätere Anwenden der Fehler vereinfacht und eine Übersicht über die injizierten Fehler erstellt.

Zum Anwenden und Kompilieren der Fehler, sowie zum Ausführen der Tests, wird dann diese `summary.json` geparst. Es werden nun für jeden zu injizierenden Fehler folgende Schritte durchlaufen:

1. Anwenden des Fehlers,
2. Kompilieren des Projektes[4] und
3. Ausführen des angegebenen Befehls[5].

Bevor Fehler angewandt werden, muss zuerst die Quelldatei gesichert werden, in welche injiziert wird, um diese am Ende wiederherzustellen. Denn beim Anwenden eines Fehlers wird auf die Quelldatei das Patch-File des Fehlers angewandt, wodurch diese verändert wird.

Danach wird zum Kompilieren das `compileCommand` mit den `compileCommandArgs` ausgeführt. Sollte dabei ein Fehler auftreten, wird dies über die Standardausgabe ausgegeben und in der `summary.json` vermerkt. Beim Kompilieren werden die Standardausgabe und die Standardfehlerausgabe in Dateien umgeleitet[6], um die spätere Fehlersuche zu vereinfachen.

Sollte kein Fehler aufgetreten sein, so wird das `fileToExec` mit den `fileToExecArgs` ausgeführt. Dabei werden ebenfalls Standardausgabe und Standardfehlerausgabe in Dateien umgeleitet. Der Exit-Code der Ausführung wird in der `summary.json` gespeichert. Außerdem kann in der Konfiguration ein `timeout` angegeben werden, nachdem die Ausführung dieses Befehls abgebrochen werden soll. Dies wird umgesetzt, indem ein Thread erstellt wird, der nach dem Timeout die Ausführung des Befehls beendet. Auch dies wird in der `summary.json` vermerkt. Wenn `multipleRuns` in der Konfiguration gesetzt ist, wird das Ausführen von `fileToExec` den gesetzten Wert mal wiederholt.

Um das Projekt am Ende wieder herzustellen muss nach dem Anwenden, Kompilieren und Testen der Fehler die Quelldatei mittels des Backups in ihren Ursprungszustand überführt werden.

[4] Das Projekt wird durch die Ausführung des `compileCommand` aus der Konfigurationsdatei kompiliert.

[5] Für Tests wird `fileToExec` aus der Konfigurationsdatei ausgeführt.

[6] Die `POSIX` `dup2`-Methode ermöglicht es Standardausgabe und Standardfehlerausgabe in eine Datei umzuleiten.

5.2 Clang-Werkzeug

Das `Clang`-Werkzeug besteht, wie im letzten Kapitel beschrieben, aus zwei Komponenten, der `FrontendAction`, bzw. `SFIAction` und dem `ASTConsumer`, bzw. `SFIASTConsumer`. Für das `Clang`-Werkzeug wird nun zuerst ein `ClangTool` [4] erstellt, wobei die `CompilationDatabase` übergeben werden muss. Mit dessen `run`-Methode kann dann die `SFIAction` ausgeführt werden. Da diese aber die Liste von `FaultInjectors` in ihrem Kontruktor benötigt, musste hier die `FrontendActionFactory` [4] angepasst werden:

```
1  std::unique_ptr<FrontendActionFactory>
       newSFIFrontendActionFactory(std::vector<FaultInjector*>
       injectors){
2      class SFIFrontendActionFactory: public FrontendActionFactory{
3          public:
4              SFIFrontendActionFactory(std::vector<FaultInjector*>
                   inj):FrontendActionFactory(),injectors(inj){}
5              //Aufruf des Konstruktors von FrontendActionFactory und
6              //Liste von FaultInjectors mit der übergebenen Liste inj
                   initialisieren
7              FrontendAction *create() override{
8                  return new SFIAction(injectors);
9                  //beim erstellen der SFIAction die Liste von
                       FaultInjectors an die SFIAction übergeben
10             }
11         private:
12             std::vector<FaultInjector*> injectors;
13             //Liste von FaultInjectors als zusätzliches
                   Klassenattribut
14     };
15     return std::unique_ptr<FrontendActionFactory>(
16                 new SFIFrontendActionFactory(injectors)
17                 //Liste von FaultInjectors an den Konstruktor von
                       SFIFrontendActionFactory übergeben
18             );
19
20 };
```

5.2.1 SFIAction

Dem Konstruktor dieser Klasse wird eine Liste[7] von `FaultInjectors` übergeben, welche innerhalb dieser Aktion Injektionsstellen für die Fehler der entsprechenden Fehlerklassen finden und danach die entsprechenden Fehler injizieren. Dafür wird die Methode `CreateASTConsumer` [4] überschrieben, welche in einer `ASTFrontendAction` die `ASTConsumer` für die einzelnen Kompiliereinheiten erstel-

[7] Hier umgesetzt durch `std::vector`.

len [4]. Bei der Erstellung des `ASTConsumers` werden nun dem Konstruktor von `SFIASTConsumer` der Dateiname der Quelldatei und die Liste von `Faultinjectors` übergeben, damit bei der Fehlerinjektion nur Injektionsstellen innerhalb der angegebenen Quelldatei und nicht in ihren Includes betrachtet werden.

5.2.2 SFIASTConsumer

In dieser Klasse wird im Konstruktor jedem übergebenen `FaultInjector` der Dateiname der Quelldatei übergeben, damit beim Injizieren nur Codestellen innerhalb der Quelldatei betrachtet werden.

Nun wird noch die Funktion `HandleTranslationUnit` [4] überschrieben, welche für jede Übersetzungseinheit ausgeführt wird und den `ASTContext` übergeben bekommt. In dieser werden für jeden `FaultInjector` aus der Liste die folgenden beiden Funktionen des `FaultInjectors` ausgeführt:

- Die `matchAST`-Methode findet innerhalb des `AST` Injektionsstellen für die Fehlerklasse und speichert diese im `FaultInjector`.

- Die `inject`-Methode injiziert für jede vorher gefundenen Injektionsstelle entsprechend der Fehlerklasse einen Fehler und speichert diesen als Patch-File.

5.3 FaultInjector

In dieser Superklasse werden die Grundlagen für die Fehlerinjektion der Subklassen gelegt, welche jeweils für die Injektion einer einzelnen Fehlerklasse zuständig sind. Dafür werden zunächst die für Injektion und Ausgabe nötigen Attribute erläutert, um danach auf das Finden, Injizieren und Abspeichern der Fehler, sowie auf die Ausgabe dabei einzugehen.

Für die Injektion und Speicherung der Fehler muss in dieser Klasse zuerst das Zielverzeichnis, sowie der Dateiname der Quelldatei gespeichert werden. Dazu gibt es zwei Attribute vom Typ String, in denen diese Informationen mittels `setFileName`- und `setDirectory`-Methode gespeichert werden können. Des Weiteren muss es für die Ausgabe des Tools die Attribute `verbose` vom Typ `bool` mit einer entsprechenden `setVerbose`-Methode, welche der Klasse signalisiert die Ausgabe umfangreicher zu gestalten, sowie einer `toString`-Methode, welche den Namen der Fehlerklasse zurückgeben soll und in jeder Subklasse überschrieben werden muss, denn diese wird neben der Ausgabe auch für den Dateinamen der Patch-Files für die Fehlerklasse verwendet. Das bedeutet dann z.B. für die Fehlerklasse MIA, dass diese Methode `"MIA"` zurückgibt.

5.3.1 Finden von Injektionsstellen

Zur Wiederholung aus Abschnitt 4: Die `FaultInjector`-Klasse besitzt zum Finden von `AST`-Knoten einen `MatchFinder`, welchem dafür `ASTMatcher` samt Callbacks hinzugefügt werden müssen. Die Callbacks werden aufgerufen, sobald eine Übereinstimmung des jeweiligen `ASTMatchers` vorliegt [4]. Da diese `ASTMatcher` für jede Fehlerklasse spezifisch sind, müssen diese im Konstruktor der jeweiligen Subklasse hinzugefügt werden.

Diese Callbacks müssen Objekte vom Typ `MatchFinder::MatchCallback` [4] sein. Dies ist eine Abstrakte Klasse, dessen `run`-Methode bei einer Übereinstimmung aufgerufen wird und diese als Argument übergeben bekommt. Die hier erstellte und verwendete von `MatchCallback` abgeleitete Klasse `StmtHandler` erhält außerdem durch ihren Konstruktor den `FaultInjector`, für den dieser Callback verwendet wird und den Dateinamen der Quelldatei, in welcher Übereinstimmungen gefunden werden sollen. Außerdem wird diesem eine Liste von Strings übergeben, welche den an die `ASTMatcher` gebundenen entsprechen, da nur diese angegebenen `ASTMatcher` betrachtet werden sollen.

Die `run`-Methode iteriert nun durch alle angegebenen Strings[8], die ab nun `Bindings` genannt werden. Dann wird überprüft, ob es eine Übereinstimmung für ein `Stmt` gibt und ob diese sich in der Quelldatei befindet. Analog dazu wird das ganze für `Decl` überprüft. Da, wie in Abschnitt 4 beschrieben, nur diese beiden Typen überprüft werden müssen, bleibt es bei dieser Fallunterscheidung bezüglich des `AST`-Knotentyps:

```
1  void StmtHandler::run(const MatchFinder::MatchResult &Result){
2      for(std::string binding:bindings){
3          if(const Stmt *stmt =
               Result.Nodes.getNodeAs<clang::Stmt>(binding)){
4              std::string name(
5                  Result.Context->getSourceManager().getFilename(
6                      stmt->getLocStart()
7              ));
8              if(faultInjector->getFileName().compare(name)==0){
9              //nur Nodes aus dem zu parsenden File beachten!!
10                 if(faultInjector->checkStmt(stmt, binding,
                       *Result.Context))
11                 //nur Anweisungen die durch checkStmt erlaubt sind
12                     faultInjector->nodeCallback(binding, stmt);
13             }
14         } else if(const Decl *decl =
               Result.Nodes.getNodeAs<clang::Decl>(binding)){
15             std::string name(
16                 Result.Context->getSourceManager().getFilename(
17                     decl->getLocStart()
18             ));
```

[8]Hier wird `std::string` verwendet.

```
19                if(faultInjector->getFileName().compare(name)==0){
20                //nur Nodes aus dem zu parsenden File beachten!!
21                    if(faultInjector->checkStmt(decl, binding,
                        *Result.Context)){
22                    //nur Deklarationen die durch checkStmt erlaubt sind
23                        faultInjector->nodeCallback(binding, decl);
24                    }
25                }
26            }
27        }
28 }
```

Für jede Übereinstimmung in der Quelldatei, die einem der angegebenen `Bindings` entspricht, wird für den entsprechenden `AST`-Knoten die `checkStmt`-Methode des `FaultInjectors` aufgerufen. Diese `checkStmt`-Methode muss also für `Decl` und `Stmt` als `AST`-Knoten existieren und entscheidet darüber ob der gefundene Knoten der Liste von Injektionsstellen hinzugefügt werden soll. Das Hinzufügen geschieht durch die `nodeCallback`-Methode des `FaultInjectors`, in welcher ein `StmtBinding`[9] zu dem entsprechenden Knoten zusammen mit seinem `Binding` erstellt wird, der Liste von Injektionsstellen hinzugefügt wird und die Liste entsprechend der Positionen des Knotens in der Quelldatei sortiert wird[10]. In `FaultInjector` selbst geben beide `checkStmt`-Methoden `false` zurück, da für manche Fehlerklassen nur eine der beiden Methoden benötigt wird und somit dann nur diese eine überschrieben werden muss.

Um das ganze in Gang zu setzen muss die `matchAST`-Methode des `MatchFinders` [4] aufgerufen werden, welche auf dem `AST` nach Übereinstimmungen mit den hinzugefügten `ASTMatchern` sucht und bei Treffern die `run`-Methode des entsprechenden `MatchCallbacks`, bzw. `StmtHandlers` aufruft. Ein Aufruf der `matchAST`-Methode des `FaultInjectors` wie in Abschnitt 5.2.2 sorgt genau dafür, dass die entsprechende Methode des `MatchFinders` aufgerufen wird.

5.3.2 Injektion und Speichern der Patch-Files

Die von dem `SFIASTConsumer` aufgerufene `inject`-Methode des `FaultInjectors` sorgt für die Injektion und Speicherung von Fehlern der entsprechenden Fehlerklasse. Dazu müssen dieser eine Liste von Injektionsstellen in Form von `StmtBindings` und der `ASTContext` als Argumente übergeben werden. Der `SFIASTConsumer` übergibt dabei die Liste der gefundenen Fehlerinjektionsstellen, die in dem `FaultInjector` gespeichert ist. Zur Injektion wird dann durch alle der Methode übergebenen Injektionsstellen iteriert und folgendes ausgeführt:

[9] Siehe Abschnitt 4.
[10] Wird umgesetzt durch den Vergleich mit dem `<` -Operators der `SourceLocations` der `AST`-Knoten, welche durch die `getLocStart`-Methode [4] des Knotens erhalten werden.

- Wenn `verbose` gesetzt ist, wird die `printStep`-Methode ausgeführt, welche Informationen über die aktuell abzuarbeitende Injektionsstelle ausgibt.

- Es wird eine weitere `inject`-Methode ausgeführt, welche für die Injektion eines einzelnen Fehlers konzipiert ist. Sie gibt bei erfolgreicher Injektion den Text der Quelldatei mit dem injizierten Fehler zurück, im Fehlerfall einen leeren String. Ist `verbose` gesetzt ist, so wird entsprechend des Ergebnisses entweder

  ```
  1  -Success
  ```

  oder

  ```
  1  -Failed
  ```

  ausgegeben.

- Wenn die Injektion für die aktuelle Injektionsstelle erfolgreich ist, so wird die `writeDown`-Methode ausgeführt. Diese speichert den injizierten Fehler ab und benötigt dazu als Argumente den Text der Quelldatei mit injiziertem Fehler und für den Dateinamen beim Speichern die Nummer des aktuell zu injizierenden Fehlers.

5.3.2.1 printStep

Diese Methode gibt Informationen über die aktuell auszuführende Injektion aus. Zuerst wird die Fehlerklasse ausgegeben, zusammen mit der Nummer der aktuellen Injektionsstelle und der Gesamtzahl der Injektionsstellen der Fehlerklasse. Des Weiteren wird, wenn es sich bei der Injektionsstelle um einen einzelnen `AST`-Knoten handelt, die Position in der Quelldatei mit Hilfe der `print`-Methode der `SourceLocations` [4] für den Anfang und das Ende des Knotens, sowie die Quelltextstelle, die diesem Knoten entspricht mittels der `printPretty`-Methode eines `Stmts` [4], bzw. der `print`-Methode eines `Decls` [4], ausgegeben. Sollte es sich um eine Liste von `AST`-Knoten handeln, so wird statt dessen die Anzahl an Knoten ausgegeben.

Für die Fehlerklasse MIA könnte eine solche Ausgabe dann z.B. so aussehen:

```
1  injecting 'MIA' [18/18]
2  src/lib_json/json_value.cpp:1670:7 -
       src/lib_json/json_value.cpp:1672:7
3  if (!node->isArray()) {
4  }
```

5.3.2.2 inject-Methode (Einzelfehler)

Diese Methode ist abstrakt und muss somit in den Subklassen für die einzelnen Fehlerklassen implementiert werden. Als Argumente werden die Injektionsstelle an der injiziert werden soll, als auch der `ASTContext` übergeben.

In diesen Implementationen werden `Rewriter` verwendet, um an dem Text der Quelldatei eine textuelle Veränderung vorzunehmen, die der entsprechenden Fehlerklasse entspricht. Zum Erstellen eines solchen sind der `SourceManager`[11] und die `LangOptions`, welche Informationen über den verwendeten C-Dialekt enthalten, von nöten. Beides wird vom `ASTContext` geliefert [4].

Um den veränderten Quelltext als String zu erhalten, muss man zuerst den für die Quelldatei entsprechenden `RewriteBuffer` aus dem `Rewriter` extrahieren. Dies ist wie folgt möglich:

```
1     //R ist hier ein Rewriter
2     //Context ist der ASTContext
3     RewriteBuffer rb =
          R.getEditBuffer(Context.getSourceManager().getMainFileID());
```

Die `getMainFileID`-Methode [4] gibt dabei die ID der Quelldatei zurück, welche benötigt wird, um den entsprechenden `RewriteBuffer` zu erhalten.

Um nun den injizierten Quelltext als String zu speichern, kann man den Inhalt des `RewriteBuffers` in einen Stream schreiben und mit Hilfe von z.B. einem von `Clang` bereitgestellten `raw_string_ostream`[12] den Inhalt des Streams in einen String schreiben:

```
1     //buffer ist hier der RewriteBuffer
2     std::string str;
3     raw_string_ostream stream(str);
4     //Initialisierung eines raw_string_ostream, der in den String
          str schreibt
5     buffer.write(stream);
6     //schreiben aus dem RewriteBuffer in den Stream stream
7     stream.flush();
8     //flushen des Streams, sodass alle Änderungen am Stream,die
          nicht automatisch geflusht wurden, auch im String str
          vorhanden sind
```

Für diese Funktionalität, die veränderte Quelldatei als Text zu erhalten, bietet ein `FaultInjector` die Methode `getEditedString` an.

In den folgenden Subklassen wird in jeder `inject`-Methode auf diese Weise ein `Rewriter R` erstellt und am Ende die Rückgabe der `getEditedString`-Methode zurückgegeben.

[11] Ein `SourceManager` ist zuständig für das Cachen von Quellcode im RAM und um Informationen über eine `SourceLocation` abzurufen [4].

[12] Ein `raw_string_ostream` ist ein Outputstream, der in einen String schreibt. (`http://llvm.org/doxygen/classllvm_1_1raw__string__ostream.html` abgerufen am 29.04.2018)

5.3.2.3 writeDown

Die `writeDown`-Methode speichert zuerst die veränderte Quelldatei, um danach aus dieser das entsprechende Patch-File zu erstellen. Dazu wird die dieser Methode als Text übergebene angepasste Quelldatei mittels eines `std::ofstream` gespeichert und danach das diff-Werkzeug aus den `GNU DiffUtils` mit dem Argument `-u0` angewendet. Dieses sorgt dafür, dass das Patch-File nur Informationen, die die Ersetzung betreffen, enthält und nicht den Kontext dieser, um die Datei klein zu halten.

Beide Dateien folgen dem folgendem Namenschema:

```
dir/<Fehlerklasse>_<Fehlernummer>
```

Dabei entspricht `dir` dem Verzeichnis, in welchem die Ausgaben von `clang-sfi` gespeichert werden. Es kann in der Konfiguration oder durch das `dir`-Kommandozeilenargument gesetzt werden.

5.3.3 Subklassen für das Fehlermodell

Bei den folgenden `FaultInjectors` für die einzelnen Fehlerklassen wird jeweils auf die `ASTMatcher` eingegangen, welche im Konstruktor zum Finden von Injektionsstellen hinzugefügt werden, die `checkStmt`-Methode /-Methoden die überschrieben wird / werden, auf die zu implementierende `inject`-Methode, welche für die Injektion eines einzelnen Fehlers einer Fehlerklasse zuständig ist, und auf andere Methoden von `FaultInjector` eingegangen, die überschrieben werden müssen.

5.3.4 MIAInjector

Der Injektor für die Fehlerklasse MIA fügt dem `MatchFinder` im Konstruktor folgenden `ASTMatcher` hinzu:

```
ifStmt(
    unless(
        hasElse(
            stmt()
        )
    )
).bind("ifStmt")
```

Dieser Ausdruck findet im `AST` alle Knoten, welche einem `if`-Konstrukt ohne `else`-Konstrukt entsprechen.

Da hier nur `if`-Konstrukte gefunden werden und diese eine Subklasse von `Stmt` sind, muss hier die entsprechende `checkStmt`-Methode für `Stmt` überschrieben werden. Entsprechend der Fehlerklasse muss in der `checkStmt`-Methode noch sichergestellt werden, dass maximal 5 Anweisungen von dem `if`-Konstrukt umfasst werden und diese außer Funktionsaufrufe keine Sprünge enthalten. Das bedeutet,

dass keine Schleifen[13], Fallunterscheidungen[14], `return`-Anweisungen und andere kontrollflussverändernde Anweisungen[15] enthalten sein dürfen.

Deshalb muss nun die von dem `if`-Konstrukt umfasste Anweisung überprüft werden. Ist diese ein `CompoundStmt`, also ein Block, so wird zuerst die Anzahl der von dem Block umfassten Anweisungen überprüft. Wenn diese größer als 5 ist, so gibt `checkStmt` den Wert `false` zurück. Ansonsten muss für jede Anweisung innerhalb des Blocks überprüft werden, dass diese keinem Sprung, mit Ausnahme von Funktionsaufrufen, entspricht. Nur wenn dies für alle Anweisungen innerhalb des Blocks erfüllt ist, wird `true` zurückgegeben.

Ist die von dem `if`-Konstrukt umfasste Anweisung kein Block, so wird überprüft, ob diese Anweisung einem Sprung außer einem Funktionsaufruf entspricht und es wird entsprechend `false`, bzw. `true` zurückgegeben.

Für die Injektion dieser Fehlerklasse muss das `if`-Schlüsselwort mitsamt der Bedingung entfernt werden. Dies entspricht der Spanne der Quelltextposition des `if`-Schlüsselwortes bis zu einem Zeichen vor den von diesem umfassten Anweisungen:

```
1 SourceRange range(ifStatement->getLocStart(),
     ifStatement->getThen()->getLocStart().getLocWithOffset(-1));
2 R.RemoveText(range);
```

5.3.5 MIFSInjector

Der Injektor für diese Fehlerklasse verwendet denselben `ASTMatcher` wie der MIA-Injector, da auch hier nur `if`-Konstrukte ohne `else`-Konstrukt betrachtet werden.

Auch bei dieser Subklasse muss die `checkStmt`-Methode für den Typ `Stmt` überschrieben werden. Innerhalb dieser muss folgendes überprüft werden:

- Es dürfen maximal 5 Anweisungen und keine Sprünge, mit Ausnahme von Funktionsaufrufen, von dem `if`-Konstrukt umfasst werden.
- Das `if`-Konstrukt darf nicht die einzige Anweisung innerhalb des umgebenden Blocks sein.

Dabei wird der erste Teil wie bei dem MIAInjector überprüft.

Um zu überprüfen, ob das `if`-Konstrukt die einzige Anweisung innerhalb des umgebenen Blocks ist, muss zuerst der Elternknoten des `if`-Konstrukts bestimmt werden und danach die Anzahl an Anweisungen in diesem überprüft werden. Dazu muss der erste nicht implizite Elternknoten verwendet werden, da implizite `AST`-Knoten keinen Anweisungen im Quelltext entsprechen. Diesen Knoten erhält

[13]Dazu zählen `for`-, `while`- und `do-while`-Schleifen.
[14]Dazu zählen `if`- und `switch`-Konstrukte.
[15]Dazu zählen `continue`-, `break`-, `throw`-, `goto`-Anweisungen, sowie ein `try-catch`-Konstrukt.

man, indem man die `getParents`-Methode des `ASTContext` [4] so oft verwendet, bis als Ergebnis ein nicht impliziter Knoten als Elternknoten zurückgegeben wird. Alle Knoten der Typen `ExprWithCleanups`, `MaterializeTemporaryExpr`, `CXXBindTemporaryExpr` und `ImplicitCastExpr` sind dabei implizite Knoten[16].

Handelt es sich dabei um einen Block, so gibt die `size`-Methode des `CompoundStmt` [4] die Anzahl an Anweisungen innerhalb des Blocks an. Bei einem `case`-Element aus einem `switch`-Konstrukt entsprechen die Anweisungen zwischen zwei `case`-Elementen, bzw. einem `case`-Element und einer `break`-Anweisung einem Block. Um dann die Anzahl von Anweisungen zu erhalten, werden diese beginnend mit der Anweisung nach dem `case`-Element, welche durch die `getSubStmt` eines `SwitchCases` erhalten wird, gezählt. Die weiteren Anweisungen werden gezählt, indem durch die Kindknoten des `switch`-Konstruktes, zu dem das `case`-Element gehört, iteriert wird. Dabei wird das `switch`-Konstrukt erhalten, indem man die `getParents`-Methode des `ASTContext` verwendet. Es wird nun durch die Kindknoten des `switch`-Konstrukts iteriert bis das `case`-Element, dessen Größe überprüft werden soll, erreicht ist und ab dort jede Anweisung gezählt bis eine `break`-Anweisung, ein weiteres `case`-Element, ein `default`-Element oder das Ende des `switch`-Konstruktes erreicht ist.

Wenn nun die Anzahl an Anweisungen in dem umgebenden Block größer als 1 ist, was bedeutet, dass außer dem `if`-Konstrukt noch mindestens eine weitere Anweisung in diesem enthalten sein muss, und innerhalb des `if`-Konstrukts maximal 5 Anweisungen ohne Sprünge, mit Ausnahme von Funktionsaufrufen, vorhanden sind, so gibt die `checkStmt`-Methode `true` zurück.

In der `inject`-Methode dieses `FaultInjectors` muss nun ein gefundenes `if`-Konstrukt mitsamt Anweisungen entfernt werden. Das entspricht der Quelltextspanne zwischen dem Beginn des `if`-Konstrukts bis hin zum Ende der von diesem umfassten Anweisungen:

```
1 SourceRange range(ifStatement->getLocStart(),
    ifStatement->getThen()->getLocEnd());
2 R.RemoveText(range);
```

5.3.6 MIEBInjector

Bei diesem FaultInjector müssen `if`-Konstrukte gefunden werden, welche einen `else`-Block besitzen. Dafür wird folgender `ASTMatcher` hinzugefügt:

```
1 ifStmt(
2     hasElse(
3         stmt()
4     )
5 ).bind("ifStmt")
```

[16]Siehe `Stmt::IgnoreImplicit`-Methode im Clang Quelltext (`lib/AST/Stmt.cpp`).

Die `checkStmt`-Methode muss hier für den Typ `Stmt` überschrieben werden und muss überprüfen, dass sich maximal 5 Anweisungen innerhalb des `if`-Konstrukts befinden und diese außer Funktionsaufrufen keine Sprünge enthalten. Dies wird wie bei dem MIAInjector umgesetzt.

In der `inject`-Methode muss alles von dem `if`-Konstrukt bis auf die Anweisungen in dem `else`-Teil entfernt werden, was der Quelltextspanne von dem `if`-Schlüsselwort bis zum letzten Zeichen des `else`-Schlüsselwortes entspricht:

```
1 SourceRange range(ifStatement->getLocStart(),
    ifStatement->getElse()->getLocStart().getLocWithOffset(-1));
2 R.RemoveText(range);
```

5.3.7 MFCInjector

Bei dieser Subklasse müssen Funktionsaufrufe gefunden werden, dessen Rückgabewert nicht verwendet wird. Dies wird mit folgendem `ASTMatcher` realisiert:

```
1  callExpr(
2      ignoringImplicit(
3          unless(
4              anyOf(
5                  hasAncestor(varDecl(isDefinition())),
6                  hasParent(returnStmt()),
7                  hasParent(callExpr()),
8                  hasParent(binaryOperator()),
9                  hasParent(unaryOperator()),
10                 cxxOperatorCallExpr()
11             )
12         )
13     )
14 ).bind("FunctionCall")
```

Dabei findet `callExpr` jeden Funktionsaufruf und es wird ausgeschlossen, dass dieser Aufruf ein Operatoraufruf im Gegensatz zu einem Funktionsaufruf oder Teil einer Variableninitialisierung ist und dass dieser Funktionsaufruf nicht von einer `return`-Anweisung, einem weiteren Funktionsaufruf oder von einem Operator verwendet wird. Somit wird der Rückgabewert jeder gefundenen Anweisung nicht verwendet.

Ein Funktionsaufruf entspricht dem `AST`-Knotentyp `Stmt` und somit muss hier die `checkStmt`-Methode für diesen Typ überschrieben werden. Innerhalb dieser muss noch entsprechend der Fehlerklasse überprüft werden, ob es sich bei dem Funktionsaufruf um die einzige Anweisung innerhalb des umgebenden Blocks handelt. Wenn ja, dann wird die Anweisung nicht als Injektionsstelle verwendet. Umgesetzt wird dies wie bei dem MIFSInjector.

In der `inject`-Methode muss hier die gefundene Anweisung entfernt werden:

```
1 SourceRange range(statement->getLocStart(), statement->getLocEnd());
```

```
R.RemoveText(range);
```

5.3.8 MVIVInjector

Bei dem MVIVInjector müssen Variablendeklarationen von lokalen Variablen, welche mit einem Wert initialisiert werden, gefunden werden. Um zu gewährleisten, dass es sich um eine lokale Variable handelt, muss der Deklarationskontext, in dem diese deklariert wird, eine Funktionsdeklaration sein. Des Weiteren muss die Initialisierung mit einem Wert erfolgen, welches gewährleistet wird indem Funktionsaufrufe, new-Aufrufe, Operatoraufrufe, Konstruktoraufrufe, Variablenreferenzen und Zugriffe auf Attribute eines Objektes nicht betrachtet werden[17]. Daraus folgt dieser `ASTMatcher`:

```
varDecl(
    allOf(
        hasInitializer(
            unless(
                anyOf(
                    callExpr(),
                    hasDescendant(callExpr()),
                    cxxNewExpr(),
                    hasDescendant(cxxNewExpr()),
                    binaryOperator(),
                    hasDescendant(binaryOperator()),
                    unaryOperator(),
                    hasDescendant(unaryOperator()),
                    cxxConstructExpr(),
                    hasDescendant(cxxConstructExpr()),
                    declRefExpr(),
                    hasDescendant(declRefExpr()),
                    memberExpr(),
                    hasDescendant(memberExpr())
                )
            )
        )
        ,hasDeclContext(functionDecl())
    )
).bind("variable")
```

An dieser Stelle werden Variablendeklarationen gefunden, dementsprechend muss die `checkStmt`-Methode für den Typ `Decl` überschrieben werden. Innerhalb dieser muss überprüft werden, ob die Deklaration die einzige Anweisung in dem Umgebenen Block ist, welches wie bei dem MIAInjector überprüft wird, und ob die Deklaration eine statische lokale Variable deklariert, da diese nicht auf dem Stack

[17] `hasDescendant` sorgt dafür, dass diese Konstrukte auch innerhalb von Klammern oder Casts erkannt werden (Siehe `ASTMatcher`-Dokumentation).

gespeichert wird. Dies lässt sich mit Hilfe der `isStaticLocal`-Methode [4] einer Variablendeklaration überprüfen. Außerdem muss sichergestellt werden, dass, wenn diese Deklaration im `AST` eine `for`-Schleife als Vorfahren hat, diese in dem Rumpf der Schleife vorkommt.

Um dies zu überprüfen, wird zuerst die `getParents`-Methode des `ASTContext` so oft aufgerufen bis entweder eine `for`-Schleife gefunden wurde oder die Wurzel des `AST` erreicht wurde. Wenn eine `for`-Schleife gefunden wurde, muss überprüft werden, ob die Deklaration innerhalb des Rumpfes ist. Dafür wird die `getParents`-Methode des `ASTContext` für die Deklaration, bzw. die Vorfahren dieser, so oft aufgerufen, bis entweder der entsprechende Schleifenrumpf gefunden wurde oder die Wurzel des `AST` erreicht wurde.

Wenn nun die Deklaration nicht die einzige Anweisung innerhalb des umgebenden Blocks ist, nicht Teil eines `for`-Konstruktes ist und eine lokale Variable deklariert, so gibt die `checkStmt`-Methode `true` zurück.

Für die Injektion wird eine `VarDecl` [4] mittels des Kopierkonstruktors aus dem gefundenen `AST`-Knoten erstellt, welche bei `Clang` einer Variablendeklaration entspricht. In dieser wird dann mittels `setInit`-Methode [4] der Initialisierer entfernt. Danach muss die textuelle Entsprechung der gefundenen Variablendeklaration durch die der neu erstellten ersetzt werden. Dazu wird diese zuerst in einen String umgewandelt indem mit der `print`-Methode der `VarDecl` [4] in einen `raw_string_ostream` geschrieben wird, der dann wiederum den Inhalt als String ausgibt.

Die Ersetzung mit Hilfe eines `Rewriters` erfolgt dann so:

```
1 SourceRange range(declaration->getLocStart(),
      declaration->getLocEnd());
2 R.ReplaceText(range, textWithoutInit);
```

5.3.9 MVAVInjector

Bei diesem `FaultInjector` müssen Zuweisungen gefunden werden, dazu wird der `ASTMatcher binaryOperator` verwendet, mit der Einschränkung, des Operatornamen `"="`.

```
1 binaryOperator( hasOperatorName("="))
```

Außerdem werden dem `ASTMatcher` noch Einschränkungen für die Ausdrücke auf der linken und rechten Seite des Zuweisungsoperators übergeben. Auf der linken Seite dürfen keine Funktionsaufrufe vorkommen:

```
1 unless(hasDescendant(callExpr()))
```

Es dürfen lokale Variablen oder Variablen eines lokalen Objektes, sowie

```
1 declRefExpr(to(varDecl(hasDeclContext(functionDecl())))),
2 memberExpr(hasObjectExpression(
3     declRefExpr(to(varDecl(hasDeclContext(functionDecl()))))))
```

diese einfach dereferenziert[18], als auch als Basis eines Array-Zugriffs[19] auf der linken Seite auftreten. Bei den einzelnen Schritten hier werden jeweils noch `ignoringParenCasts` und `ignoringImplicit`[20] hinzugefügt, sodass auf der linken Seite der Zuweisung auch beliebige Klammerungen und Casts stehen dürfen.

Auf der rechten Seite der Zuweisung dürfen bei dieser Fehlerklasse nur Werte stehen, woraus der folgende Teil des `ASTMatchers` für die rechte Seite der Zuweisung folgt:

```
hasRHS(
    unless(anyOf(
        ignoringParenCasts(ignoringImplicit(callExpr())),
        ignoringParenCasts(ignoringImplicit(cxxNewExpr())),
        ignoringParenCasts(ignoringImplicit(binaryOperator())),
        ignoringParenCasts(ignoringImplicit(unaryOperator())),
        ignoringParenCasts(ignoringImplicit(cxxConstructExpr())),
        ignoringParenCasts(ignoringImplicit(declRefExpr())),
        ignoringParenCasts(ignoringImplicit(memberExpr()))
    ))
)
```

Bei dieser Subklasse muss die `checkStmt`-Methode für den Typ `Stmt` überschrieben werden. In dieser muss überprüft werden, ob die gefundene Anweisung die einzige in dem umgebenen Block ist und ob die Anweisung kein Teil eines `for`-Konstruktes ist, was wie bei dem MVIVInjektor umgesetzt wird.

Bei der `inject`-Methode wird hier die gefundene Anweisung entfernt:

```
SourceRange range(statement->getLocStart(), statement->getLocEnd());
R.RemoveText(range);
```

5.3.10 MVAEInjector

Der MVAEInjector arbeitet genau wie der MVAVInjector im vorherigen Abschnitt mit dem einzigen Unterschied, dass auf der rechten Seite der Zuweisung keine Werte sondern Ausdrücke stehen sollen. Deshalb muss der Teil des `ASTMatchers` für die rechte Seite so angepasst werden:

```
hasRHS(
    anyOf(
        ignoringParenCasts(ignoringImplicit(callExpr())),
        ignoringParenCasts(ignoringImplicit(cxxNewExpr())),
        ignoringParenCasts(ignoringImplicit(binaryOperator())),
        ignoringParenCasts(ignoringImplicit(unaryOperator())),
        ignoringParenCasts(ignoringImplicit(cxxConstructExpr())),
        ignoringParenCasts(ignoringImplicit(declRefExpr())),
```

[18] Dies entspricht der Verwendung des Dereferenzierungsoperators.
[19] Umgesetzt mit dem `ASTMatcher` `arraySubscriptExpr(hasBase(...))`.
[20] `http://clang.llvm.org/docs/LibASTMatchersReference.html` (abgerufen am 29.04.2018)

```
            ignoringParenCasts(ignoringImplicit(memberExpr()))
        )
)
```

5.3.11 WVAVInjector

Der WVAVInjector arbeitet genau wie der MVAVInjector mit zwei Unterschieden. Bei der `checkStmt`-Methode muss nicht überprüft werden, ob der `AST`-Knoten die einzige Anweisung innerhalb des umgebenden Blocks ist und die Zuweisung wird in der `inject`-Methode nicht entfernt, sondern das letzte Byte des Wertes, welcher zugewiesen wird, wird invertiert. Dazu wird zuerst die Spannweite des Quelltextes der rechte Seite der Zuweisung, welche mittels der `getRHS`-Methode des `BinaryOperators` [4] erhalten wird, verwendet, um mit der `getRewrittenText`-Methode des `Rewriters` [4] den Text der rechten Seite der Zuweisung zu erhalten. Dieser wird nun um `^0xFF` erweitert, um das letzte Byte zu invertieren:

```
SourceRange range(rightSide->getLocStart(), rightSide->getLocEnd());
std::string text = R.getRewrittenText(range);
R.ReplaceText(range, text+"^0xFF");
```

Sollte es sich bei der rechten Seite um ein boolsches Literal handeln, so wird statt dessen der Wert `true` durch `false` ersetzt, bzw. anders herum.

5.3.12 MLOCInjector

Bei dieser Subklasse müssen durch die hinzugefügten `ASTMatcher` Anweisungen gefunden werden, bei denen innerhalb einer Sprungbedingung ein binärer Operator verwendet wird. Dabei wird jeweils ein `ASTMatcher` für `switch`-, `if`-, `do-while`-, `while`- und `for`-Konstrukte, welche in ihrer Bedingung einen binären Operator haben hinzugefügt. Für ein `for`-Konstrukt sieht der `ASTMatcher` dann z.B. so aus:

```
forStmt(
    hasCondition(
        anyOf(
            binaryOperator(),
            hasDescendant(binaryOperator())
        )
    )
).bind("for")
```

Hier muss die `checkStmt`-Methode für den Typ `Stmt` überschrieben werden. Sie gibt immer `false` aus, da die durch den `ASTMatcher` gefundenen Knoten nicht den Injektionsstellen entsprechen.

Bei der gefundenen Anweisung wird mittels `getCond`-Methode [4] die Bedingung für den Sprung in dieser Anweisung erhalten. Es wird rekursiv durch alle Kindknoten der Bedingung iteriert. Für jeden dabei gefundenen binären Operator, welcher

einem logischen Oder entspricht, wird für den linken und rechten Operanden jeweils überprüft, ob dieser auch einem logischen Oder entspricht, wenn nicht, so wird dieser den Injektionsstellen hinzugefügt. Dabei wird in dem `StmtBinding` gespeichert, welcher Operand entfernt werden soll.

Die `inject`-Methode entfernt dann entweder die Quelltextstelle vom Anfang des linken Operanden des Operators bis einschließlich dem Operator selbst, oder von dem Operator bis zum Ende des rechten Operanden, je nachdem welcher Operand durch das `StmtBinding` beschrieben wird:

```
SourceLocation start, end;
if(stmtbinding.left){
    start = binaryoperator->getLHS()->getLocStart();
    end =
        binaryoperator->getRHS()->getLocStart().getLocWithOffset(-1);
}else {
    start = binaryoperator->getOperatorLoc();
    end = binaryoperator->getRHS()->getLocEnd();
}
SourceRange range(start, end);
R.RemoveText(range);
```

5.3.13 MLACInjector

Der MLACInjector wird genauso umgesetzt wie der MLOCInjector mit dem Unterschied, dass Injektionsstellen Konjunktionen sind, also binären Operatoren, welche einem logischen Und entsprechen.

5.3.14 WAEPInjector

Bei dieser Subklasse müssen Funktionsaufrufe gefunden werden, bei denen mindestens ein Argument ein arithmetischer Ausdruck ist. Dazu wird ein `ASTMatcher` verwendet, welcher Funktionsaufrufe findet, bei denen mindestens ein binärer Operator in einem der Argumente verwendet wird:

```
callExpr(
    allOf(
        unless(
            cudaKernelExpr(),
            cxxOperatorCallExpr(),
            userDefinedLiteral()
        ),
        hasDescendant(
            expr(
                binaryOperator()
            )
        )
    )
```

```
14 ).bind("functionCall")
```

Die checkStmt-Methode muss für den Typ Stmt überschrieben werden. Sie gibt immer false zurück, da hier nicht die durch den ASTMatcher gefundenen AST-Knoten als Injektionsstellen verwendet werden. In dieser Methode wird durch die Argumente des Funktionsaufrufs iteriert, indem durch dessen Kindknoten im AST iteriert wird. Für jeden dieser Knoten, der ein arithmetischer Ausdruck, also ein arithmetischer binärer Operator[21], ist, wird der am weitesten rechte binäre Operator mittels Aufrufen der getRHS der BinaryOperators [4] erhalten. Dieser wird dann als Injektionsstelle hinzugefügt.

In der inject-Methode wird dann der letzte Operand der Injektionsstelle, also der Text von dem Operator bis zum Ende des rechten Operanden des Operators, entfernt:

```
1 SourceRange range(binaryOperator->getOperatorLoc(),
      binaryOperator->getRHS()->getLocEnd());
2 R.RemoveText(range);
```

5.3.15 WPFVInjector

Bei dieser Subklasse werden durch den ASTMatcher Funktionsaufrufe gesucht, welche als Argument eine Variable übergeben bekommen:

```
1  callExpr(
2      allOf(
3          unless(anyOf(
4              cudaKernelCallExpr(),
5              cxxOperatorCallExpr(),
6              userDefinedLiteral()
7          )),
8          hasAnyArgument(
9              anyOf(
10                 declRefExpr(
11                   to(varDecl(hasDeclContext(functionDecl())))
12                 ),
13                 implicitCastExpr(hasSourceExpression(
14                     declRefExpr(
15                       to(varDecl(hasDeclContext(functionDecl())))
16                     )
17                 ))
18             )
19         )
20     )
21 ).bind("functionCall")
```

[21]Dazu zählen Multiplikation, Division, Modulo, Addition, Subtraktion, Bitshift, Konjunktion, Disjunktion und XOR.

Die `checkStmt`-Methode muss hier für den Typ `Stmt` überschrieben werden. Es wird immer `false` zurückgegeben, da die Funktionsaufrufe nicht die Injektionsstellen sind.

Nun wird durch alle Argumente des gefundenen Funktionsaufrufs iteriert und für jedes Argument, welches einer Variable entspricht, wird der Kontext der Deklaration bestimmt. Wenn dieser einer Funktionsdeklaration entspricht, ist die Variable lokal.

Aber damit eine Injektion möglich ist, muss noch mindestens eine weitere lokale Variable oder ein Parameter vom selben Typ vorhanden sein. Dies wird überprüft indem durch alle Variablendeklarationen innerhalb des Deklarationskontextes iteriert wird und überprüft wird, ob die Variablen, die vom selben Typ sind und an der Stelle des gefundenen Funktionsaufrufs bereits deklariert sind[22]. Außerdem wird durch die Parameter der Methode, in der sich der Funktionsaufruf befindet, iteriert und geprüft, ob es Parameter gibt, welche den selben Typ wie das zu überprüfende Argument besitzen. Wenn mindestens eine weitere solche lokale Variable oder Parameter existiert, wird das Argument den Injektionsstellen hinzugefügt.

Bei der Injektion in der `inject`-Methode werden für eine Injektionsstelle wieder zuerst alle Parameter gesucht, die den selben Typ wie die Variable der Injektionsstelle haben und danach alle Variablen gesucht, die diesem Typ entsprechen und innerhalb der umgebenden Funktion vor der Injektionsstelle deklariert wurden und erreichbar sind. Wenn es sich bei der Injektionsstelle um eine Referenz auf einen Parameter handelt und ein weiterer Parameter mit dem selben Typ vorhanden ist, wird dieser nun als Argument verwendet, sonst eine lokale Variable vom selben Typ. Handelt es sich bei dem gefundenen Argument um eine lokale Variable, so wird zuerst nach einer weiteren lokalen Variable vom selben Typ gesucht und danach nach einem Parameter.

Bei der Injektion wird dann der Text des Arguments durch den Variablennamen der gefundenen Variable bzw. des Parameters vom selben Typ ersetzt:

```
1 std::string variableName(localVariable->getName().data());
2 R.ReplaceText(argument->getSourceRange(), variableName);
```

5.3.16 MLPAInjector

Bei dieser Subklasse müssen durch den `ASTMatcher` Blöcke, also `CompoundStmts` gefunden werden:

```
1 compoundStmt().bind("compoundStmt")
```

`CompoundStmts` sind vom Typ `Stmt`, deshalb muss die `checkStmt`-Methode dieses Typs überschrieben werden. Diese gibt immer `false` zurück, da die gefunde-

[22] Dies wird überprüft indem die Deklaration der Variable vor dem Funktionsaufruf geschieht und durch Closure an der Stelle erreichbar ist.

nen AST-Knoten nicht den Injektionsstellen entsprechen. Innerhalb dieser Methode müssen nun der Fehlerklasse entsprechende Injektionsstellen gefunden werden, welche immer einer zusammenhängenden Menge von Anweisungen innerhalb des Blocks entsprechen.

Dazu werden zuerst maximale entfernbare Mengen von Anweisungen erstellt, welche ohne Syntaxfehler entfernt werden können und die keine Sprünge außer Funktionsaufrufe enthalten. Dafür wird durch alle Anweisungen innerhalb des Blocks iteriert und die aktuelle Anweisung wird einer Liste von Anweisungen, welche später einer dieser maximalen Mengen entspricht, hinzugefügt, falls diese Anweisung keinem Sprung, der kein Funktionsaufruf ist, und keiner Deklaration[23] entspricht. Ansonsten ist die Liste maximal und es wird eine neue Liste begonnen.

Nun werden für jede so erstellte Liste Injektionsstellen erstellt. Dazu werden Teillisten dieser zuerst in maximal der Größe der Liste minus 1, bzw. fünf Anweisungen, je nachdem welche Zahl kleiner ist, erstellt. Dadurch ist gewährleistet, dass mindestens eine Anweisung übrig bleibt, wenn die Anweisungen der Injektionsstelle entfernt werden. Danach werden für die selbe Liste Teillisten jeder kleineren Größe erstellt.

Die Teillisten werden mit Hilfe von Iteratoren des `std::vector` erstellt und dann als Injektionsstellen hinzugefügt:

```
1 std::vector<const Stmt*> list(maxList.begin()+begin,
    maxList.begin()+begin+size);
```

In der `inject`-Methode wird durch die Liste von Anweisungen iteriert und dabei die erste und letzte Quelltextposition bestimmt, damit alle Anweisungen dieser Liste entfernt werden können. Die Injektion läuft dann so:

```
1 SourceRange range(firstPosition, lastPosition);
2 R.RemoveText(R, Conetxt);
```

5.4 Zusammenfassung

In diesem Kapitel wurde die Implementierung der Entwurfsentscheidungen aus dem Abschnitt 4 beschrieben.

Dabei wurde darauf eingegangen wie `clang-sfi` die entworfene Konfiguration parst, Fehler injiziert, diese anwendet und Tests durchführt. Die Konfigurationsdatei im `JSON`-Format wird mit dem `JSON`-Parser von Nlohmann geparst. Kommandozeilenargumente werden mit Hilfe des `CommonOptionParsers` von `Clang` geparst, damit der Aufruf dieses Werkzeugs der selben Syntax wie der Aufruf anderer `Clang`-Werkzeuge folgt.

Zur Verwendung der `LibTooling` wird aus den Kommandozeilenargumenten, bzw. der `compile_commands.json` eine `CompilationDatabase` erstellt, welche be-

[23] Da dies in der Studie [6] bei dieser Fehlerklasse nicht vorkommt.

nötigt wird, um eine Aktion auf dem `Clang`-Frontend auszuführen. In dieser werden mit Hilfe von `ASTMatchern` Injektionsstellen für die zu injizierenden Fehlerklassen gefunden, danach nochmals überprüft und gegebenenfalls angepasst, um mit Hilfe von `Rewritern` Fehler des Fehlermodells aus Abschnitt 3 zu injizieren.

Die angepassten Quelltexte werden dann für jeden Fehler gespeichert und mit Hilfe der `GNU DiffUtils` werden Patch-Files für jeden einzelnen Fehler erstellt.

Zum Kompilieren des Projekts, in welches injiziert werden soll, wird der in der Konfiguration angegebene Befehl verwendet. Dasselbe gilt für Tests.

Nach dem Injizieren, bzw. Kompilieren und Ausführen wird eine Zusammenfassung über die Aktionen des Werkzeugs in einer `JSON`-Datei gespeichert, um eine Übersicht zu erhalten. Des Weiteren werden alle Ausgaben während des Kompilierens und Ausführens in Dateien weitergeleitet um eine Auswertung der Tests und gegebenenfalls eine Fehlersuche zu vereinfachen.

Im folgenden Kapitel wird `clang-sfi` evaluiert, indem das Werkzeug angewendet wird und die Ergebnisse mit denen von `SAFE` und von Hand gesuchten Injektionsstellen verglichen werden, um somit Rückschlüsse auf Korrektheit und Vollständigkeit der injizierten Fehler zu ziehen.

6 Evaluation

In diesem Kapitel geht es um die Evaluation des Werkzeugs `clang-sfi`. Dazu werden die Injektionen von `clang-sfi` mit denen von `SAFE` in das Open-Source Werkzeug `jsoncpp`[1] und in die selbst erstellte Datei `test.cpp`[2] verglichen. Dabei werden alle Patch-Files von Hand verglichen. Außerdem werden für die Datei `json_value.cpp` aus dem Projekt `jsoncpp` und die Datei `test.cpp` auch von Hand alle Injektionsstellen für das Fehlermodell aus Abschnitt 3 gesucht und das Ergebnis mit in den Vergleich aufgenommen.

Um den Aufwand zu verringern wird hier ein Werkzeug verwendet, um Patch-Files von `clang-sfi` den entsprechenden Patch-Files aus `SAFE` zuzuordnen. Dazu werden die Patch-Files anhand ihrer Fehlerklasse und der Zeilen, die durch das entsprechende Patch-File gelöscht werden, verglichen. Bei Übereinstimmungen werden diese einander zugeordnet und es müssen nur noch die einander zugeordneten miteinander verglichen werden, wodurch der Gesamtaufwand um ein wesentliches sinkt. Sollte es für ein Patch-File keine Entsprechungen durch das jeweils andere Werkzeug geben, so wird dieses Patch-File in einer Liste aufgeführt und es muss manuell entschieden werden, ob dies trotzdem eine korrekte Entscheidung des Werkzeugs war. Dabei ist aber auch zu beachten, dass, wie in Abschnitt 3 beschrieben, Initialisierungen in den beiden Werkzeugen unterschiedlich gesehen werden und somit Unterschiede in den Fehlerklassen MVIV, MVAV, MVAE und WVAV zu erwarten sind.

Daraus ergibt sich die Aufteilung dieses Kapitels in den Vergleich der beiden Werkzeuge und den von Hand gefundenen Injektionsstellen, wobei zuerst auf die nicht von beiden Werkzeugen gefundenen Injektionstellen, und dann auf die Unterschiede bei der Injektion, einschließlich der dabei entstandenen Syntaxfehler, eingegangen wird, und einem Fazit.

	A	B	C	D	E	F	G	H
1	Fehlerklasse	SAFE	clang-sfi	nur in		unterschiedliche	Syntaxfehler	
2				SAFE	clang-sfi	Stellen	SAFE	clang-sfi
3	MFC	61	54	9	2	11	1	0
4	MVIV	2	1	1	0	1		0
5	MVAV	0	3	0	3	3		0
6	WVAV	0	3	0	3	3		0
7	MVAE	11	14	0	3	3		0
8	MIA	54	18	40	4	44		1 (Präprozessor)
9	SMIA		62	0	8	8		
10	MIFS	55	15	40	0	40		0
11	SMIFS		55	0	0	0		0
12	MIEB	10	9	2	1	3	8	1 (Präprozessor)
13	SMIEB		10	0	0	0		
14	MLOC	15	15	0	0	0		0
15	MLAC	10	17	0	7	7		0
16	MLPA	57	64	0	6	6		0
17	WPFV	30	23	2	2	4		0
18	WAEP	34	16	6	2	8	2	0
19	Summe	339	252	100	33	133	11	2

Abbildung 6.1: Diese Tabelle zeigt eine Übersicht über die Anzahl an injizierten Fehlern, die Unterschiede der Injektionen von `clang-sfi` und `SAFE` und die durch die Injektion entstandenen Syntaxfehler. Injiziert wurde in die Datei `json_value.cpp` aus dem Projekt `jsoncpp`.

6.1 Evaluation der Ergebnisse der beiden Werkzeuge

In diesem Abschnitt werden die durch `SAFE` und `clang-sfi` injizierten Fehler in die Dateien `json_value.cpp` und `json_reader.cpp`[3] aus dem Projekt `jsoncpp`[4] und `test.cpp` aus dem Anhang jeweils miteinander, sowie mit den von Hand gefundenen Injektionstellen für die Dateien verglichen. Dabei wird auf die Unterschiede der beiden Ergebnisse für eine Datei in die injiziert wird und Abweichungen zu dem Fehlermodell aus Abschnitt 3, sowie auf die Unterschiede bei der Injektion und dabei aufgetretene Fehler, eingegangen.

6.1.1 Vergleich der Injektionsstellen

In diesem Abschnitt werden die Injektionsstellen, die aus der manuellen Analyse des Quelltextes der beiden genannten Dateien hervorgehen, mit den durch `SAFE`

[1] `https://github.com/open-source-parsers/jsoncpp` (abgerufen am 29.04.2018)
[2] Siehe Anhang.
[3] Bei dieser Datei werden nur die Ergebnisse der beiden Werkzeuge miteinander verglichen.
[4] Die Datei `json_writer.cpp` aus diesem Projekt konnte nicht mit in den Vergleich genommen werden, da `SAFE` bei dieser Datei einen Fehler ausgibt und somit keine Injektion möglich war.

und `clang-sfi` gefundenen verglichen, um somit später Rückschlüsse auf die Genauigkeit der Werkzeuge schließen zu können.

6.1.1.1 `jsoncpp`

Bei der Injektion in die Datei `json_value.cpp` aus dem Projekt `jsoncpp` sind, wie man in Abbildung 6.1 sehen kann, in jeder Fehlerklasse Unterschiede zwischen den beiden Werkzeugen, außer bei der Fehlerklasse MLOC[5]. Deshalb wird nun zuerst auf die Unterschiede, die bezüglich der einzelnen Fehlerklassen aufgefallen sind, eingegangen. Dazu muss zuerst erwähnt werden, dass in dieser Datei Präprozessordirektiven für bedingte Kompilierung vorhanden sind, welche compilerabhängig entscheiden, welcher Teil des Quelltextes kompiliert werden soll. Auf Grund dieser kann `SAFE` 7 Fehlerinjektionsstellen[6] nicht finden und bei einem `if`-Konstrukt wird aus der Sicht der beiden Werkzeuge jeweils eine andere Anweisung umschlossen. Außerdem gibt es bei den zu injizierenden Interfacefehlern zwischen den beiden Werkzeugen noch die Unterschiede, dass `clang-sfi` für jede Injektionsstelle, wie in Abschnitt 3 beschrieben, nur einen Fehler injiziert, und `SAFE` alle möglichen Fehler injiziert. Das bedeutet für die Fehlerklasse WAEP, dass für jeden arithmetischen Ausdruck nicht nur der letzte Operand entfernt wird, sondern für jeden ein eigener Fehler injiziert wird, und für die Klasse WPFV, dass jede passende Variable als Fehler injiziert wird.

Bei der manuellen Analyse des Quelltextes, hat sich ergeben, dass unter Berücksichtigung der selben Präprozessordirektiven, wie `clang-sfi`, alle Injektionsstellen gefunden wurden, die `clang-sfi` auch gefunden hat. Außerdem sind vier Stellen aufgefallen, welche wie eine Zuweisung aussehen, aber einem Aufruf eines überladenen Zuweisungsoperators entsprechen. Da die Möglichkeit diesen Operator zu überladen C++-spezifisch ist [9] und diese nicht dem ursprünglichen Fehlermodell aus `G-SWFIT` [6] entsprechen, wo nach einer `mov`-Anweisung im binären Code gesucht wird [6], werden diese hier nicht weiter betrachtet[7] und im Folgenden müssen nur noch die Unterschiede der Injektionsstellen der beiden Werkzeuge miteinander verglichen werden.

Bezüglich der Fehlerklassen, deren Injektionsstellen eine `if`-Anweisung betreffen, also MIA, MIFS und MIEB, entsprechen alle Fehler, die `SAFE` im Gegensatz zu `clang-sfi` gefunden hat, nicht der entsprechenden Fehlerklasse. Jede dieser Fehlerklassen verbietet das Injizieren, falls innerhalb der von der entsprechenden

[5]Die Fehlerklassen SMIA, SMIEB und SMIFS sind dabei nur für den Vergleich in `clang-sfi` erstellt worden. Bei diesen werden innerhalb der Anweisungen `return`-Anweisungen erlaubt und Deklarationen verboten.

[6]Diese Fehlerinjektionsstellen gehören zu den Klassen MFC (2 Fehler), MVAV (1 Fehler), WVAV (1 Fehler), MLPA (3 Fehler).

[7]Sollte man die Verwendung eines überladenen Zuweisungsoperators als Zuweisung sehen wollen, so müsste man einen entsprechenden `ASTMatcher` für überladene Operatoren mit Einschränkungen analog zu dem entsprechenden `ASTMatcher` für Zuweisungen hinzufügen.

`if`-Anweisung umschlossenen Anweisungen ein Sprung, mit Ausnahme eines Funktionsaufrufs, vorkommt. Bei diesen insgesamt 82 Injektionen, die bei `clang-sfi` nicht auftauchen, kommt jeweils eine `return`-Anweisung innerhalb der umschlossenen Anweisungen vor, welches einem Sprung entspricht und somit entsprechen diese injizierten Fehler nicht dem in Abschnitt 3 beschrieben Fehlermodell.

`clang-sfi` injiziert im Gegensatz zu `SAFE` bei diesen Klassen 5 weitere Fehler. Bei einem der für die Fehlerklasse MIEB injizierten Fehler, befindet sich eine Variablendeklaration innerhalb der umschlossenen Anweisungen, was dem Fehlermodell nicht widerspricht. Für die Fehlerklasse MIA werden von `SAFE` zwei Injektionsstellen nicht erkannt, welche keine Anweisungen umschließen, da sie einen leeren Block als Rumpf besitzen. Die anderen beiden von `SAFE` nicht erkannten Fehlerinjektionsstellen sind `if`-Konstrukte die jeweils nur einen Funktionsaufruf umschließen, welches auch dem Fehlermodell entspricht.

Für die Fehlerklasse MVIV findet `SAFE` eine Initialisierung innerhalb eines `for`-Konstruktes, welches durch das Fehlermodell aber ausgeschlossen wird. Bei den anderen Fehlerklassen der Zuweisungsfehler[8] findet `clang-sfi` 7 Injektionsstellen mehr, an denen `SAFE` nicht injiziert. 5 dieser entsprechen Zuweisungen zu einer einfach dereferenzierten lokalen Variable, bzw. einem Element eines lokalen Arrays. Da beides auf lokale auf dem Stack liegende Speicherstellen verweisen kann, sind diese Injektionsstellen mit dem Fehlermodell vereinbar. Die anderen beiden Injektionsstellen sind Zuweisungen zu einem Attribut eines Objekts, welches als Parameter übergeben wird.

Die 9 Injektionsstellen, die von `SAFE` für die Fehlerklasse MFC mehr gefunden wurden, entprechen nicht dem Fehlermodell, da der entfernte Funktionsaufruf entweder die einzige Anweisung im umgebenden Block ist, oder der die Injektionsstelle nicht einem Funktionsaufruf, sondern einem Aufruf eines überschriebenen Zuweisungsoperators entspricht.

Die Injektion von Fehlern der Fehlerklasse MLAC entfernt einen Operanden aus einer Konjunktion, welche als Bedingung für einen Sprung verwendet wird. Dazu zählen also die Bedingungen von `for`-, `while`-, `do-while`- und `if`-Konstrukten. `SAFE` jedoch beachtet die Bedingungen von Schleifen nicht. Deshalb injiziert `clang-sfi` 5 Fehler mehr, bei denen es sich um Injektionsstellen innerhalb der Bedingung von Schleifen handelt. Des Weiteren fehlt bei `SAFE` auch eine Injektionsstelle die ein `if`-Konstrukt betrifft. Neben dieser fehlen bei `SAFE` auch drei Injektionsstellen für die MLPA-Fehlerklasse, welche jeweils einer einzelnen zu entfernenden Zuweisung entsprechen.

Auf Seiten von `SAFE` werden bezüglich der Interfacefehler aus dem Fehlermodell insgesamt 8 Fehler injiziert, die nicht von `clang-sfi` injiziert werden. Diese entsprechen Injektionen in Operatoraufrufe[9]. Auf Seiten von `clang-sfi`, werden

[8]Dazu zählen die Fehlerklassen MVAV, MVAE und WVAV.

[9]Davon sind 6 Aufrufe des Arrayzugriffsoperators.

4 weitere dem Fehlermodell entsprechende Injektionen durchgeführt.

Bei dem Vergleich der Injektionen der beiden Werkzeuge in die Datei `json_reader.cpp` hat sich gezeigt, dass bei den Fehlerklassen, welche ein `if`-Konstrukt betreffen, also MIA, MIFS und MIEB, durch `SAFE` nicht nur injiziert wird, wenn sich innerhalb der umschlossenen Anweisungen eine `return`-Anweisung befindet, sondern auch wenn es sich um eine `break`-Anweisungen handelt. Beides widerspricht dem Fehlermodell, da innerhalb dieser Anweisungen keine Sprünge außer Funktionsaufrufen vorhanden sein dürfen.

Bei den Fehlerklassen der Zuweisungsfehler injiziert `clang-sfi` hier genauso wie in die Datei `json_value.cpp`, wenn in Attribute von lokalen Objekten geschrieben wird, sowie in einfach dereferenzierte lokale Variablen, bzw. lokal angelegte Arrays. Außerdem injiziert `clang-sfi` im Gegensatz zu `SAFE` auch Zuweisungsfehler bei denen Zuweisungen zu Parametern einer Funktion auftreten, welche zum Teil Referenzen entsprechen. Bezüglich der anderen Fehlerklassen gab es keine Auffälligkeiten, die nicht auch bei der Injektion in `json_value.cpp` aufgetreten sind.

6.1.1.2 Injektion in `test.cpp`

	A	B	C	D	E	F	G	H
1	Fehlerklasse	SAFE	clang-sfi	nur in		unterschiedliche	Syntaxfehler	
2				SAFE	clang-sfi	Stellen	SAFE	clang-sfi
3	MFC	9	9	0	0	0	0	0
4	MVIV	15	9	6	0	6	0	0
5	MVAV	7	9	0	2	2	0	0
6	WVAV	7	9	0	2	2	0	0
7	MVAE	4	4	0	0	0	0	0
8	MIA	6	3	3	0	3		0
9	SMIA		5	1	0	1	0	
10	MIFS	4	2	2	0	2		0
11	SMIFS		3	1	0	1	0	0
12	MIEB	2	2	0	0	0	1	0
13	SMIEB		2	0	0	0		
14	MLOC	2	2	0	0	0	0	0
15	MLAC	4	4	0	0	0	0	0
16	MLPA	53	53	0	0	0	0	0
17	WPFV	10	5	0	0	0	0	0
18	WAEP	0	0	0	0	0	0	0
19	Summe	123	111	11	4	15	1	0

Abbildung 6.2: Diese Tabelle zeigt eine Übersicht über die Anzahl an injizierten Fehlern, die Unterschiede der Injektionen von `clang-sfi` und `SAFE` und die durch die Injektion entstandenen Syntaxfehler. Injiziert wurde in die Datei `test.cpp` aus dem Anhang.

Bei der Injektion in die Datei `test.cpp` aus dem Anhang kommt es bei den Fehlerklassen MVIV, MVAV, WVAV, MIA und MIFS zu Unterschieden zwischen

den Injektionsstellen der beiden Werkzeuge, wie man in Abbildung 6.2 erkennen kann. Die Ergebnisse der manuellen Analyse entsprechen hier, ähnlich wie im letzten Abschnitt, vollständig den Injektionsstellen, in die `clang-sfi` injiziert hat. Somit müssen auch in diesem Abschnitt nur die Unterschiede zwischen den beiden Werkzeugen betrachtet werden.

Zuerst werden nun die 10 unterschiedlichen gefundenen Injektionstellen der Zuweisungsfehlerklassen[10] betrachtet. Dabei sind 6 Unterschiede der Injektionsstellen der Annahme geschuldet, dass Initialisierungen für `clang-sfi` nur während der Deklaration geschehen können[11] und bei `SAFE` immer die erste Zuweisung zu einer Variablen einer Initialisierung entspricht, was in Abschnitt 3 beschrieben wurde. Insgesamt findet `SAFE` in dieser Datei zwei Zuweisungen, die es als Initialisierungen für die Fehlerklasse MVIV ansieht. Dementsprechend sind diese zwei Injektionsstellen bei `clang-sfi` der Klasse MVAV zugeordnet und da alle MVAV-Injektionsstellen auch WVAV-Injektionsstellen sind, findet `clang-sfi` auch zwei weitere WVAV-Injektionsstellen. Die weiteren vier Injektionsstellen für die Fehlerklasse MVIV, die nur `SAFE` findet, befinden sich jeweils innerhalb eines `for`-Konstrukts und entsprechen somit nicht dem in dieser Ausarbeitung beschriebenen Fehlermodell.

Bei den Fehlerklassen MIA und MIFS kommt es insgesamt zu 5 Injektionsstellen, in die nur das Werkzeug `SAFE` injiziert. Diese entsprechen aber nicht dem Fehlermodell, da in den vom entsprechenden `if`-Konstrukt umschlossenen Anweisungen keine Sprünge außer Funktionsaufrufen erlaubt sind und an jeder dieser Injektionsstellen das entsprechende `if`-Konstrukt mindestens eine `break`- oder `return`-Anweisung umschließt.

6.1.2 Vergleich der Injektionen

In diesem Abschnitt werden die Injektionen durch `SAFE` und `clang-sfi` bezüglich des Fehlermodells aus Abschnitt 3 verglichen. Tabelle 6.1 stellt dabei eine Übersicht über die Injektionen der beiden Werkzeuge für die verschiedenen Fehlerklassen dar. Wie man in der Tabelle erkennen kann, entfernt `SAFE` bei den Fehlerklassen der Zuweisungsfehler die Anweisungen im Gegensatz zu `clang-sfi` nicht, sondern weist der Variable auf der linken Seite der Zuweisung sich selbst zu. Außerdem wurde bei der Injektion in `jsoncpp` und `test.cpp` auch noch beobachtet, dass gleichbleibende Teile von Anweisungen durch `SAFE` bei der Injektion in semantische Äquivalente umgeformt wurden. Dazu zählen

- Zugriffe auf Attribute eines Objekts innerhalb dessen Mitgliedsfunktion. Dabei wird auf das Attribut mittels Zeigerarithmetik durch Verwendung

[10] Diese entsprechen MVIV, MVAV, WVAV und MVAE.
[11] Siehe Abschnitt 3.

<table>
<tr><th>Fehlerklasse</th><th>Code</th><th>SAFE</th><th>clang-sfi</th></tr>
<tr><td>MFC</td><td>someFunction();</td><td>{};</td><td>;</td></tr>
<tr><td>MIA</td><td>if(condition)
doSth();</td><td>{doSth();;}</td><td>doSth();</td></tr>
<tr><td></td><td>if(condition)
{
doSth();
doSth();
}</td><td>{
doSth();
doSth();
}</td><td>{
doSth();
doSth();
}</td></tr>
<tr><td>MIEB</td><td>if(condition)
doSth();
else
doSth();</td><td>{doSth();;}</td><td>doSth();</td></tr>
<tr><td></td><td>if(condition)
doSth();
else
{doSth();}</td><td>{doSth();}}</td><td>doSth();</td></tr>
<tr><td>MIFS</td><td>if(condition)
doSth();</td><td>{}</td><td>;</td></tr>
<tr><td></td><td>if(condition)
{doSth();}</td><td>{}</td><td></td></tr>
<tr><td>MLPA</td><td>doSth();
doSth();
doSth();</td><td>{}</td><td>;</td></tr>
<tr><td>MVIV</td><td>int i = 1;</td><td>int i = i;</td><td>int i;</td></tr>
<tr><td>MVIV (nur `SAFE`)</td><td>i = 1;</td><td>i = (1);</td><td></td></tr>
<tr><td>MVAV</td><td>i = 1;</td><td>i = (i);</td><td>;</td></tr>
<tr><td>MVAE</td><td>i = (1+j);</td><td>i = i;</td><td>;</td></tr>
<tr><td>WVAV</td><td>i = 128;</td><td>i = (128)^0xFF;</td><td>i = 128^0xFF;</td></tr>
<tr><td></td><td>isA = true;</td><td>isA = (true)^0xFF;</td><td>isA = false;</td></tr>
<tr><td>WPFV</td><td>doSth(i, 1);</td><td>doSth(j, 1);</td><td>doSth(j, 1);</td></tr>
<tr><td>WAEP</td><td>doSth(i+j, 1);</td><td>doSth((i), 1);</td><td>doSth(i, 1);</td></tr>
<tr><td>MLAC</td><td>if(testSth(i) && testSth(j))</td><td>if((testSth)(i))</td><td>if(testSth(i))</td></tr>
<tr><td>MLOC</td><td>if(testSth(i) || testSth(j))</td><td>if((testSth)(i))</td><td>if(testSth(i))</td></tr>
</table>

Tabelle 6.1: Übersicht über die Injektionen für die Fehlerklassen aus Abschnitt 3 mit den Werkzeugen `SAFE` und `clang-sfi`, welche aus der Analyse der erstellten Patch-Files hervorgegangen sind.

```
void Path::addPathInArg(const JSONCPP_STRING& /*path*/,
                        const InArgs& in,
                        InArgs::const_iterator& itInArg,
                        PathArgument::Kind kind) {
  if (itInArg == in.end()) {
    // Error: missing argument %d

  } else {
    args_.push_back(**itInArg++);
  }}
}

void Path::invalidPath(const JSONCPP_STRING& /*path*/, int /*location*/) {
  // Error: invalid path.
}
```

```
void Path::addPathInArg(const JSONCPP_STRING& /*path*/,
                        const InArgs& in,
                        InArgs::const_iterator& itInArg,
                        PathArgument::Kind kind) {
  if (itInArg == in.end()) {
    // Error: missing argument %d
  } else if ((*itInArg)->kind_ != kind) {
    // Error: bad argument type
  } else {
    args_.push_back(**itInArg++);
  }
}

void Path::invalidPath(const JSONCPP_STRING& /*path*/, int /*location*/) {
  // Error: invalid path.
}
```

Abbildung 6.3: Hier wird die Anwendung eines für `json_value.cpp` aus dem Projekt `jsoncpp` durch `SAFE` erstellten Patchfiles der Fehlerklasse MIEB dargestellt. Auf der rechten Seite sieht man einen Auschnitt der origenalen Quelltextdatei. Auf der linken Seite sieht man denselben Teil der Quelltextdatei, nachdem das Patchfile angewendet wurde.

des `this`-Zeigers zugegriffen. So wird dann z.B. der Zugriff auf das `cstr_`-Attribut zu `( (* (this)).cstr_)`[12].

- Bei Aufrufen von Überschriebenen Operatoren wird die Infix-Notation in eine Präfixnotation aufgelöst. Dadurch wird dann z.B. `(*it).first == key` zu `((((((it).operator*) ()).first).operator==) ((key)))`[13].

- Viele Konstrukte werden mit teilweise mehreren Paaren von öffnenden und schließenden runden Klammern umschlossen, wie man z.B. in der Tabelle 6.1 bei der Fehlerklasse MLOC sehen kann.

Insgesamt treten bei der Injektion mit dem Werkzeug `SAFE` in `json_value.cpp` aus dem Projekt `jsoncpp` 11 und bei der Injektion mit `clang-sfi` zwei Syntaxfehler auf. Bei der Injektion in die Datei `test.cpp` aus dem Anhang tritt insgesamt nur ein Syntaxfehler bei der Injektion mit `SAFE` auf. Alle Syntaxfehler die bei `SAFE` auftreten entsprechen jeweils einer nicht mehr passenden Anzahl an öffnenden und schließenden Klammern[14]. Ein Beispiel für einen solchen Fehler bei einer MIEB-Injektion kann man in Abbildung 6.3 erkennen. Diese Beobachtung wurde bei MIEB-Injektionen, bei denen der `else`-Teil des betreffenden `if`-Konstrukts aus einem Block besteht, immer und bei MIEB-Injektionen gemacht, bei denen ein `if-else`-Konstrukt betroffen ist, bei welchem das `if`-Konstrukt in einem der `else`-Konstrukte einen Block umfasst[15], gemacht.

[12] Dies ist ein Auschnitt einer MLAC-Injektion mit `SAFE` in die Datei `json_value.cpp` aus dem Projekt `jsoncpp`.
[13] Dies ist ein Auschnitt einer MLAC-Injektion mit `SAFE` in die Datei `json_value.cpp` aus dem Projekt `jsoncpp`.
[14] Betroffen sind sowohl runde, als auch eckige, als auch geschweifte Klammern.
[15] Listing 6.1 ist ein Minimalbeispiel dafür.

```
if(false)
    doSth();
else if(true){
    doSth();
}
```

Listing 6.1: Minimalbeispiel um einen Syntaxfehler bei der Injektion der Fehlerklasse MIEB mit `SAFE` zu provozieren.

Bei `clang-sfi` entstehen die beiden Syntaxfehler jeweils an der selben Stelle im Quelltext, die in Listing 6.2 dargestellt wird.

```
if (removed)
#if JSON_HAS_RVALUE_REFERENCES
*removed = std::move(it->second);
#else
*removed = it->second;
#endif
```

Listing 6.2: Dieser Quelltext ist ein Ausschnitt aus der Datei `json_value.cpp` aus dem Projekt `jsoncpp`.

Die Stelle entspricht einem `if`-Konstrukt, welches genau eine Anweisung umfasst, welche aber durch ein Präprozesssor-If-Konstrukt bestimmt wird. Bei der Injektion wird nun neben dem `if`-Schlüsselwort und der Bedingung auch noch ein Teil des Präprozesssor-If-Konstrukts entfernt, wodurch dieses nicht mehr syntaktisch korrekt ist. Bei der Injektion der Fehlerklassen MIFS und MIA entsteht dann an der in Listing 6.2 gezeigten Quelltextstelle ein solcher Syntaxfehler.

6.2 Fazit

Insgesamt hat sich in diesem Kapitel durch die Injektion der beiden Werkzeuge `clang-sfi` und `SAFE` in das Projekt `jsoncpp` und die Datei `test.cpp` aus dem Anhang gezeigt, dass die Injektionen von `clang-sfi` dem zugrunde liegenden Fehlermodell aus Abschnitt 3 entsprechen, wohingegen `SAFE` teilweise stark davon abweicht indem Einschränkungen, die für die Injektionsstellen gelten müssen, nicht beachtet werden, wie z.B. dass bei der Fehlerklasse MIFS innerhalb der von einem `if`-Konstrukt umfassten Anweisungen keine Sprünge vorhanden sein dürfen und `SAFE` an diesen Stellen aber injiziert, wenn sich eine `break`- oder `return`-Anweisung innerhalb dieser Anweisungen befindet. Außerdem hat `clang-sfi` alle Fehlerinjektionsstellen gefunden, die bei der manuellen Analyse der Dateien `json_value.cpp` und `test.cpp` gefunden wurden und somit injiziert das in dieser Arbeit vorgestellte Werkzeug alle Fehler des Fehlermodells in diesen Dateien.

Bei der Injektion der Fehler der einzelnen Fehlerklassen hat sich herausgestellt, dass `SAFE` teilweise nicht nötige Umformungen von gleichbleibenden Quelltextstellen in semantische Äquivalente macht. Dies führt zu Fehlern, die beim Program-

mieren so nicht passieren würden. Außerdem kam es bei den Injektionen relativ häufig zu Syntaxfehlern, was zu nicht repräsentativen Fehlern führt, da diese Syntaxfehler spätestens vom Compiler ausgegeben werden. Bei `clang-sfi` kam es bei den in dieser Arbeit durchgeführten Injektionen auch zu zwei Syntaxfehlern, die aber nur unter sehr spezifischen Umständen auftreten. Denn damit diese auftreten, muss innerhalb einer Quelltextspanne, die bei der Injektion eines Fehlers entfernt werden soll, ein Teil einer Präprozessor-Konstruktion mit gelöscht werden. Sollte dabei ein komplettes Präprozessor-If-Konstrukt mit entfernt werden, kommt es nicht zu einem solchen Fehler. Neben den Syntaxfehlern muss hier noch erwähnt werden, dass `SAFE` nicht in jeden C++-Code Fehler injizieren kann. Ein Beispiel dafür ist, wie oben beschrieben, die Datei `json_writer.cpp` aus dem Projekt `jsoncpp`.

Wie in einigen Fehlerklassen beschrieben, entfernt `clang-sfi` Anweisungen, die in der entsprechenden Fehlerklasse fehlen. Dies führt dazu, dass kein unnötiger Quelltext entsteht. Im Gegensatz dazu entfernt `SAFE` bei den Fehlerklassen MVAV und MVAE die entsprechenden Anweisungen oder Anweisungsteile bei MVIV nicht, sondern weist der entsprechenden Variable sich selbst zu.

7 Gesamtfazit und Zusammenfassung

7.1 Zusammenfassung

Bei `Software Fault Injection (SFI)` werden Fehler in Software injiziert mit der Zielsetzung Bugs zu emulieren [5]. Da es bei der Komplexität von Software heutzutage quasi unmöglich ist, die Fehlerfreiheit eines Softwareprojekts zu garantieren, kann man z.B. fehlerhaften Zuständen zur Laufzeit durch die Implementierung von Fehlertoleranzmechanismen entgegenwirken [8, 5]. `SFI` kann dann verwendet werden um diese Fehlertoleranzmechanismen zu bewerten, Prognosen für worst-case Szenarien zu erstellen oder Maße für die Zuverlässigkeit von Systemen in Anwesenheit von Fehlern beim Dependability Benchmarking zu bewerten [5].

In dieser Ausarbeitung wurde das Werkzeug `clang-sfi` für `SFI` erstellt und durch einen Vergleich mit dem bereits bestehenden Werkzeug `SAFE` und einer manuellen Analyse evaluiert. Die Erstellung eines eigenen Werkzeugs ist nötig gewesen, da `SAFE` z.B. nicht auf aktuellen C++-Code angewandt werden konnte.

Das `clang-sfi` zu Grunde liegende Fehlermodell konnte größtenteils aus einer Studie von Duares [5] für die `G-SWFIT`-Technik übernommen werden. In dieser wurde ein für realistische Fehler in Software repräsentatives Fehlermodell vorgestellt, welches aus der Analyse von Open-Source Projekten hervorging. Es waren lediglich ein paar kleine Anpassungen nötig, da in der Studie [5] eine Injektion in binären Code und nicht in den eigentlichen Quelltext angestrebt wurde. Dies bringt zwar den Vorteil, dass für eine Fehlerinjektion der Quellcode der Software nicht benötigt wird, jedoch können gegenüber der Injektion in den Quelltext Ungenauigkeiten auftreten [3].

Bei dem Entwurf wurde entschieden, dass für das Abstrahieren von Quelltext `Clang` verwendet werden soll, da dieses C++-Frontend bereits häufig genutzt wird und somit gut getestet ist. Mit diesem kann unter Anderem aktueller C++-Code geparst werden und dessen `LibTooling`[1] bietet die Möglichkeit den `AST` von `Clang` innerhalb von einem Stand-Alone Werkzeug zu verwenden. Außerdem können mit Hilfe von `ASTMatchern`[2] Ausdrücke erstellt werden, welche Codestellen beschrei-

[1] `https://clang.llvm.org/docs/LibTooling.html` (abgerufen am 29.04.2018)
[2] `https://clang.llvm.org/docs/LibASTMatchers.html` (abgerufen am 29.04.2018)

ben. Dies vereinfacht die Suche innerhalb des `AST`.

Dieser `AST` wurde dann in der `FaultInjector`-Klasse von `clang-sfi` verwendet um Injektionsstellen für Fehler entsprechend des Fehlermodells zu finden. Um Konfigurierbarkeit und Erweiterbarkeit zu erreichen wurden die Unterklassen von `FaulInjector` für die Injektion von Fehlern je einer Fehlerklasse entworfen. Somit muss für die Konfigurierbarkeit des Fehlermodells nur die Liste an zu verwendenden `FaultInjectors` angepasst werden. Zum Erweitern des Fehlermodells muss entsprechen nur eine neue Subklasse erstellt werden.

Für die Speicherung der Fehler haben sich die `GNU DiffUtils` als nützlich gezeigt.

Zur Evaluation von `clang-sfi` wurden die Ergebnisse, also die Injektionsstellen und die Injektionen an sich, mit den Ergebnissen von `SAFE` und einer manuellen Analyse verglichen. Dabei hat sich herausgestellt, dass `clang-sfi` alle Injektionsstellen der manuellen Analyse gefunden hat. `SAFE` hingegen findet einige Injektionsstellen nicht und weicht von dem Fehlermodell aus `G-SWFIT` ab, indem einige Einschränkungen, wie z.B. dass für die Fehlerklasse MIFS auch injiziert wurde, wenn innerhalb der von dem entsprechenden `if`-Konstrukt umschlossenen Anweisungen `break`- oder `return`-Anweisungen vorhanden waren, obwohl Sprünge außer Funktionsaufrufen bei diesen verboten sind. Des Weiteren entfernt `clang-sfi` im Gegensatz zu `SAFE` Anweisungen immer, wenn die entsprechende Fehlerklasse dieses verlangt und gleichbleibender Quelltext wird nicht umgeformt.

Bei `SAFE` traten insgesamt relativ viele Syntaxfehler bei Injektionen auf. Bei der Injektion mit `clang-sfi` kam es aber auch zu 2 Syntaxfehlern, da ein Teil eines Präprozessor-If-Konstrukts mit entfernt wurde. Dieses Problem tritt aber, wie oben beschrieben, nur unter sehr spezifischen Umständen auf.

7.2 Fazit und Ausblick

Insgesamt kann man sagen, dass `clang-sfi` die in dieser Arbeit besprochenen Ziele erfolgreich umsetzt, denn `clang-sfi` ist

- quelloffen,
- konfigurierbar,
- das Fehlermodell ist erweiterbar,
- es ist möglich in aktuellen C++-Code Fehler zu injizieren, die dem Fehlermodell aus Abschnitt 3 entsprechen und
- es ist möglich die injizierten Fehler automatisch anzuwenden, das Projekt zu kompilieren und Tests durchzuführen.

`clang-sfi` bietet nun also jedem die Möglichkeit `SFI` bei seinen Projekten anzuwenden, ohne dabei auf nicht-freie und nicht-quelloffene Werkzeuge zurückgreifen zu müssen. Außerdem kann jeder dieses Werkzeug durch die Möglichkeiten der Konfiguration und Erweiterung an seine speziellen Bedürfnisse anpassen.

8 Anhang

8.1 test.cpp

```
int huhu(){
  int i=0;
  switch(i){
    case 0:
    {

    }
    break;
    case 1:
    break;
    case 2:
    break;
  }
  return 3;
}

class Testitest1{
  public:
  Testitest1(){}
  private:
  int test1(){return 3;}
};
struct{
  int a,b,c;
  union S{
    char x;
    int y;
  };
}a;

int _main(int argc, const char **argv, bool oO){
  55;
  int i=0;
  3;
  if(true)
    3;
  6;
  i=3;
  7;
```

```
40    return 12;
41  }
42
43  bool testitest(){
44    return false;
45  }
46  bool testitest(bool z){
47    char x = true;
48    _main(1, 0,z);
49    return false;
50  }
51  bool asfawgw = true;
52  bool x = testitest();
53  class Testitest{
54    public:
55    Testitest(){}
56    Testitest(bool xyz){
57      int x = 10;
58      for(int i = 0 ; i < 10 ; i++){
59        int j=9;
60        int z ;
61        z=200;
62        x=20;
63      }
64      for(int i = 0 ; i < 10 ; i++){
65        int j=9;
66      }
67      x=100;
68      for(x=101;x>100;x--);
69      bool a = testitest();
70      a = testitest();
71      bool b = true;
72      b = testitest();
73      bool c;
74      a=true;
75      c=true;
76      c = testitest();
77      testitest();
78      testitest() && testitest();
79      initialize("abc");
80      initialize(testitest()?"abc":"cde");
81
82      initialize(a,c);
83      initialize(a,true);
84      initialize(true,a);
85      initialize(a);
86      initialize(true);
87      if(testitest()&&testitest()||false)
88        initialize(testitest()&&testitest()||testitest()||false);
89      else
```

```
90         initialize(testitest()&&testitest()||testitest()||false,
               testitest()&&testitest()||testitest());
91     }
92     private:
93     void initialize(bool b){
94     }
95     void initialize(bool b, bool c){
96     }
97     void initialize(char* str){
98
99       if(str[0]==123)
100        return;
101    }
102    bool huhu(){
103      return testitest();
104      int a;
105
106    }
107    protected:
108    int length(char* str){
109      int l = 0;
110      char f;
111      f='c';
112      f='d';
113      int i = 0;
114      while(1){
115        if(str[i]!=0){
116          l++;
117        }else{
118          i++;
119          break;
120        }
121      }
122      f='a';
123      return l;
124    }
125  };
126
127  int main(int argc, const char **argv){
128    for(int i = 0; true ; i++){
129      3;
130      if(i==5)
131        break;
132    }
133    Testitest *t = new Testitest();
134    if(argc==2){
135      if(argv[1][0] == '2'){
136        int i = 2;
137        i++;
138        i = i-1;
```

```
139         for(int j=0;j<3;j++){
140             2;
141         }
142         3;
143     }else if(argv[1][0] == '3' && true)
144         11;
145   }
146
147   if(1){
148     2;
149     2;
150     2;
151     2;
152     2;
153   }
154
155
156   if(true){
157     return 0;
158   }
159   return 2;
160 }
```

Literatur

[1] *Clang 7 documentation*. https://clang.llvm.org/docs/index.html. Accessed: 2018-04-29.

[2] *clang: a C language family frontend for LLVM*. https://clang.llvm.org/. Accessed: 2018-04-09.

[3] D. Cotroneo und R. Natella. "Fault Injection for Software Certification". In: *IEEE Security Privacy* 11.4 (Juli 2013), S. 38–45. ISSN: 1540-7993. DOI: 10.1109/MSP.2013.54.

[4] *Dokumentation von Clang*. http://clang.llvm.org/doxygen/. Accessed: 2018-04-29.

[5] J. A. Duraes und H. S. Madeira. "Emulation of Software Faults: A Field Data Study and a Practical Approach". In: *IEEE Transactions on Software Engineering* 32.11 (Nov. 2007), S. 849–867. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.113.

[6] *Emulation of Software Faults: a Field Data Study and a Practical Approach. Appendix A - G-SWFIT fault emulation operators*. http://wpage.unina.it/roberto.natella/misc/fault-emulation-annex-e0849s.pdf. Accessed: 2018-03-20.

[7] *Extensible Markup Language (XML)*. https://www.w3.org/XML/Core/. Accessed: 2018-04-29.

[8] R. Natella u. a. "On Fault Representativeness of Software Fault Injection". In: *IEEE Transactions on Software Engineering* 39.1 (Jan. 2013), S. 80–96. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.124.

[9] *Programming Languages — C++*. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf. Accessed: 2018-03-26.

[10] *RDF 1.1 JSON Alternate Serialization (RDF/JSON). W3C Editor's Draft 07 November 2013*. https://dvcs.w3.org/hg/rdf/raw-file/default/rdf-json/index.html. Accessed: 2018-04-29.

Abbildungsverzeichnis

2.1 Vorgehen von SAFE bei der Fehlerinjektion [3] 4

3.1 Bei dem Operator für die Fehlerklasse MIEB werden aus einem `if-else`-Konstrukt das `if`-Konstrukt und das `else`-Schlüsselwort entfernt, sodass nur noch die Anweisungen des `else`-Blocks übrig bleiben [6]. 12
3.2 Bei dem Operator für die Fehlerklasse MIA wird bei einem `if`-Konstrukt das `if`-Schlüsselwort entfernt, sodass nur noch die Anweisungen innerhalb des Konstrukts übrig bleiben [6]. 13

4.1 Darstellung der Zusammenhänge eines `Clang`-Werkzeuges und der Umsetzung in `clang-sfi` 19
4.2 UML-Diagramm der `FaultInjector`-Klasse 21

6.1 Diese Tabelle zeigt eine Übersicht über die Anzahl an injizierten Fehlern, die Unterschiede der Injektionen von `clang-sfi` und `SAFE` und die durch die Injektion entstandenen Syntaxfehler. Injiziert wurde in die Datei `json_value.cpp` aus dem Projekt `jsoncpp`. . . 48
6.2 Diese Tabelle zeigt eine Übersicht über die Anzahl an injizierten Fehlern, die Unterschiede der Injektionen von `clang-sfi` und `SAFE` und die durch die Injektion entstandenen Syntaxfehler. Injiziert wurde in die Datei `test.cpp` aus dem Anhang. 51
6.3 Hier wird die Anwendung eines für `json_value.cpp` aus dem Projekt `jsoncpp` durch `SAFE` erstellten Patchfiles der Fehlerklasse MIEB dargestellt. Auf der rechten Seite sieht man einen Auschnitt der origenalen Quelltextdatei. Auf der linken Seite sieht man denselben Teil der Quelltextdatei, nachdem das Patchfile angewendet wurde. 54

Eidesstattliche Versicherung

______________________________ ____________________

Name, Vorname Matr.-Nr.

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

____

____

____

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

__________________________ _______________________

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

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

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

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

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____________________________ _________________________

Ort, Datum Unterschrift