technische universität
dortmund

Bachelorarbeit

Fehlerinjektion an Software-Schnittstellen mit AspectC++

Christoph-Cordt von Egidy
6. Juni 2017

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

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

Inhaltsverzeichnis

1. Einleitung **1**
- 1.1. Motivation . . . 1
- 1.2. Ziele . . . 2

2. Grundlagen **3**
- 2.1. Softwarefehler . . . 3
- 2.2. Fehlerinjektion . . . 4
  - 2.2.1. Fehlerinjektion an Software-Schnittstellen . . . 5
  - 2.2.2. Bisherige Ansätze . . . 5
- 2.3. AspectC++ . . . 7

3. Entwurf **9**
- 3.1. Anforderungen und Ziele . . . 9
- 3.2. Struktur . . . 10
- 3.3. Konfiguration . . . 10
  - 3.3.1. Software-Schnittstellen . . . 10
  - 3.3.2. Fehlerinjektionskampagnen . . . 11
- 3.4. Automatische Verarbeitung . . . 12
  - 3.4.1. Durchführen von Testläufen . . . 12
  - 3.4.2. Logging . . . 13
  - 3.4.3. Vergleich der Ergebnisse . . . 14

4. Implementierung **15**
- 4.1. Übersicht . . . 15
- 4.2. Konfiguration . . . 15
  - 4.2.1. Software-Schnittstellen . . . 16
  - 4.2.2. Fehlerinjektionskampagnen . . . 17
- 4.3. Zusammensetzen des Aspect-Headers . . . 18
- 4.4. Konfigurationsschnittstelle . . . 20
- 4.5. Logging . . . 21
- 4.6. Trigger . . . 24
- 4.7. Injektion von Fehlzuständen . . . 24
- 4.8. Kompilierung . . . 26
- 4.9. Durchführen der Experimente . . . 26
- 4.10. Vergleich der Ergebnisse . . . 26

5. Evaluation **27**
5.1. Getestete Software 27
5.2. Konfigurierbarkeit 28
5.2.1. Ansatzpunkte 28
5.2.2. Injizierbare Fehler 29
5.2.3. Trigger 32
5.3. Automatisierung 34
5.4. Performance 34
5.5. Plattformunabhängigkeit 36
5.6. Flexibilität 37
5.7. Nützlichkeit 38
5.7.1. Beispiel: Speicherreservierung in *jpeg-compressor* 38

6. Zusammenfassung und Ausblick **41**
6.1. Zusammenfassung 41
6.2. Mögliche Anwendungsfelder 41
6.3. Mögliche Erweiterungen 42
6.3.1. Parallelisierung der Experimente 42
6.3.2. Weitere Injektionsmodi 42
6.3.3. Erweiterte Injektion an Funktionsparametern 43
6.3.4. Mehrere aktive Schnittstellen 43
6.3.5. Unterstützung bei Konfiguration und Auswertung 43

Literaturverzeichnis **45**

Abbildungsverzeichnis **47**

Tabellenverzeichnis **49**

Listingverzeichnis **51**

A. *AOFIT* **53**

B. Konfigurationen **55**
B.1. *jpeg-compressor* 55
B.2. *lzham_codec* 58

1. Einleitung

Mit zunehmender Komplexität und dem Zurückgreifen auf die Funktionalität mehrerer Programmbibliotheken erhöht sich bei Software die Wahrscheinlichkeit, dass sie Fehler enthält oder nicht korrekt auf fehlerhaftes Verhalten von Komponenten reagiert, mit denen sie interagiert. Unerwünschtes Verhalten von Software aufgrund von Fehlern kann gerade in sicherheitskritischen Anwendungsbereichen schwerwiegende Folgen haben. Es ist daher unumgänglich, Software möglichst zuverlässig zu gestalten, also so, dass sie kontinuierlich die erwarteten Dienste erbringt und mit möglichem Fehlverhalten externer Komponenten adäquat umgeht.

Als Ergänzung zu Softwaretests gibt es dafür das Mittel der Fehlerinjektion. Diese bietet eine effiziente Möglichkeit, durch die gezielte Einbringung von Fehlern in Software und Softwarekomponenten deren Verhalten in kritischen Situationen zu analysieren. Auf Grundlage der Ergebnisse kann die Zuverlässigkeit der getesteten Software bewertet und verbessert werden.

1.1. Motivation

Von den vielen Bereichen der Fehlerinjektion befasst sich diese Arbeit mit der Injektion von Fehlzuständen an Software-Schnittstellen. Im Vergleich zu Injektionen von Quellcodeänderungen, Änderungen der kompilierten Software oder der Manipulation einzelner Bits im Speicher bietet diese Ebene die Möglichkeit, Unregelmäßigkeiten bei der Interaktion von Software und Softwarekomponenten direkt zu simulieren. Es müssen also keine Defekte simuliert werden, die nicht notwendigerweise zu Fehlzuständen und Fehlverhalten führen. Stattdessen können deren Auswirkungen direkt erzeugt werden.

Es existieren bereits viele verschiedene Ansätze, Fehlerinjektion an Software-Schnittstellen durchzuführen. Sie unterscheiden sich in ihrer Zielsetzung, der technischen Herangehensweise sowie dem gebotenen Funktionsumfang und der Flexibilität. Bisher ist allerdings noch nicht untersucht worden, ob, beziehungsweise wie gut, sich ein aspektorientierter Ansatz für diese Art der Fehlerinjektion eignet. Die Vermutung ist, dass dieser Vorteile bringt, da durch den direkten Zugriff auf den Quellcode der Schnittstellen mehr Kontrolle über die Injektionen ausgeübt werden kann.

1.2. Ziele

Diese Arbeit soll untersuchen, inwiefern ein aspektorientierter Ansatz für Fehlerinjektion an Software-Schnittstellen geeignet ist. Als Schnittstellen sollen nicht nur jene zwischen einer Software und von ihr verwendeten Programmbibliotheken angesehen werden, sondern ebenfalls Schnittstellen zwischen verschiedenen Komponenten einer Software. Dafür wird ein Konzept formuliert und implementiert, welches die Injektion von Fehlzuständen an Software-Schnittstellen von in C++ geschriebener Software mittels AspectC++ ermöglicht. Dieses soll unter anderem eine hohe Flexibilität und Automatisierung bieten und möglichst aussagekräftige Ergebnisse liefern. Anhand der Implementierung sollen die Vor- und Nachteile der Idee gezeigt und anschließend ein Ausblick auf mögliche Verbesserungen gegeben werden.

2. Grundlagen

Dieses Kapitel bietet eine Einführung in die Bereiche Softwarefehler und Fehlerinjektion. Anschließend wird das in der Implementierung verwendete AspectC++ erläutert.

2.1. Softwarefehler

An Software wird im Allgemeinen der Anspruch gestellt, zuverlässig zu sein. Dabei vereint der Begriff Zuverlässigkeit eine Menge von Eigenschaften wie Verfügbarkeit, Beständigkeit oder Sicherheit, was bedeutet, dass die Software stets bereit ist, ihre Dienste kontinuierlich zu erbringen, ohne dabei negative Auswirkungen auf den Anwender oder ihre Umgebung zu haben [2]. Die Gewährleistung dieser Eigenschaften kann durch eine Reihe von Faktoren auf verschiedenen Ebenen gefährdet werden. Neben Defekten in der Hardware oder negativen äußeren Einflüssen auf diese, bezeichnen Softwarefehler jene die Zuverlässigkeit gefährdenden Faktoren, die in der Software enthalten sind.

Unter dem Oberbegriff eines Fehlers wird zwischen den drei verschiedenen Begriffen Defekt (engl.: *fault*), Fehlzustand (engl.: *error*) und Fehlverhalten (engl.: *failure*) differenziert. Ein Defekt im Quellcode gelangt beim Übersetzungsvorgang in die Software beziehungsweise in eine ihrer Komponenten. Wird bei der Ausführung der Software die Stelle erreicht, deren Quellcode fehlerhaft ist, so wird der Defekt aktiviert und als Fehlzustand bezeichnet. Dieser zunächst software-interne Fehlzustand kann sich durch Interaktion zwischen Softwarekomponenten verbreiten (auch propagieren genannt) und so nach außen, also an Schnittstellen zu anderen Systemen oder Benutzern, sichtbar werden, indem das auftretende Verhalten vom Erwarteten abweicht. An dieser Stelle zeigt die Software ein Fehlverhalten [10]. Der Prozess vom Defekt über den Fehlzustand bis zum Fehlverhalten, veranschaulicht in Abbildung 2.1, wird als Fehlerkette bezeichnet [6].

Dadurch, dass vom Fehlverhalten sicherheitskritischer Software in Systemen wie beispielsweise in Verkehrsmitteln oder medizinischen Geräten eine große Gefahr ausgehen kann, ist es notwendig, Softwarekomponenten tolerant gegenüber Fehlzuständen zu gestalten, die sich zur Laufzeit möglicherweise über Schnittstellen zu anderen Softwarekomponenten, Funktionsbibliotheken, zum Betriebssystem oder zu Benutzern verbreiten. Die Mittel zum Erreichen dieses Ziels werden Fehlertoleranzalgorithmen und -mechanismen (engl.: *FTAMs*) genannt.

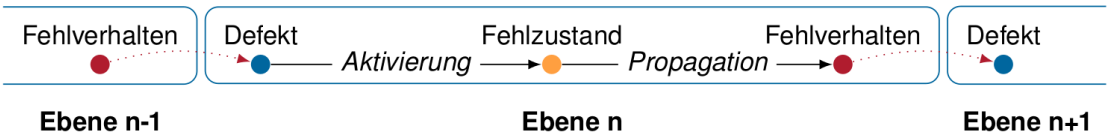


Abbildung 2.1.: Die Fehlerkette beschreibt die Entwicklung vom Defekt über den daraus entstehenden fehlerhaften Systemzustand bis zum von den Erwartungen abweichenden Systemverhalten. Die Ursache des Defekts kann auf einer niedrigeren Ebene liegen, zum Beispiel in der Hardware. Das Fehlverhalten kann zu nachfolgenden Ebenen propagieren [6, Abb. 2.4].

2.2. Fehlerinjektion

Fehlerinjektion bezeichnet die beabsichtigte Einbringung von Fehlern unterschiedlicher Art an verschiedenen Stellen auf der Hardware- oder Software-Ebene. Durch die Simulierung von Defekten oder Fehlzuständen werden Fehlerketten eingeleitet oder nachgeahmt [10]. Neben der physischen Simulation natürlicher Fehlerquellen auf Hardware-Ebene, bei der Systeme etwa ionischer Strahlung oder erhöhter Temperatur ausgesetzt werden, kann Fehlerinjektion auch mittels Software umgesetzt werden. Dabei wird entweder die einem System zugrunde liegende Hardware durch eine Software simuliert, welche mögliche Hardwarefehler nachahmt, oder die zu testende Software wird selbst manipuliert (engl.: *software-implemented fault injection*) [12]. Diese Arbeit behandelt ausschließlich die Injektion von Fehlern auf Software-Ebene.

Dabei kann an drei Stellen angesetzt werden. Um Defekte zu simulieren, können fehlerhafte Änderungen in den Quellcode eingebracht werden (engl.: *injection of code changes*). Während der Ausführung können Fehlzustände, für die üblicherweise Fehler in oder äußere Einflüsse auf Hardware verantwortlich sind, durch Manipulation von Speicherbereichen und Registern herbeigeführt werden (engl.: *injection of data errors*). Die dritte Methode setzt an den Schnittstellen zwischen Softwarekomponenten an (engl.: *injection of interface errors*) [10].

Üblicherweise werden unter einer definierten Arbeitslast (engl.: *workload*) mehrere Experimente ausgeführt, in denen jeweils ein Fehler injiziert wird, der aus einer vorgegebenen Menge stammt oder anhand von Regeln erzeugt wird. Ein Fehlermodell beschreibt dabei den Raum der möglichen Fehler unter drei Gesichtspunkten [10]:

- *Was:* Der Defekt oder Fehlzustand, welcher injiziert wird
- *Wann:* Der Zeitpunkt der Injektion
- *Wo:* Die Stelle innerhalb der untersuchten Software, an welcher injiziert wird

Zur Bewertung der Auswirkungen des injizierten Fehlers kann das Verhalten der getesteten Software in einem Experiment ohne injizierten Fehler (engl.: *golden run*) jeweils mit dem durch ein Fehlerinjektionsexperiment beeinflussten Verhalten verglichen werden. Die Durchführung mehrerer Fehlerinjektionsexperimente nennt man eine Fehlerinjektionskampagne.

Fehlerinjektion wird zusätzlich zu Softwaretests als Mittel zur Analyse, Einstufung und Verbesserung der Zuverlässigkeit von Software verwendet. Die Vorteile liegen darin, dass sie im Gegensatz zur Verifikation von Software auch in zunehmend komplexeren Systemen gut anwendbar ist [10]. Weiterhin bietet sie die Möglichkeit, das Verhalten von Systemen unter verschiedenen, realen Bedingungen entsprechenden, Arbeits- und Fehlerlasten zu bewerten und mögliche Schwachstellen zu identifizieren [14].

Die Anwendungsbereiche von Fehlerinjektion werden in die drei Kategorien Zuverlässigkeitsprognose, Zuverlässigkeitsbewertung und Verbesserung von Fehlertoleranzalgorithmen und -mechanismen unterteilt. Bei der Zuverlässigkeitsprognose werden die Fehlertoleranz von Systemen sowie die Verbreitung von Fehlzuständen analysiert und Wahrscheinlichkeiten für das Auftreten und die korrekte Behandlung von Fehlzuständen und Fehlverhalten gebildet. Zuverlässigkeitsbewertung bedeutet die Einstufung und den Vergleich der Zuverlässigkeit von Systemen unter Berücksichtigung mehrerer Perspektiven, wie beispielsweise Verfügbarkeit oder Stabilität. Zur Verbesserung von *FTAMs* wird schließlich ein besonderer Fokus auf ausführliche und qualitative Ergebnisse aus den Fehlerinjektionsexperimenten gelegt, um daraus Rückschlüsse auf die Ursachen von Fehlern zu ziehen [10].

2.2.1. Fehlerinjektion an Software-Schnittstellen

Bei der Fehlerinjektion an Software-Schnittstellen wird die Interaktion von Softwarekomponenten mit Systemen, Hardware, Benutzern oder anderen Softwarekomponenten manipuliert. Als Schnittstellen gelten zum Beispiel die Abfrage von Eingaben von Benutzern, die Funktionen einer von einer Software genutzten Bibliothek (*API*) oder die Funktionen einzelner Komponenten innerhalb der selben Software. An den Schnittstellen werden Fehlzustände simuliert, indem die während der Ausfürung auftretenden Ein- und Ausgabewerte ersetzt oder geändert werden. Dabei gibt es verschiedene Herangehensweisen, die anhand bisheriger Arbeiten zur Fehlerinjektion an Software-Schnittstellen erläutern werden sollen.

2.2.2. Bisherige Ansätze

Barton P. Miller, Lars Frederiksen und Bryan So testeten 1990 die Zuverlässigkeit knapp 90 bekannter Anwendungsprogramme für UNIX [9]. Mit den Hilfsprogrammen *fuzz* und *ptyjig* wurde an der Schnittstelle zum Benutzer angesetzt. Dabei generiert *fuzz* unter Berücksichtigung einiger Konfigurationsparameter zufällige

Eingabeströme, die über eine Pipe an die Standardeingabe der zu testenden Anwendungen weitergeleitet werden. Für interaktive Anwendungen wie den Texteditor `vi` emuliert *ptyjig* ein Terminal und kann in die Pipeline zwischen *fuzz* und der zu testenden Software eingereiht werden. Der Ansatz, den die Autoren selbst als naiv klingend bezeichnen, brachte jedoch ungefähr ein Viertel der getesteten Programme zum Absturz und ließ Rückschlüsse auf einige typische Programmierfehler zu.

Das Tool *FIG* bietet die Möglichkeit, die Wiederherstellungsmechanismen von Software bei Rückgabe von Fehlercodes der GNU C-Standardbibliothek `glibc` zu prüfen [4]. Dazu wird mithilfe der Umgebungsvariable `LD_PRELOAD` zuerst die dynamische Programmbibliothek `libfig.so` geladen, welche die Funktionen der C-Standardbibliothek überschreibt. Über Konfigurationsdateien sind die Funktionen, an denen Fehler injiziert werden sollen, sowie der Umfang der Protokollierung anpassbar. Unter den mit *FIG* getesteten Programmen wurden beispielsweise in Emacs und der Berkeley-Datenbank Schwachstellen im Umgang mit fehlschlagenden Speicherreservierungen, Lese- und Schreibvorgängen gefunden, die zu Abstürzen oder Datenverlust führten.

Einen technisch ähnlichen Ansatz befolgt das Fehlerinjektions-Framework Hovac, zielt dabei jedoch auf die Zuverlässigkeitsbewertung von Software ab, welche Programmbibliotheken von Drittanbietern benutzt [5]. Der bisher nur für Windows implementierte Ansatz lädt ebenfalls eine spezielle Programmbibliothek, welche die Funktionen der zu testenden Programmbibliothek überschreibt. Eine Fehlerinjektion ist hier sowohl vor dem Aufruf der originalen Funktion durch die Manipulation ihrer Argumente als auch danach durch die Manipulation ihres Rückgabewertes möglich.

Das Fehlerinjektions-Framework *LFI* setzt ebenfalls an der Schnittstelle zu dynamischen Programmbibliotheken an, indem es deren Rückgabewerte manipuliert, jedoch beschränkt es sich dabei nicht wie *FIG* auf die `glibc` [8]. Der enthaltene *LFI profiler* analysiert Programmbibliotheken und liefert die möglichen Rückgabewerte jeder exportierten Funktion, was dabei hilft, zu injizierende Fehler zu bestimmen. Auf dieser Grundlage werden automatisch zwei sogenannte Fehlerinjektionsszenarien generiert. Das erste beschreibt die Injektion von Fehlern bei jeder Nutzung von Funktionen einer analysierten Programmbibliothek, während das zweite die zu manipulierenden Funktionsaufrufe sowie den zu injizierenden Fehler anhand eines konfigurierbaren Wahrscheinlichkeitswertes auswählt. Die Fehlerinjektionsszenarien sind durch eine dafür entwickelte Sprache sowie durch die Möglichkeit, eigene Injektions-Trigger zu schreiben, präzise konfigurierbar.

2.3. AspectC++

AspectC++ erweitert die objektorientierte Programmiersprache C++ um das Konzept der aspektorientierten Programmierung. Innerhalb eines *Aspekts* wählen sogenannte *pointcut-expressions* eine Menge von *join points* aus, bei denen es sich um Stellen in gegebenem Quellcode handelt, an welchen mit *advices* zusätzliche Funktionalität eingefügt werden kann [13]. Diese *advices* können an verschiedenen Stellen im Kontrollfluss der Software ansetzen. Bezogen auf Funktionen sind das etwa die Stellen vor, nach oder anstatt eines Funktionsaufrufs oder einer Funktionsausführung.

Die Codezeile des *advices* in Listing 2.1 wird nach jedem Aufruf einer beliebigen Funktion der Klasse `Example` mit Rückgabewert vom Typ `bool` ausgeführt. Mithilfe des von AspectC++ bereitgestellten Objekts `tjp` wird die konkrete Signatur des jeweiligen *join points*, also der Funktion, die der *pointcut-expression* entspricht, ausgegeben. Über das Objekt `tjp` kann auch auf die Argumente einer Funktion sowie auf weitere Informationen wie etwa Dateinamen oder Zeilennummern zugegriffen werden.

Aspekte werden in Aspect-Header-Dateien abgelegt. Der AspectC++-Compiler webt ihren Inhalt an den festgelegten Stellen in den originalen Quellcode ein. Der entstehende gültige C++-Code wird anschließend mit einem üblichen C++-Compiler übersetzt.

```
1 aspect test {
2     advice call("bool Example::%(...)") : after() {
3         cout<<tjp->signature()<<" was called"<<endl;
4     }
5 };
```

Listing 2.1: Aspekt mit Advice-Code.

3. Entwurf

In diesem Kapitel wird der Entwurf zur Umsetzung von Fehlerinjektion an Software-Schnittstellen mit AspectC++ dargelegt. Zuerst werden die Anforderungen an den Entwurf festgelegt. Anschließend wird eine Übersicht über die Struktur des Entwurfs gegeben, welcher danach detailliert erläutert wird.

3.1. Anforderungen und Ziele

Das Ziel ist, eine konfigurierbare Möglichkeit zur Fehlerinjektion an Software-Schnittstellen unter Nutzung von AspectC++ zu bieten, die über einen hohen Automatisierungsgrad verfügt und aussagekräftige Ergebnisse liefert. Im Folgenden werden die Anforderungen und Ziele aufgelistet und erläutert:

Konfigurierbarkeit
: Es soll detailliert konfigurierbar sein, wann welche Fehlzustände wo injiziert werden.

Automatisierung
: Dem Anwender soll Arbeit abgenommen werden, indem ein möglichst hoher Grad an Automatisierung realisiert wird.

Performance
: Die Anzahl der Kompilierungsvorgänge soll minimiert werden, da diese zeitintensiv sind.

Plattformunabhängigkeit
: Die entwickelte Lösung soll flexibel einsetzbar und daher nicht auf eine spezifische Plattform zugeschnitten sein.

Flexibilität
: Der Einsatzbereich soll sich nicht auf einzelne Programme oder Programmbibliotheken beschränken.

Nützlichkeit
: Es sollen Ergebnisse geliefert werden, die zur Nachrüstung und der Verbesserung von Fehlertoleranzalgorithmen und -mechanismen eingesetzt werden können. Dafür müssen die Abläufe möglichst umfangreich protokolliert werden.

3.2. Struktur

Am Anfang steht die Konfiguration der Software-Schnittstellen und der Fehlerinjektionskampagne, die durch den Anwender vorgenommen wird. Die Umsetzung der Vorgaben aus der Konfiguration soll danach automatisiert erfolgen. Die Maßnahmen, welche die Fehlerinjektion realisieren, werden in einem Aspect-Header abgelegt und gelangen durch das Kompilieren in die zu testende Software. Mit dieser werden dann Testläufe ausgeführt, deren Ergebnisse mit einem Referenzergebnis verglichen werden. Dieses stammt aus einem Testlauf ohne injizierte Fehlzustände, dem *golden run*. Das Resultat des automatisierten Prozesses ist der Log, also die Protokollierung der durchgeführten Experimente und deren Auswirkungen, woraus der Anwender Rückschlüsse auf die von ihm getestete Software ziehen kann. Diese Struktur wird in Abbildung 3.1 veranschaulicht.

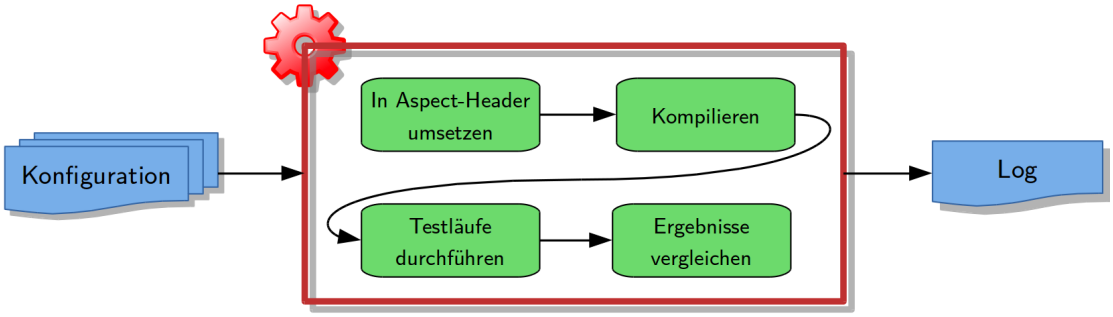


Abbildung 3.1.: Die Struktur des Entwurfs: Die Konfiguration wird automatisiert umgesetzt und erzeugt Ergebnisse in Form eines Logs.

3.3. Konfiguration

Die Konfiguration umfasst die Spezifizierung von Software-Schnittstellen und Fehlerinjektionskampagnen. Software-Schnittstellen legen dabei fest, an welcher Stelle Fehlzustände injiziert werden sollen, während Fehlerinjektionskampagnen eine Menge von Fehlerinjektionsexperimenten mit ihren Rahmenbedingungen beschreiben.

3.3.1. Software-Schnittstellen

Schnittstellen zwischen Softwarekomponenten und Betriebssystem sowie zwischen Softwarekomponenten untereinander werden durch die Bereitstellung und die Nutzung von Funktionen realisiert. Als Schnittstelle gilt im Folgenden also eine nicht leere Menge von Funktionen. Die Festlegung der Schnittstellen beantwortet die sich bei der Fehlerinjektion stellende Frage nach dem *Wo*.

Funktionen gleichen Namens können durch Variation der Anzahl und der Typen der Parameter überladen werden, weshalb es sinnvoll ist, in der Konfiguration einer Schnittstelle das Abdecken mehrerer Funktionen zu ermöglichen. Soll der Umgang mit der Rückgabe von Fehlercodes unabhängig davon, welche Funktion einer Softwarekomponente diese erzeugt, untersucht werden, so muss eine Schnittstellendefinition die Menge der abzudeckenden Funktionen auch anhand ihres Rückgabetyps darstellen können.

Beim Aufruf einer Funktion mit Parametern werden ihr Argumente übergeben. Nach der Ausführung können Funktionen Ergebnisse in Form von Rückgabewerten liefern. Dadurch ergeben sich für die Fehlerinjektion an einer Schnittstelle zwei mögliche Ansatzpunkte: die übergebenen Argumente sowie die zurückgegebenen Werte. Bei der Konfiguration einer Schnittstelle muss also angegeben werden, an welcher dieser Stellen Fehlzustände injiziert werden sollen.

In Schnittstellendefinitionen sollen bei der Angabe der Typen für Parameter und Rückgabewerte sowohl primitive Datentypen als auch in der untersuchten Software neu definierte Datentypen und Klassen angegeben werden können.

3.3.2. Fehlerinjektionskampagnen

Die Konfiguration einer Injektionskampagne spezifiziert die einzelnen Fehlerinjektionsexperimente und die dafür notwendigen Rahmenbedingungen auf Grundlage der zuvor spezifizierten Schnittstellen. Die Beschreibung der Fehlerinjektionsexperimente beantwortet die Fragen nach dem *Was* und *Wann*.

Für die Experimente einer Kampagne wird eine Arbeitslast festgelegt, unter der jedes einzelne Experiment durchgeführt wird. Ein Experiment gibt für eine Schnittstelle an, zu welchem Zeitpunkt welcher Fehlzustand an dieser injiziert werden soll. Der Zeitpunkt wird dabei durch sogenannte *Trigger* bestimmt, also Regeln, welche die Injektion eines Fehlzustands auslösen. Dafür wären etwa folgende Arten von Triggern denkbar:

Wahrscheinlichkeit
: Bei jeder Nutzung der spezifizierten Schnittstelle während eines Testlaufes wird der festgelegte Fehlzustand mit einer in der Kampagne anzugebenden Wahrscheinlichkeit injiziert.

Anzahl der Aufrufe
: Der festgelegte Fehlzustand wird periodisch nach einer in der Kampagne anzugebenden Anzahl von Aufrufen der Funktionen injiziert, welche durch die verwendete Schnittstelle beschrieben werden.

An jeder Möglichkeit
: Für jede unter der in der Fehlerinjektionskampagne festgelegten Arbeitslast

vorkommende Nutzung der im Experiment angegebenen Schnittstelle wird ein Unterexperiment durchgeführt, bei dem der festgelegte Fehlzustand bei der jeweils nächsten Nutzung injiziert wird. Für eine spezifizierte Schnittstelle wird also jede Möglichkeit zur Injektion des festgelegten Fehlzustands unter der gegebenen Arbeitslast genau einmal wahrgenommen.

Auf welche Art und Weise der zu injizierende Fehlzustand erzeugt wird, bestimmt der Injektionsmodus. Die folgende Beschreibung liefert Beispiele für mögliche Injektionsmodi:

Ersetzen
: Der vorliegende Wert wird durch einen im Experiment spezifizierten Wert ersetzt.

Invertieren
: Der vorliegende Wert wird invertiert, indem logische Werte negiert und bei Zahlenwerten das Vorzeichen gewechselt wird.

Verschieben
: Der vorliegende Zahlenwert wird um $+1$ oder -1 verändert.

Um das Kompilieren in die automatische Verarbeitung zu integrieren, muss der dazu notwendige Befehl hinterlegt werden. Zum Vergleich der Ergebnisse aus den Testläufen unter Fehlerlast mit dem Ergebnis aus dem *golden run* kann ein entsprechender Befehl angegeben werden.

3.4. Automatische Verarbeitung

Aus der Konfiguration der Schnittstellen und der Kampagne soll automatisch ein Aspect-Header generiert werden, der die Trigger, die Injektion von Fehlzuständen und das Logging implementiert. Zudem soll er eine Schnittstelle bereitstellen, über die diese Funktionen zur Laufzeit vor der Durchführung eines Experiments konfiguriert werden können.

Beim Kompilieren wird die Funktionalität des Aspect-Headers in die zu testende Software integriert. Es entsteht die im Weiteren zur Durchführung von Fehlerinjektionsexperimenten verwendete modifizierte Software.

3.4.1. Durchführen von Testläufen

Zu Beginn einer Kampagne wird zuerst ein *golden run* ausgeführt, in dem die modifizierte Software mit der festgelegten Arbeitslast, jedoch ohne die Injektion von Fehlzuständen gestartet wird. Anschließend wird für jedes in der Kampagne spezifizierte Experiment ein Testlauf ausgeführt. Dazu wird die modifizierte

Software mit der festgelegten Arbeitslast gestartet und über die zuvor integrierte Schnittstelle für das jeweilige Experiment konfiguriert.

3.4.2. Logging

Um den größtmöglichen Nutzen aus einer Fehlerinjektionskampagne zu ziehen, müssen die Abläufe und Ergebnisse detailliert protokolliert werden. Das bedeutet, die Umsetzung der in der Konfiguration spezifizierten Werte für das *Was*, *Wann* und *Wo* der Fehlerinjektion sowie ihre Auswirkung auf das Verhalten der Software im Log festzuhalten.

Dadurch, dass eine Schnittstelle mehrere Funktionen beschreiben kann, muss für die spätere Auswertung im Log aufgeführt werden, an welcher dieser Funktionen ein Fehlzustand des Experiments injiziert wird. Die entsprechende Funktion kann dabei über ihre Signatur, den Namen der Datei, in welcher sie implementiert ist und die Zeilennummer identifiziert werden, weshalb diese Informationen protokolliert werden sollten. Wichtig für die Beschreibung der Vorgänge ist ebenfalls das Aufführen des originalen Wertes, der anschließend durch die Injektion des Fehlzustands manipuliert wird.

Die Forderung der Konfigurierbarkeit soll auch für die Protokollierung gelten. Dabei soll es dem Anwender ermöglicht werden, die zu protokollierenden Informationen auszuwählen oder die Protokollierung komplett abzuschalten. Einen Kompromiss zwischen Konfigurierbarkeit und Komfort bieten dabei vordefinierte Stufen, wie sie in Tabelle 3.1 beschrieben sind. Ein Haken in der Spalte „Experimente“ bedeutet, dass die zuvor beschriebenen Informationen zu den durchgeführten Fehlerinjektionsexperimenten protokolliert werden. „Setup“ steht für sonstige Meldungen des automatisierten Ablaufes, die vor oder nach Fehlerinjektionsexperimenten auftreten können.

Stufe	**Zeitstempel**	**Experimente**	**PID & Thread**	**Setup**
0	–	–	–	–
1	✓	–	–	✓
2	–	✓	✓	✓
3	✓	✓	✓	✓

Tabelle 3.1.: Entwurf zur Konfiguration der Protokollierung anhand festgelegter Stufen

3.4.3. Vergleich der Ergebnisse

Nach jedem Testlauf soll dessen Ergebnis mit dem des *golden run* verglichen werden, um die möglichen Auswirkungen der Fehlerinjektion zu identifizieren. Dies kann durch die Ausführung eines in der Kampagne hinterlegten Vergleichsbefehls nach jedem Experiment erfolgen.

4. Implementierung

Dieses Kapitel beschreibt die Umsetzung des zuvor erläuterten Entwurfs. Dies umfasst den Aufbau der Konfigurationsdateien, Details zur implementierten Software sowie zu deren Nutzung. Diese Software wird im Folgenden *AOFIT* genannt, was für *Aspect-Oriented Fault Injection Tool* steht. Die Wahl der Programmiersprache ist auf Python gefallen, da es plattformunabhängig ist, das für die Konfiguration verwendete JSON nativ unterstützt und eine komfortable Möglichkeit zum Verwalten von Kindprozessen bietet. Die dafür verwendeten Funktionen setzen eine Python-Version ab 3.5 voraus.

4.1. Übersicht

AOFIT nimmt bei der Durchführung der automatisierten Abläufe eine zentrale Rolle ein. Nach der Generierung des Aspect-Headers aus den Informationen der Schnittstellen- und Kampagnenkonfiguration initiiert es das Kompilieren der Software. Mit der entstandenen modifizierten Software werden zunächst der *golden run* und anschließend die in der Kampagne festgelegten Experimente durchgeführt. Nach jedem Experiment wird dessen Ergebnis durch das vom Anwender angegebene Programm mit dem Referenzergebnis des *golden run* verglichen. Während der Durchführung der Experimente werden Details zu den Injektionen protokolliert. Die Ausgaben des nach jedem Experiment ausgeführten Vergleichsprogramms werden anschließend ebenfalls dem Protokoll hinzugefügt. Diese Abläufe und Zusammenhänge sind in Abbildung 4.1 visualisiert.

4.2. Konfiguration

Die Konfiguration wird mit zwei Konfigurationsdateien vorgenommen: eine beschreibt die Software-Schnittstellen, eine die Fehlerinjektionskampagne. Die Trennung in zwei Dateien ergibt Sinn, da eine Software-Schnittstelle in beliebig vielen verschiedenen Experimenten verwendet werden kann. Auf Basis einer Schnittstellenkonfiguration können so auch mehrere Kampagnen angelegt werden. Zusätzlich fördert die Abgrenzung der Schnittstellen von den darauf basierenden Kampagnen die Übersicht.

Als Datenformat wurde JSON gewählt, da es eine gute Lesbarkeit bietet und von den Standardbibliotheken vieler Programmiersprachen, darunter auch Py-

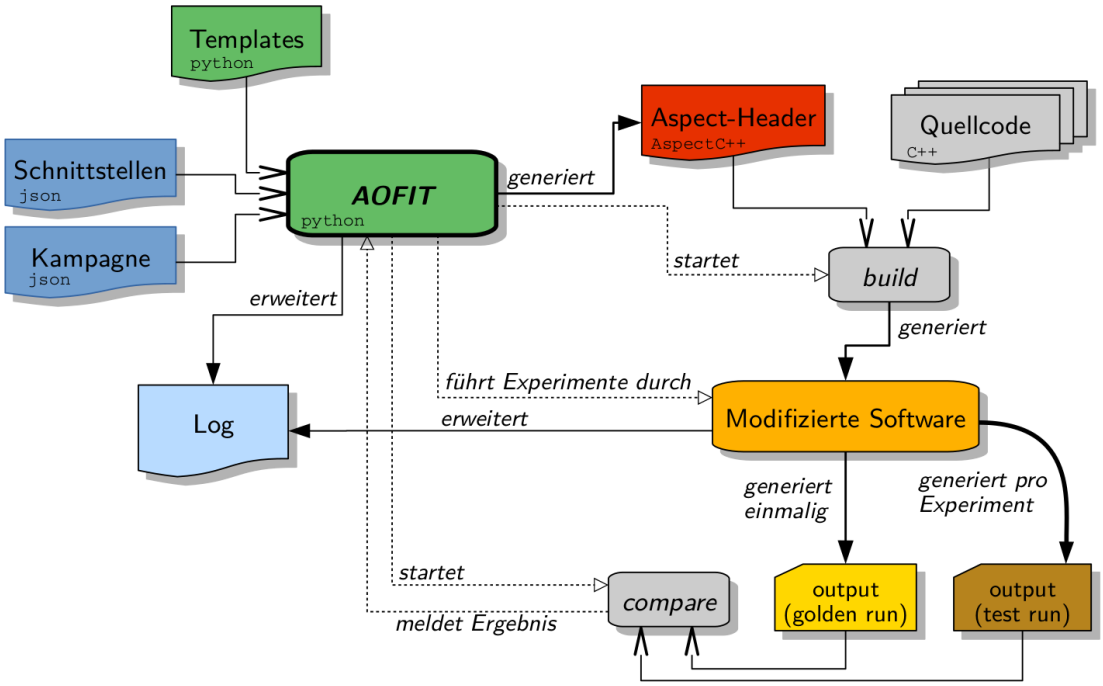


Abbildung 4.1.: Struktur der Implementierung und der automatisierten Umsetzung der Konfiguration

thon, unterstützt wird. Zur Beschreibung der Strukturen der Konfigurationsdateien wurden Schemata angelegt, anhand derer die Dateien vor der Auswertung von *AOFIT* validiert werden können, sofern das Paket *jsonschema*[1] installiert ist.

4.2.1. Software-Schnittstellen

Die Definition einer Software-Schnittstelle erfolgt über die Attribute *namespace*, *className*, *funcName* und *injectAt*. Das Nichtangeben eines oder mehrerer der ersten drei Attribute wird als Wildcard interpretiert. Abhängig davon, ob mit *injectAt* der Rückgabewert oder Argumente als Injektionsziel festgelegt werden, muss deren Typ mit *resultType* beziehungsweise *argType* angegeben werden. Alternativ kann die Funktionsmenge einer Schnittstelle mit dem Attribut *pointCutExp* direkt per *pointcut-expression* beschrieben werden, wobei die Angabe der Informationen für *injectAt* und *resultType* beziehungsweise *argType* trotzdem erforderlich ist. Zur Verwendung in der Kampagne sowie zur Identifikation im Protokoll muss für jede Schnittstelle über das Attribut *id* ein eindeutiger Name festgelegt werden. Da dieser auch für die in Abschnitt 4.3 beschriebene Generierung von Wertevektoren verwendet wird, darf er nur aus Zeichen bestehen, die in C++-Bezeichnern erlaubt sind.

[1] `https://pypi.python.org/pypi/jsonschema/2.6.0`

Die Definition von Software-Schnittstellen wird im Folgenden anhand zweier Beispiele veranschaulicht, die Teil der in Kapitel 5 erläuterten Konfiguration sind.

Listing 4.1 zeigt die Definition der Schnittstelle `primitive_bool_argument`. Das dadurch beschriebene Injektionsziel ist das erste boolesche Argument der Funktion `code_block` der Klasse `jpeg_encoder` im Namespace `jpge` und ihrer möglichen Überladungen. Details zur Auswahl der Argumente sind in Abschnitt 4.7 beschrieben.

Ein Beispiel für die Definition von Schnittstellen allein über den Rückgabewert zeigt Listing 4.2. Die Schnittstelle `enum_result` umfasst alle Funktionen im Namespace `jpgd`, die einen Wert vom Typ `jpgd_status` zurückgeben, wobei es sich um eine Enumeration von Erfolgs- und Fehlercodes handelt. Diese können so innerhalb eines Experiments unabhängig von spezifischen Funktionen injiziert werden.

In der Konfigurationsdatei können beliebig viele Software-Schnittstellen definiert werden.

```
{
    "id": "primitive_bool_argument",
    "interface": {
        "namespace": "jpge",
        "className": "jpeg_encoder",
        "funcName": "code_block",
        "argType": "bool"
    },
    "injectAt": "argument"
}
```

Listing 4.1: Definition der Schnittstelle `primitive_bool_argument`

```
{
    "id": "enum_result",
    "interface": {
        "namespace": "jpgd",
        "resultType": "jpgd::jpgd_status"
    },
    "injectAt": "result"
}
```

Listing 4.2: Definition der Schnittstelle `enum_result`

4.2.2. Fehlerinjektionskampagnen

Jede Fehlerinjektionskampagne wird in einer eigenen Konfigurationsdatei abgelegt. Diese enthält die Rahmenbedingungen der Kampagne und beliebig viele Ex-

perimente. Die Attribute der Rahmenbedingungen erklärt Tabelle 4.1. Dabei ist *experiments* eine Liste von Fehlerinjektionsexperimenten, von denen jedes einzelne gemäß Tabelle 4.2 aufgebaut ist.

Listing 4.3 zeigt zur Veranschaulichung die Rahmenbedingungen und ein Experiment der in Kapitel 5 vorgestellten Konfiguration. Das im Experiment vorkommende Attribut *num* dient der Übersichtlichkeit und wird von *AOFIT* nicht verarbeitet.

Attribut	**Typ**	**Zweck**
customIncludes	Array	Liste von Headerdateien, die zur Auflösung eventuell neu definierter Datentypen und Enumerationen benötigt werden
buildCommand	Array	Befehl zum Kompilieren der Software (jedes Element ist ein Argument)
programPath	String	Pfad zur modifizierten Software
goldenRunParams	Array	Liste der Parameter für den *golden run*, über die dessen Ergebnis gesondert von den Ergebnissen der Experimente abgelegt wird
workloadParams	Array	Liste der Parameter für die Experimente
comparisonCommand	Array	Befehl und Argumente zum Vergleich der Ergebnisse
logLevel	Int	Logging-Stufe
experiments	Array	Liste der Experimente (Struktur siehe Tabelle 4.2)

Tabelle 4.1.: Struktur der Konfigurationsdatei einer Kampagne

4.3. Zusammensetzen des Aspect-Headers

AOFIT generiert aus einer angegebenen Konfiguration, bestehend aus Schnittstellen- und Kampagnen-Konfigurationsdatei, einen Aspect-Header, der die Möglichkeit, Fehlzustände zu injizieren, beim Kompilieren in die zu testende Software integriert. Die Basis für die Generierung liefert eine Reihe von generischen Code-Fragmenten mit Platzhaltern, die im Folgenden als Templates bezeichnet werden. Diese werden anhand der Informationen aus den Konfigurationsdateien spezialisiert und kombiniert.

In ein Aspect-Header-Template werden für jede in der Konfiguration festgelegte Schnittstelle zwei Blöcke Advice-Code eingefügt, die auf dem Template zur Injektion an Rückgabewerten oder an Argumenten basieren. Dabei wird je ein Advice mit dem Wahrscheinlichkeits- und der andere mit dem Aufrufzahltrigger

Attribut	Typ	optional	Zweck
target	String	nein	Die verwendete Schnittstelle
injection_mode	String	nein	Modus der Fehlerinjektion: *replace*, *offset+*, *offset-* oder *invert*
errorValue	String	ja	Der beim Injektionsmodus *replace* zu injizierende Fehlzustand als C++-Code
callCount	Int	ja	Periode für den Aufrufzahltrigger
probability	Int	ja	Wert für den Wahrscheinlichkeitstrigger
seed	Int	ja	Seed für den Pseudo-Zufallszahlen-Generator des Wahrscheinlichkeitstriggers
each_occurrence_once	bool	ja	Schalter für das Durchführen von Unterexperimenten für jede Schnittstellennutzung

Tabelle 4.2.: Struktur eines Fehlerinjektionsexperiments

versehen, also durch die entsprechenden Templates aus Listing 4.4 ersetzt. Die *pointcut-expressions* der Advices werden aus den in der entsprechenden Schnittstellendefinition angegebenen Informationen zum *Wo* zusammengesetzt. Ist dort bereits eine *pointcut-expression* angegeben, wir diese übernommen.

Für jede Schnittstelle wird dem Aspekt ein `std::vector` hinzugefügt, der alle in der Kampagnenkonfiguration für die entsprechende Schnittstelle angegebenen Fehlzustände enthält. Der Name des Vektors bildet sich aus dem Namen seiner zugehörigen Schnittstelle und wird bei der Generierung des Aspect-Headers in die entsprechenden Advices eingesetzt. Dies ermöglicht es, in der Konfiguration der Kampagne beliebig viele Fehlzustände anzugeben, die beim Kompilieren in der modifizierten Software gespeichert werden. Zur Laufzeit kann daraus dann der für das jeweilige Experiment eingetragene Fehlzustand ausgewählt und injiziert werden. Listing 4.5 zeigt zur Veranschaulichung einen solchen Vektor für eine Schnittstelle namens `primitive_charArray_argument`, welcher zwei Werte vom Typ `const char*` speichert.

Weitere Informationen zur Generierung der Advices sind die Typen der Argumente oder Rückgabewerte sowie eine eindeutige Nummer, welche zur Prüfung auf Aktivierung, zum Auslesen der Konfiguration der Trigger und des zu injizierenden Fehlzustands sowie für die Zählung der Aufrufe einer Schnittstelle verwendet wird. Diese Nummer wird, bei 0 beginnend, fortlaufend für jeden generierten Advice vergeben.

Listing 4.6 zeigt einen von *AOFIT* generierten Advice. Dieser basiert auf dem

```
{
    "customIncludes": ["jpge.h", "jpgd.h"],
    "buildCommand": ["make"],
    "programPath": "./encoder",
    "goldenRunParams": ["images/ship.png", "images/ship_golden_50.jpg", "50"],
    "workloadParams": ["images/ship.png", "images/ship_experiment_50.jpg",
 ↪  "50"],
    "comparisonCommand": ["images/comp_jpeg.py"],
    "logLevel": 3,

    "experiments": [
        {"num": 3, "target": "primitive_bool_argument", "injection_mode":
 ↪  "replace", "errorValue": "false", "probability": 5, "seed": 24317}
    ]
}
```

Listing 4.3: Beispiel für eine Fehlerinjektionskampagne (gekürzt)

```
trigger_probabilities = """if(std::rand()%100 < probabilities[$id]) {"""
trigger_callCounts = """callCounts[$id]++;
      if(callCounts[$id]==callCountLimits[$id]) {"""
trigger_callCounts_reset = """if(!each_occurrence_once) callCounts[$id] =
 ↪  0;"""
```

Listing 4.4: Templates für die Trigger. Das Dollarzeichen leitet den Bezeichner eines Platzhalters ein, der im Laufe der Generierung des Aspect-Headers von *AOFIT* ersetzt wird.

Template für die Injektion an Rückgabewerten und wurde mit dem Aufrufzahltrigger versehen. Die *pointcut-expression* sowie die gelb hinterlegten Stellen sind durch die Spezialisierung des Templates anhand der Informationen der Schnittstellen- und Kampagnenkonfiguration entstanden.

Das Template zur Injektion an Argumenten, dessen Aufbau dem des Templates zur Injektion an Rückgabewerten ähnelt, findet sich im Anhang in Listing A.3.

Der Aspect-Header wird in einer Datei mit dem Namen der Konfigurationsdatei der Schnittstellen und der Endung `.ah` im Arbeitsverzeichnis von *AOFIT* abgelegt.

4.4. Konfigurationsschnittstelle

Damit die Anzahl der Kompiliervorgänge, wie in Kapitel 3 gefordert, minimal bleibt, muss möglichst viel zur Laufzeit konfigurierbar sein. Dazu wird über den Aspect-Header eine Konfigurationsschnittstelle in die zu testende Software integriert, die 11 zusätzliche Parameter zur Verfügung stellt.

Dies wird durch einen *around*-advice für die `main`-Methode realisiert. Dieser ver-

```
1 std::vector<const char*> valueVector_primitive_charArray_argument =
  ↪  {"nil.0", (const char[]){'n', 'i', 'l', '.', '1'}, };
```

Listing 4.5: Beispiel eines Vektors, der zwei zu injizierende Werte speichert

arbeitet zunächst die Informationen der zusätzlichen Parameter und konfiguriert die Fehlerinjektionsumgebung für das aktuelle Experiment. Anschließend zieht er die Anzahl der Parameter der Konfigurationsschnittstelle vom Parameter `argc` ab und führt die originale `main`-Methode per `tjp->proceed()` aus. Diese wird nun aufgrund der Information über die Anzahl der Argumente nicht auf die erweiterten Konfigurationsdaten in `argv` zugreifen.

Da der Anwender mit dieser Schnittstelle zwischen *AOFIT* und der modifizierten Software nicht unmittelbar in Berührung kommt, verzichtet sie auf eine komfortable Gestaltung. Für jeden Parameter muss ein Argument angegeben werden und die Reihenfolge ist statisch. Die folgende Liste beschreibt für jede Position nach den regulären Argumenten die Parameter der Konfigurationsschnittstelle:

1. Beschreibung des Experiments
2. Liste aktiver Advices
3. Konfiguration des Wahrscheinlichkeitstriggers
4. Konfiguration des Aufrufzahltriggers
5. Nummer zur Auswahl des zu injizierenden Fehlzustands
6. Logging-Stufe
7. Logging-Pfad
8. Schalter für *each_occurrence_once*
9. Injektionsmodus
10. Name der aktiven Schnittstelle
11. Seed für den Pseudo-Zufallszahlen-Generator

Beispiele für die Nutzung der Schnittstelle bieten Listing 4.7 und 4.8, die auf der Kampagne in Listing B.2 basieren. Hier wird das bereits durch *AOFIT* modifizierte Programm `encoder` aufgerufen und über die ersten drei Argumente mit der Arbeitslast konfiguriert. Ab dem vierten Argument folgen dann die Informationen, welche von der Konfigurationsschnittstelle verarbeitet werden.

4.5. Logging

Für die Protokollierung wird *spdlog*[2] verwendet. Dabei handelt es sich um eine plattformunabhängige und schnelle Logging-Bibliothek, deren Verhalten detail-

[2] `https://github.com/gabime/spdlog`

```
1  advice call("int jpgd::...::decode(...)") : after() {
2    occurrences[ 5 ]++;
3    if(activeTargets[ 5 ]) {
4      int* result = tjp->result();
5      callCounts[5]++;
6      if(callCounts[5]==callCountLimits[5]) {
7        if(injection_mode != "replace") {
8          inject(tjp, result);
9        } else {
10         if(*result != valueVector_primitive_int_result [valueID[ 5 ]]) {
11           log(tjp->filename(), tjp->signature(), tjp->line(), *result,
 ↪    valueVector_primitive_int_result [valueID[ 5 ]]);
12           fileLogger->flush();
13           *result = valueVector_primitive_int_result [valueID[ 5 ]];
14         } else {
15           log(tjp->filename(), tjp->signature(), tjp->line(), *result,
 ↪    valueVector_primitive_int_result [valueID[ 5 ]], "value planned to inject
 ↪    equals result");
16         }
17       }
18       if(!each_occurrence_once) callCounts[5] = 0;
19     }
20   }
21 }
```

Listing 4.6: Generierter Advice-Code für die Injektion an Rückgabewerten mit dem Aufrufzahltrigger. Gelb hinterlegte Stellen und die *pointcut-expression* sind durch den Austausch von Platzhaltern des Advice-Code-Templates entstanden.

liert konfigurierbar ist. Zudem besteht sie nur aus Header-Dateien und kann so unkompliziert eingebunden werden.

Wie in der Übersicht in Abbildung 4.1 zu sehen, entsteht eine Log-Datei, die aus zwei Quellen gespeist wird. *AOFIT* fügt nach jedem Experiment den Return-Code der modifizierten Software sowie die Standard- und Fehlerausgabe des ausgeführten Vergleichsbefehls ein. Während jedes Experiments protokolliert die modifizierte Software unter Beachtung der festgelegten Logging-Stufe Details zu den injizierten Fehlzuständen und der benötigten Laufzeit. Dafür wird die in Listing 4.9 aufgeführte Template-Funktion `log` verwendet, die als Argumente den Dateinamen, die Signatur und die Zeilennummer des *join points* sowie jeweils eine Referenz auf den ursprünglichen und den durch die Injektion veränderten Wert an der Schnittstelle erwartet und daraus einen Logeintrag generiert. Eine Überladung ermöglicht das Hinzufügen eines individuellen Texts, was im Injektionsmodus *replace* benutzt wird, um anzuzeigen, dass der an der Schnittstelle vorliegende Wert bereits dem zu injizierenden Fehlzustand entspricht.

```
./encoder images/ship.png images/ship_golden_50.jpg 50 'Golden Run' -1 _ _ _
↪  3 batest_cp.log 0 <none> <none> 0
```

Listing 4.7: Aufruf des modifizierten Programms '`encoder`' für den *golden run*

```
./encoder images/ship.png images/ship_experiment_50.jpg 50 'Experiment 13' 8
↪  8:1 _ 8:0 3 batest_cp.log 0 offset- primitive_int_argument2 2925
```

Listing 4.8: Aufruf des modifizierten Programms '`encoder`' für Experiment 13

Der Container `std::vector` besitzt für boolesche Werte aus Gründen effizienterer Speichernutzung eine Spezialisierung, die ein abweichendes Verhalten zeigt. Für die von der Funktion `log` erwarteten Referenzen liefert `std::vector<bool>` für den veränderten Wert eine Referenz vom Typ `std::vector<bool>::reference`, welche sich vom Typ der Referenz auf den originalen Wert unterscheidet. Daher existiert für beide Überladungen der Funktion `log` eine Spezialisierung für Aufrufe mit booleschen Werten, welche die spezielle Referenz per *typecast* in einen booleschen Wert konvertiert und der generischen `log`-Funktion übergibt. Eine davon ist in Listing 4.9 zu sehen.

```
template <typename T> void log(string filename, string sig, int line, const T&
↪  val_original, const T& val_changed) {
 if(logLevel >= 2) {
   std::ostringstream strm;
   strm << "function: '" << sig << "' in file '" << filename << "' in line "
↪  << line << ":"<<std::endl;
   strm << std::boolalpha << "  " << val_original << " ---> " << val_changed
↪  << std::endl;

   fileLogger->info("[injection] "+strm.str());
 }
}
template <typename T> void log(string filename, string sig, int line, const T&
↪  val_original, const std::vector<bool>::reference val_changed) {
 log(filename, sig, line, val_original, static_cast<const
↪  bool>(val_changed));
}
```

Listing 4.9: Die Template-Funktion `log` und die notwendige Spezialisierung für Referenzen vom Typ `std::vector<bool>::reference`

4.6. Trigger

Ein Trigger prüft bei jeder Nutzung einer im aktuellen Experiment aktiven Schnittstelle, ob eine Injektion ausgelöst werden soll. Im Folgenden wird beschrieben, wie die verschiedenen Trigger-Arten implementiert sind.

Der Wahrscheinlichkeitstrigger vergleicht eine mit `std::rand`, dem Pseudo-Zufallszahlen-Generator der C++-Standardbibliothek, erzeugte und per Modulo-Operator auf das Intervall $[0, 99]$ umgerechnete Zahl mit dem für das Experiment festgelegten Wahrscheinlichkeitswert.

Der Aufrufzahltrigger erhöht bei jeder Nutzung der Schnittstelle, für die er eingerichtet ist, einen Zähler und löst aus, sobald die festgelegte Anzahl an Nutzungen erreicht ist. In diesem Fall wird der Zähler zurückgesetzt, damit der Trigger periodisch auslösen kann.

Für die Unterexperimente des Experimentmodus *each_occurrence_once* wird der Aufrufzahltrigger verwendet. Im *golden run* wird für jede Schnittstelle gezählt, wie oft diese genutzt wird. Die Ergebnisse werden in einer Datei namens *occurrences* im Arbeitsverzeichnis von *AOFIT* abgelegt. In einem Experiment im Modus *each_occurrence_once* führt *AOFIT* für die angegebene Schnittstelle, die im *golden run* n-fach verwendet wurde, n Unterexperimente durch. Der Aufrufzahltrigger wird dabei mit den Werten 1 bis n konfiguriert. Da ein periodisches Auslösen dieses Triggers hier nicht erwünscht ist, wird das Zurücksetzen des Zählers unterlassen.

4.7. Injektion von Fehlzuständen

Ein bei der Generierung des Aspect-Headers erstellter Advice prüft in jedem Experiment beim Erreichen des *join points*, an dem er ansetzt, ob er als aktiv konfiguriert ist und führt in diesem Fall die Injektion und das Logging durch, sofern der enthaltene Trigger auslöst.

Die Injektion eines Fehlzustands wird durch die Manipulation eines Arguments oder des Rückgabewertes einer Funktion realisiert. Einen Zugriff darauf in Form eines Pointers erhält man mithilfe der vom Objekt `tjp` bereitgestellten Funktionen `result` und `arg`. In der Schnittstellenkonfiguration wird das Argument über seinen Typ definiert, während die Funktion `arg` das Argument, auf welches sie einen Pointer zurückgibt, anhand seiner Position in der Signatur auswählt. Zur Ermittlung dieses Indexes wird die im Aspekt-Template enthaltene Funktion `checkArgs` (Listing A.2) verwendet. Mithilfe der zur Kompilierzeit feststehenden Konstante `JoinPoint::ARGS`, welche die Anzahl der Argumente der Funktion eines *join points* enthält sowie mit dem Typen des gesuchten Arguments wird das rekursive struct-Template `ArgChecker` (Listing A.1) instanziiert. Der erste Index, welcher für ein Argument des gesuchten Typs steht, wird an die höheren Instanzen weiter-

gegeben und schließlich von `checkArgs` zurückgegeben. Dieses Vorgehen orientiert sich an dem Aspect-Header „Trace.ah“ aus dem Test „StaticTrace“, der Teil des Quellcodepakets des AspectC++-Compilers ist.

Die Injektion des Fehlzustands kann auf zwei Arten erfolgen. Im Injektionsmodus *replace* wird über das fünfte Argument der Konfigurationsschnittstelle der zu injizierende Fehlzustand aus dem `std::vector` der entsprechenden Schnittstelle ausgewählt. Dieser wird per Zuweisung injiziert, sofern er nicht bereits dem aktuellen Zustand entspricht.

Für die übrigen Injektionsmodi wird die Template-Funktion `inject` verwendet. Diese bietet vier Spezialisierungen, die Folgendes gewährleisten:

1. *invert* bildet bei booleschen Werten die Negation und wechselt bei Zahlenwerten das Vorzeichen.
2. *invert* kann auf Zahlenwerte nur angewendet werden, wenn deren Typ vorzeichenbehaftet ist.
3. *offset+*, *offset-* und *invert* können nur auf Werte arithmetischen Typs angewendet werden.
4. *offset+* und *offset-* können nicht auf boolesche Werte angewendet werden.

Die Abgrenzung zwischen den einzelnen Spezialisierungen erfolgt dabei mithilfe des seit dem C++-11-Standard existierenden `struct`-Templates `enable_if`. Dieses bietet eine komfortable Möglichkeit, über die Ausnutzung von Template-Substitutionsfehlern Spezialisierungen über boolesche Ausdrücke vorzunehmen. Listing 4.10 zeigt die mit dieser Technik durch die Template-Funktion `inject` erfolgende Prüfung auf vorzeichenbehaftete und arithmetische Typen.

```
1 //for boolean values: invert only
2 template<class JP> void inject(JP *tjp, bool *location) {...}
3 //arithmetic and signed: offset+, offset- and invert allowed
4 template <class JP, typename T, typename
   ↪  std::enable_if<std::is_signed<T>::value && std::is_arithmetic<T>::value,
   ↪  bool>::type = false> void inject (JP *tjp, T *location) {...}
5 //arithmetic and unsigned: only offset+ and offset- allowed
6 template<class JP, typename T, typename
   ↪  std::enable_if<!std::is_signed<T>::value && std::is_arithmetic<T>::value,
   ↪  bool>::type = false> void inject(JP *tjp, T *location) {...}
7 //non-arithmetic: no injection-mode but replace allowed
8 template<class JP, typename T, typename
   ↪  std::enable_if<!std::is_arithmetic<T>::value, bool>::type = false> void
   ↪  inject(JP *tjp, T *location) {...}
```

Listing 4.10: Nutzung von `enable_if` zur Maßschneiderung der Spezialisierungen der `inject`-Funktion

4.8. Kompilierung

Im Anschluss an die Generierung des Aspect-Headers wird das Kompilieren der zu testenden Software angestoßen, indem der dazu in der Kampagnenkonfiguration angegebene Befehl ausgeführt wird. Es wird vorausgesetzt, dass der dadurch initiierte Prozess den AspectC++-Compiler verwendet und die zuvor generierte Aspect-Header-Datei einbezieht. Durch die Verwendung von *spdlog* und `enable_if` sowie durch die Art, wie die `std::vector`-Instanzen gebildet werden, müssen der AspectC++- sowie der Backend-Compiler den C++11-Standard verwenden. Die Logging-Bibliothek erfordert je nach verwendeter Plattform ein Linken gegen die Bibliothek `pthread`. Weiterhin müssen sich die Header-Dateien, in denen *spdlog* implementiert ist, in einem vom Präprozessor durchsuchten Verzeichnis befinden.

Für die Konfiguration zur Laufzeit muss in der originalen Software eine `main`-Funktion mit den entsprechenden Funktionsparametern für die Anzahl und den Inhalt der Argumente vorhanden sein. Dies ist üblicherweise für alle Progamme der Fall, die über die Konsole bedient werden können.

Ist der Rückgabecode des ausgeführten Kompilierbefehls nicht 0, so wird davon ausgegangen, dass das Kompilieren fehlgeschlagen ist. Die Kampagne kann daher nicht ausgeführt werden und *AOFIT* beendet sich mit einer Fehlermeldung.

4.9. Durchführen der Experimente

Die in der Kampagnenkonfiguration festgelegten Experimente werden in der Reihenfolge ihres Auftretens nacheinander durchgeführt. Dabei wird für jedes Experiment die modifizierte Software gestartet und über ihre Parameter konfiguriert. Während die Arbeitslast in jedem Experiment gleich bleibt, werden die Details des jeweiligen Experiments und des zuerst durchgeführten *golden runs* über die Konfigurationsschnittstelle übermittelt, die der Software durch das Einweben des Aspekts hinzugefügt wurde.

4.10. Vergleich der Ergebnisse

Der in der Kampagnenkonfiguration im Attribut *comparisonCommand* hinterlegte Befehl wird im Anschluss an jedes Experiment ausgeführt. Die Wahl dieses Befehls obliegt dem Anwender, da nur dieser weiß, in welcher Form die Ergebnisse der von ihm getesteten Software vorliegen und nach welchen Kriterien sie verglichen werden sollen. Überlegungen, den Komfort an dieser Stelle zu verbessern, finden sich in Kapitel 6.

5. Evaluation

In diesem Kapitel wird die Implementierung auf die Erfüllung der in Kapitel 3 formulierten Anforderungen untersucht.

5.1. Getestete Software

Für die Bewertung der Eigenschaften von *AOFIT* wurde es an den zwei in C++ geschriebenen Projekten *jpeg-compressor*[1] und *lzham_codec*[2] getestet.

Jpeg-compressor verfolgt das Ziel, Bilddateien verschiedener Formate mit dem besten Verhältnis von Bildqualität zu Dateigröße ins jpeg-Format zu konvertieren, ohne dabei auf Geschwindigkeit oder benötigten Speicher zu achten. Der Grad der Kompression ist dabei ganzzahlig zwischen 1 (niedrigste Bildqualität) und 100 (höchste Bildqualität) wählbar.

Lzham_codec bietet verlustfreie Kompression beliebiger Daten mit ähnlichen Kompressionsraten wie das von Igor Pavlov entwickelte *LZMA*[3] bei höherer Kompressionsgeschwindigkeit. Die Kompressionsstufen sind ganzzahlig zwischen 0 für höchste Geschwindigkeit und 4 für höchste Kompression wählbar.

Beide Projekte enthalten ein Hilfsprogramm, welches eine Benutzerschnittstelle für die gebotene Funktionalität bereitstellt. Bei *jpeg-compressor* heißt dieses Programm `encoder`, bei *lzham_codec* `lzhamtest`. Die Arbeitslast lässt sich anhand von Eingabedateien und der Wahl des Kompressionsgrades für jeden Testlauf klar definieren. Als Eingabe für die durchgeführten Kampagnen wurde für *jpeg-compressor* die projekteigene Datei `ship.png` und für *lzham_codec* der AspectC++-Compiler `ac++` in der Version 2.2 verwendet.

Die von den Hilfsprogrammen erzeugten Ergebnisse liegen in Form von Dateien vor und lassen sich daher gut vergleichen, wofür zwei Vergleichs-Skripte geschrieben wurden. Aus der vom Hilfsprogramm `encoder` während eines Experiments generierten Bilddatei und jener aus dem *golden run* wird mit `compare`[4] eine dritte Datei generiert, die deren Unterschiede visualisiert. Die mit `lzhamtest` komprimierten Dateien werden auf ihre Größe im Vergleich zum Ergebnis des *golden run* untersucht.

[1] `https://github.com/pornel/jpeg-compressor`
[2] `https://github.com/richgel999/lzham_codec`
[3] `http://7-zip.org/sdk.html`
[4] `https://www.imagemagick.org/script/compare.php`

Für jedes Projekt wurde je eine Schnittstellenkonfiguration und eine Kampagne angelegt, die in Anhang B zu finden ist. Die Experimente in den Kampagnen sind zur besseren Übersicht mit einem zusätzlichen `num`-Attribut nummeriert.

Am Projekt *jpeg-compressor* wird die Injektion an Argumenten und Rückgabewerten getestet. Dabei werden Fehlzustände mit primitiven und per `typedef` neu definierten Datentypen sowie in Form von Feldern, Objekten und Werten aus Enumerationen injiziert. Dabei sorgt die Verwendung des Aufrufzahltriggers, der mit dem Wert 1 konfiguriert wird, dafür, dass das entsprechende Experiment auf jeden Fall zu mindestens einer Injektion führt, sofern die verwendete Schnittstelle unter der angegebenen Arbeitslast genutzt wird. Die Injektion von Fehlzuständen an Schnittstellen zu externen Programmbibliotheken wird anhand der Speicherallokationsfunktion `malloc` aus der C++-Standardbibliothek getestet.

Das Projekt *lzham_codec* wird zur Überprüfung der Anforderung verwendet, Fehlzustände auch an Schnittstellen zwischen einzelnen Komponenten der gleichen Software injizieren zu können. Dafür wurden vier Schnittstellen innerhalb des Moduls `lzhamcomp` definiert, die in jeweils einem Experiment verwendet werden.

Damit *lzham_codec* unter den in Abschnitt 4.8 beschriebenen Voraussetzungen kompiliert, muss in der Header-Datei `lzham_types.h` des Moduls *lzhamdecomp* dafür gesorgt werden, dass die `#undef`-Anweisungen zu Beginn auf jeden Fall ausgeführt werden. Seit dem C++11-Standard existieren diese Konstanten bereits im Header `<cstdint>` und führen so bei erneuter Definition zu Kompilierfehlern. Ausgenommen davon sind die Makros UINT*_MIN.

5.2. Konfigurierbarkeit

Die Konfiguration bietet dem Anwender umfangreiche Möglichkeiten bei der Gestaltung von Fehlerinjektionskampagnen und der Definition der ihnen zugrunde liegenden Software-Schnittstellen. Die folgenden drei Abschnitte analysieren die Ausprägung dieser Möglichkeiten im Bezug auf das *Wo*, *Was* und *Wann* der Fehlerinjektion.

5.2.1. Ansatzpunkte

Wie gefordert ist es möglich, Fehlzustände sowohl an Argumenten als auch an Rückgabewerten von Funktionen zu injizieren. Die Menge der Funktionen, welche eine Software-Schnittstelle umfasst, ist dabei präzise durch die Angabe von Namespaces, Klassen- und Funktionsnamen sowie Rückgabe- oder Parametertypen einschränkbar. Dabei kann man entweder wie beispielsweise in der Schnittstelle `primitive_bool_result` einzelne Funktionen (und ihre möglichen Überladungen), oder eine womöglich größere Funktionsmenge anhand weniger restriktiver Schnittstellendefinitionen beschreiben. Als Beispiel dafür gilt die Schnittstelle

`enum_result`, welche alle Funktionen im Namespace `jpgd` beschreibt, die einen Rückgabewert vom Typ `jpgd_status` haben. In diesem konkreten Beispiel existiert allerdings nur eine davon repräsentierte Funktion.

Als Alternative zu den dafür bereitgestellten Attributen kann auch direkt eine *pointcut-expression* angegeben werden. Diese ermöglicht zusätzlich die genauere Beschreibung von Funktionsparametern, was genutzt werden kann, um eine spezielle Überladung einer Funktion auszuwählen. Ein Beispiel bietet die Definition der Schnittstelle `malloc`, bei der das `pointCutExp`-Attribut verwendet wird. Die Angabe des Rückgabetyps ist für die Speicherung der Fehlerwerte im Aspect-Header trotzdem erforderlich.

Durch diese Form der Konfiguration kann an zwei Arten von Schnittstellen angesetzt werden. Zum einen sind dies Schnittstellen zwischen der getesteten Software und von ihr genutzten externen Programmbibliotheken, also beispielsweise der Standard-C++-Bibliothek, wie im Beispiel in Abschnitt 5.7.1. Zum anderen ist es auch möglich, an Schnittstellen zwischen verschiedenen Komponenten der selben Software anzusetzen, ohne dass diese in Form von dynamisch gelinkten Bibliotheken vorliegen müssen. Dies wird anhand der Software *lzham_codec* mit der in Anhang B.2 aufgeführten Konfiguration verdeutlicht.

Die konfigurierten Schnittstellen setzen an den öffentlichen Funktionen `init` und `put_bytes` der Klasse `lzcompressor` an, die im Modul `lzhamcomp` verwendet werden. In den Experimenten 1 und 4 wird durch die Manipulation von Rückgabewerten getestet, wie der aufrufende Code mit möglicherweise fehlerhaften Ergebnissen dieser Schnittstellen umgeht. In den Experimenten 2 und 3 wird hingegen das Verhalten des aufgerufenen Codes bei manipulierten Eingabewerten untersucht.

Listing 5.1 zeigt das gekürzte Protokoll des Experiments 2. Der Parameter `buf_len` der Funktion `put_bytes` wurde hier mit einer Wahrscheinlichkeit von 20% um den Wert 1 erhöht. Die Software terminiert zwar ohne Fehlercode, liefert allerdings eine Datei, die gut 10 Kilobytes größer als das Ergebnis des *golden run* mit 5,4 Megabytes ist. Das manuelle Dekomprimieren des Testergebnisses mit einer unmodifizierten Version von `lzhamtest` brach mit folgender Meldung ab:

```
Error: Decompressor wrote too many bytes to destination file!
```

Die entpackte Datei war 18 Bytes größer als der als Eingabe verwendete Aspect-C++-Compiler `ac++` und ließ sich im Gegensatz zu diesem nicht mehr als Programm ausführen.

5.2.2. Injizierbare Fehler

Bei der Fehlerinjektion an Software-Schnittstellen werden Fehlzustände injiziert. Die Fehlzustände können in der implementierten Lösung zum einen durch feste Werte angegeben werden, die in der Konfiguration der Kampagne hinterlegt wer-

```
[24.05.2017 17:01:45.241112] [Experiment 2] [pid 18211, thread 18211] [setup] LogLevel: 3.
 ↪  Experiment begins.
[24.05.2017 17:01:45.241153] [Experiment 2] [pid 18211, thread 18211] [setup] target
 ↪  configuration:
target         probability   callCount     occurrence    injection mode
-----------------------------------------------------------------------
put_bytes_len  20            0             0             offset+

[24.05.2017 17:01:45.389653] [Experiment 2] [pid 18211, thread 18211] [injection] function:
 ↪  'bool lzham::lzcompressor::put_bytes(const void *,unsigned int)' in file
 ↪  'lzhamcomp/lzham_lzcomp.cpp' in line 249:
262144 ---> 262145
[...]
[24.05.2017 17:01:54.500953] [Experiment 2] [pid 18211, thread 18211] [setup] main function
 ↪  terminated with result 0
[24.05.2017 17:01:54.500957] [Experiment 2] [pid 18211, thread 18211] [setup] runtime: 9.253834
 ↪  seconds

[24.05.2017 17:01:54.500959] [Experiment 2] [pid 18211, thread 18211] [setup] Experiment ends.
#=============================================================================================
 ↪  ======#

Experiment 2: program returned 0
comparison command returned 0 with output:
======
output is 10749 bytes bigger

======
```

Listing 5.1: Manipulation des Parameters `buf_len`

den und im Injektionsmodus *replace* den Inhalt des Arguments beziehungsweise des Rückgabewertes ersetzen. Dies ermöglicht die Injektion von Werten aller in C++ enthaltenen primitiven Datentypen sowie darauf basierenden neu definierten Typen und Enumerationen. Zum anderen können sie sich aus der Manipulation der zum Zeitpunkt der Injektion vorliegenden Zustände ergeben, was durch die Injektionsmodi *offset-*, *offset+* und *invert* ermöglicht wird.

5.2.2.1. Injektionsmodus *replace*

Listing 5.2 zeigt einen Auszug aus dem Log der Testläufe mit *jpeg-compressor*, basierend auf der Konfiguration aus Anhang B.1, der die Injektion des booleschen Wertes `false` am Rückgabewert der Funktion `read_image` protokolliert. In Listing 5.3 wird in einem Experiment der selben Kampagne an einem Funktionsargument vom Typ `int` der Wert 0 injiziert.

Der Umgang mit per `typedef` definierten Typen und Werten aus Enumerationen wird in den Experimenten 17 bis 20 getestet.

Es ist ebenfalls möglich, Felder und Objekte zu injizieren. In Experiment 16 (Listing 5.4) wird ein Feld mit `char`-Werten an einem Funktionsargument injiziert. Experiment 15 nutzt an der selben Schnittstelle die für Zeichenketten erlaubte Notation mit Anführungszeichen.

```
1 [24.05.2017 16:53:19.136461] [Experiment 1] [pid 17441, thread 17441] [setup] LogLevel: 3.
   ↪  Experiment begins.
2 [24.05.2017 16:53:19.136493] [Experiment 1] [pid 17441, thread 17441] [setup] target
   ↪  configuration:
3 target                probability   callCount     occurrence    injection mode
4 ------------------------------------------------------------------------------
5 primitive_bool_result  0             1             0             replace
6
7 [24.05.2017 16:53:19.139389] [Experiment 1] [pid 17441, thread 17441] [injection] function:
   ↪  'bool jpge::jpeg_encoder::read_image(const unsigned char *,int,int,int)' in file
   ↪  'jpge.cpp' in line 1044:
8        true ---> false
```

Listing 5.2: Injektion an booleschem Rückgabewert

```
1 [24.05.2017 16:53:30.413501] [Experiment 9] [pid 17491, thread 17491] [setup] LogLevel: 3.
   ↪  Experiment begins.
2 [24.05.2017 16:53:30.413543] [Experiment 9] [pid 17491, thread 17491] [setup] target
   ↪  configuration:
3 target                  probability   callCount     occurrence    injection mode
4 --------------------------------------------------------------------------------
5 primitive_int_argument  0             1             0             replace
6
7 [24.05.2017 16:53:30.415166] [Experiment 9] [pid 17491, thread 17491] [injection] function:
   ↪  'void jpge::image::subsample(jpge::image &,int)' in file 'jpge.cpp' in line 948:
8        2 ---> 0
```

Listing 5.3: Injektion an einem Argument vom Typ `int`

In Experiment 22 (Listing 5.5) wird ein Objekt der Klasse `jpge::image` mit dem Standard-Konstruktor angelegt und injiziert.

5.2.2.2. Injektionsmodi *offset+*, *offset-* und *invert*

Mit *offset+* und *offset-* können Werte eines Zahlentyps um jeweils 1 erhöht oder verringert werden. Das ermöglicht die Injektion eines *OffByOne*-Fehlzustands (vgl. [5, 7])). Im Modus *invert* wird bei Zahlenwerten das Vorzeichen getauscht und bei booleschen Werten die logische Negation gebildet, was unter anderem die Auswirkung eines *bit-flips* (vgl. [10, 1]) simulieren kann.

Den Injektionsmodus *invert* wird beispielsweise in Experiment 14 verwendet. Darin wird mit einer Wahrscheinlichkeit von 1% das Vorzeichen des der Funktion `image::set_px` übergebenen Arguments vom Typ `float` invertiert. Dieses Argument beinhaltet einen Farbwert, welcher an der ebenfalls übergebenen Position in die Instanz der Klasse *image*, deren Funktion aufgerufen wurde, eingefügt wird. Abbildung 5.1a zeigt das Ergebnis des *golden runs*, bei dem die Eingabedatei mit der Qualitätsstufe 50 komprimiert wurde. In Abbildung 5.1b sind die Auswirkungen des Experiments zu erkennen, die durch viele kleine Bildfehler zum Ausdruck kommen. Im Ergebnis des durch das Vergleichsskript ausgeführten Bildvergleichs, zu sehen in Abbildung 5.1c, ist die Beeinflussung des Kompressionsprozesses durch

```
[24.05.2017 16:53:53.059690] [Experiment 16] [pid 17538, thread 17538] [setup] LogLevel: 3.
 ↪  Experiment begins.
[24.05.2017 16:53:53.059721] [Experiment 16] [pid 17538, thread 17538] [setup] target
 ↪  configuration:
target                        probability   callCount     occurrence    injection mode
-------------------------------------------------------------------------------------
primitive_charArray_argument  0             1             0             replace

[24.05.2017 16:53:53.060584] [Experiment 16] [pid 17538, thread 17538] [injection] function:
 ↪  'bool jpge::compress_image_to_jpeg_file(const char *,int,int,int,const unsigned char
 ↪  *,const jpge::params &)' in file 'encoder.cpp' in line 388:
       images/ship_experiment_50.jpg ---> nil.1
```

Listing 5.4: Injektion eines Feldes

```
[24.05.2017 16:53:59.045290] [Experiment 22] [pid 17583, thread 17583] [setup] LogLevel: 3.
 ↪  Experiment begins.
[24.05.2017 16:53:59.045331] [Experiment 22] [pid 17583, thread 17583] [setup] target
 ↪  configuration:
target           probability   callCount     occurrence    injection mode
-----------------------------------------------------------------------
object_argument  0             1             0             replace

[24.05.2017 16:53:59.046941] [Experiment 22] [pid 17583, thread 17583] [injection] function:
 ↪  'void jpge::image::subsample(jpge::image &,int)' in file 'jpge.cpp' in line 948:
       0x130000000e0 ---> 0xba5ce0
```

Listing 5.5: Injektion eines Objekts

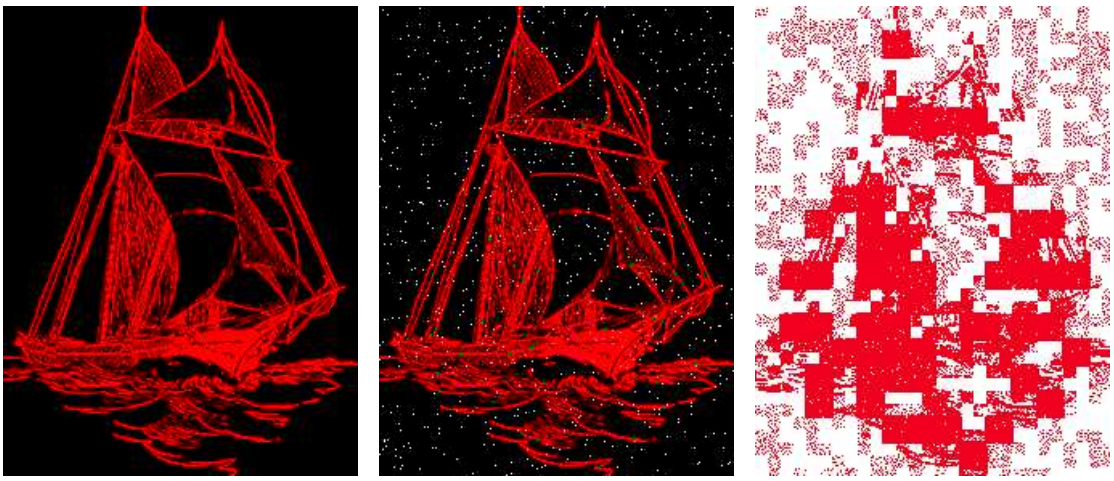
das Injektionsexperiment deutlich zu erkennen, da das Vergleichsskript bei der Eingabe zweier gleicher Bilder ein komplett weißes Bild ausgibt.

5.2.3. Trigger

Die implementierte Software stellt drei Trigger zur Verfügung. Fehlzustände können an Software-Schnittstellen damit nach Wahrscheinlichkeit, nach Anzahl der Schnittstellennutzung sowie bei jeder unter der gegebenen Arbeitslast vorkommenden Nutzung einzeln injiziert werden. Das Verhalten der ersten beiden Trigger kann für jedes Experiment durch die Angabe von Aufrufzahlen beziehungsweise Wahrscheinlichkeitswerten konfiguriert werden.

Die Injektion anhand von Wahrscheinlichkeitswerten basiert auf pseudozufälligen Zahlenfolgen. Diese werden mithilfe eines sogenannten *seeds* generiert, der bei der Nutzung des Wahrscheinlichkeitstriggers in der Konfiguration des Experiments angegeben werden kann. Da der gleiche *seed* immer für die gleiche Zahlenfolge sorgt, können auf Wahrscheinlichkeitswerten basierende Experimente durch erneute Ausführung mit dem selben *seed* reproduziert werden.

Durch die Variation der Wahrscheinlichkeitswerte kann bei der Injektion an Schnittstellen zu Funktionen des Betriebssystems die unterschiedliche Auslastung benötigter Ressourcen simuliert werden. Je höher diese Auslastung ist, desto wahrscheinlicher schlagen zum Beispiel von Software angefragte Speicherreservierungen fehl. Ein Beispiel bietet Experiment 23, welches Aufrufe von `malloc` in 75% der



(a) Ergebnis des *golden runs* (b) Ergebnis des Experiments (c) Ergebnis des Bildvergleichs

Abbildung 5.1.: Ergebnisse aus Experiment 14

Fälle fehlschlagen lässt.

Die Wahrscheinlichkeitswerte 0 und 100 sorgen dafür, dass in einem Experiment eine Injektion entweder nie oder aber bei jeder Nutzung der zugehörigen Schnittstelle stattfindet.

Der Trigger, welcher Injektionen periodisch nach einer angegebenen Anzahl von Nutzungen der Schnittstellen auslöst, verhält sich besser vorhersagbar. Hier wird im Rahmen eines Experiments ebenfalls nur ein Teil der Schnittstellennutzungen mit Fehlzuständen behaftet, jedoch immer in einem festen Intervall. Damit lassen sich beispielsweise Konflikte bei der Ressourcennutzung konkurrierender Prozesse simulieren. Das Fehlschlagen einer Ressourcenanforderung eines Prozesses A könnte zur Ursache haben, dass die angeforderte Ressource alle n Anforderungen nicht verfügbar ist, da sie ebenfalls von einem Prozess B genutzt wird. Prozess B gibt sie bei voller Auslastung wieder frei, sodass die nächsten $n-1$ Anforderungen von Prozess A erfolgreich sind.

Der dritte Trigger bietet im Vergleich zu den anderen die höchste Abdeckung. Hiermit kann das Verhalten der getesteten Software bei der Injektion eines Fehlzustandes an jeder unter einer gegebenen Arbeitslast vorkommenden Nutzung der festgelegten Schnittstelle untersucht werden. Das Beispiel in Abschnitt 5.7.1 verdeutlicht dies, indem für jeden vorkommenden Aufruf von `malloc` dessen Fehlschlagen simuliert wird. Anders als bei der Verwendung des Wahrscheinlichkeitstriggers mit dem Wert 100 oder des Aufrufzahltriggers mit dem Wert 1, wird hier pro Unterexperiment genau ein Fehlzustand injiziert. Experimente, die erstere Konfigurationen benutzen, können bei denjenigen Schnittstellennutzungen keine Fehlzustände injizieren, die auf eine Injektion folgen, welche die getestete Software

zum Abstürzen bringt. Hinzu kommt, dass bei der Injektion mehrerer Fehlzustände pro Experiment Wechselwirkungen möglich sind, deren nachträgliche Analyse möglicherweise sehr komplex ist. Diese bleiben bei der Verwendung des dritten Triggers aus.

Die Auswertung der implementierten Injektionsmöglichkeiten zeigt, dass die Anforderungen an die Konfigurierbarkeit im Bezug auf das *Wo*, *Was* und *Wann* erfüllt wurden. Das damit bereitgestellte Fehlermodell ermöglicht den Einsatz von Fehlerinjektion für vielfältige Zwecke.

5.3. Automatisierung

Die Automatisierung umfasst die Umsetzung dessen, was in der Konfiguration festgelegt wird. Das beinhaltet die Generierung eines spezifischen Aspect-Headers, das Kompilieren der zu testenden Software unter Einbeziehung dieses Headers, die Durchführung und Protokollierung der Testläufe sowie das Vergleichen jedes Ergebnisses mithilfe des hinterlegten Befehls. Unterstützung beim Anlegen der Konfigurationsdateien oder der Interpretation der Ergebnisse ist bisher nicht vorhanden. Mögliche Ansätze dazu sind in Abschnitt 6.3.5 aufgeführt.

5.4. Performance

Im Folgenden werden die nicht-funktionalen Eigenschaften der Software untersucht. Anhand von Testwerten werden die Auswirkungen von *AOFIT* auf die damit getestete Software sowie die Kompilier- und Laufzeit dargestellt.

Tabelle 5.1 zeigt eine Gegenüberstellung von Kompilierzeiten für die beiden getesteten Projekte. Die Zeiten der Spalte `g++` wurden dabei beim Kompilieren der unmodifizierten Projekte gemessen. Die Messwerte aus der Spalte `ag++` resultieren aus Kompiliervorgängen mit AspectC++ unter Einbeziehung der angehängten Konfigurationen. Das Einweben des daraus generierten Aspect-Headers führt zu deutlich erhöhten Kompilierzeiten. Diese lassen sich zwar durch Parallelisierung, im Beispiel mit vier parallelen Jobs, sowohl absolut, als auch prozentual senken. Der dennoch erhöhte Zeitbedarf macht allerdings deutlich, wie wichtig die hohe Konfigurierbarkeit der kompilierten Software zur Laufzeit ist. Durch diese ist ein erneutes Kompilieren nur bei Änderungen an den Schnittstellendefinitionen oder den Fehlzustandswerten notwendig.

Tabelle 5.2 vergleicht die Dateigrößen der unter verschiedenen Bedingungen kompilierten Projekte. Für `lzham_codec` ist die Betrachtung in das ausführbare Hilfsprogramm `lzhamtest` und die dynamische Bibliothek `liblzhamcomp.so` aufgeteilt. Die Spalte „header (base)“ beinhaltet die Größen der Dateien, die beim

	g++		ag++	
	/	-j4	/	-j4
lzham_codec	7.5s	5.2s	69.6s	40s
jpeg-compressor	6.6s	4.3s	30.6s	17s

Tabelle 5.1.: Vergleich der Kompilierzeiten

Kompilieren mit einem von *AOFIT* ohne Advice-Code für Fehlerinjektion generierten Aspect-Header entstanden sind. „Header (full)“ bedeutet, dass die Projekte mit dem Aspect-Header kompiliert wurden, welcher aus der angehängten Konfiguration entstanden ist.

Die Differenz zwischen den Werten aus „original“ und „header (base)“ entsteht durch den Code des Aspect-Headers, der grundlegende Funktionalitäten wie das Logging und die Konfiguration zur Laufzeit bereitstellt. Der Unterschied zwischen „header (base)“ und „header (full)“ ist durch den Code bedingt, der durch die jeweiligen Advices zur Fehlerinjektion an den in der Konfiguration festgelegten Schnittstellen hinzugefügt wird.

		original	**header (base)**	**header (full)**
lzham_codec	lzhamtest	52.6kb	201kb	233.9kb
	liblzhamcomp.so	121.4kb	113kb	324.3kb
jpeg-compressor	encoder	292.7kb	435kb	723.6kb

Tabelle 5.2.: Vergleich der Programmgrößen

Tabelle 5.3 stellt die Anzahl der Log-Meldungen und die Laufzeiten unter drei verschiedenen Log-Stufen von vier Experimente aus der Kampagne zu *jpeg-compressor* gegenüber. Die Experimente wurden gewählt, da sie die meisten Injektionen und dadurch auch die meisten Log-Meldungen zur Folge hatten. Als Log-Meldung wurde die Meldung zu einer Injektion gezählt, welche immer aus drei Zeilen besteht, weshalb unter der Log-Stufe 1 keine Meldungen angegeben sind. Um mögliche Ausreißer bei der Laufzeitmessung auszugleichen, enthält die entsprechende Spalte gerundete Mittelwerte über mehrere Testläufe.

Zunächst ist im Vergleich zwischen dem *golden run* und den Experimenten erkennbar, dass die Injektion von Fehlzuständen während eines Experiments eine deutlich negative Auswirkung auf die Laufzeit hat. Die verwendete Logging-Bibliothek *spdlog* bietet zwar eine hohe Performance. Diese kann in der Implementierung allerdings nicht abgerufen werden, da die gepufferten Logmeldungen vor jeder Injektion in die Logdatei geschrieben werden. Das hat den Grund, dass ansonsten bei Injektionen, welche die untersuchte Software zum Abstürzen bringen, keinerlei Informationen zu dem dafür verantwortlichen Fehlzustand protokolliert und dadurch keine Diagnosemöglichkeiten geliefert werden. Ein Ansatz für besse-

re Performance könnte die Möglichkeit sein, das explizite Schreiben des Logs vor Injektionen in der Konfiguration zu deaktivieren. Entstehen durch Experimente unprotokollierte Abstürze, so können diese Experimente mit der langsameren, aber sicheren Logging-Variante erneut durchgeführt werden, um Informationen über die Ursache zu erhalten.

Bei den Experimenten 6 und 17 sowie 13 und 14 zeigt sich bei einer ähnlichen beziehugsweise gleichen Anzahl von Meldungen bei gleicher Log-Stufe auch ein ähnliches Verhältnis von Meldungen und Laufzeit. Bei den Experimenten 13 und 14 zeigen sich beim Wechsel der Log-Stufe von 2 auf 3, wodurch pro Experiment zusätzlich ein genauer Zeitstempel protokolliert wird, Performanceverluste von ca. 15-25%. Auffällig ist jedoch, dass diese Experimente unter der niedrigsten Log-Stufe, bei welcher keinerlei Informationen zu Injektionen protokolliert werden, die meiste Zeit benötigen.

Experiment	**Modus**	**Log-Stufe**	**Laufzeit**	**Meldungen**	$\frac{\textbf{Meldungen}}{\textbf{Sekunde}}$
golden run	-	-	0.009s	-	-
6	*invert*	1	6s	-	-
		2		293	48.8
		3			
13	*offset-*	1	18.7s	-	-
		2	10s	1208	120.8
		3	13.3s		90.8
14	*invert*	1	16.6s	-	-
		2	9.1s	1208	132.7
		3	10.6s		114
17	*replace*	1	5.4s	-	-
		2		276	51.1
		3			

Tabelle 5.3.: Stichproben zu Laufzeit und Logging aus den Experimenten mit *jpeg-compressor*

5.5. Plattformunabhängigkeit

Der implementierte Ansatz ist in der Theorie weitestgehend plattformunabhängig. Die Schnittmenge der als unterstützt angegebenen Plattformen der verwendeten Logging-Bibliothek *spdlog*, des AspectC++-Compilers und Python beläuft sich auf die Plattformen Linux, MacOS X und Windows.

Die Testläufe für dieses Kapitel wurden unter einem 64-Bit Linux mit dem AspectC++-Compiler in der Version 2.2 und `g++` in der Version 5.4 durchgeführt.

AOFIT wurde mit der Kampagne für *jpeg-compressor* zudem erfolgreich unter einem MacOS Sierra mit der gleichen AspectC++-Version und `clang` 802.0.42 getestet. Auffällig war dabei jedoch, dass als Threadnummer für jedes Experiment die gleiche 18-stellige Zahl protokolliert wurde, was vermutlich auf ein Fehlverhalten von *spdlog* zurückzuführen ist.

Bei der Verwendung verschiedener Systeme muss sowohl bei *AOFIT* als auch bei in Python geschriebenen Vergleichsskripten darauf geachtet werden, das sogenannte *Shebang*, also die mit `#!` beginnende erste Zeile, auf den systemspezifischen Pfad des Python 3.5 Interpreters anzupassen. Alternativ können die Skripte in der Konsole beziehungsweise über das Attribut *comparisonCommand* der Kampagnenkonfiguration explizit mit dem Python-Interpreter aufgerufen werden.

5.6. Flexibilität

Damit eine Software mit *AOFIT* untersucht werden kann, müssen Voraussetzungen erfüllt sein, die in Abschnitt 4.8 erläutert sind. Diese werden im Folgenden bei der Beantwortung der Frage nach dem Grad der Flexibilität berücksichtigt.

C++ ist gerade im Bereich eingebetteter Systeme weit verbreitet. Durch die häufige Verwendung stellt die Anforderung, dass die zu testende Software in C++ geschrieben sein muss, keine große Einschränkung des Einsatzbereiches dar.

Da AspectC++ mit dem Einweben der Aspekte in den Ziel-Quellcode gültigen C++-Code generiert und diesen an einen gewöhnlichen C++-Compiler übergibt, ist die Anforderung der Kompilierbarkeit mit AspectC++ in der Theorie ebenfalls keine Einschränkung. In der Praxis ist es jedoch häufig kompliziert, den AspectC++-Compiler in das Build-System einer vorhandenen Software zu integrieren. Durch die äußerst hohe Komplexität von C++, die Verwendung verschiedener Sprachstandards oder die Nutzung implementierungs- und compilerspezifischer Elemente kann es vorkommen, dass Quellcode nicht ohne Änderungen mit AspectC++ kompilierbar ist. Viele Schwierigkeiten lassen sich jedoch mit den folgenden Optionen für `ag++`, den Wrapper für den AspectC++-Compiler `ac++`, beheben [3]:

`--Xcompiler:`
: auf diesen Schalter folgende Optionen werden vom AspectC++-Compiler nicht ausgewertet. Dies kann helfen, wenn für den Backend-Compiler bestimmte Optionen vom AspectC++-Compiler missinterpretiert werden.

`-p | --path:`
: `ag++` nimmt als Projektverzeichnis standardmäßig das aktuelle Arbeitsverzeichnis an. Falls das Projektverzeichnis abweicht, kann es mit dieser Option explizit angegeben werden.

5.7. Nützlichkeit

Eine Logdatei, die mit der maximalen Logging-Stufe (vgl. Tabelle 3.1) erstellt wurde, enthält umfangreiche Informationen über die Durchführung einer Fehlerinjektionskampagne. Durch den Vergleich des Ergebnisses mit dem Referenzergebnis aus dem *golden run* und durch den Statuscode, mit dem die getestete Software terminiert, kann man zunächst Experimente finden, die einen besonderen Einfluss hatten, etwa indem sie zu Abstürzen oder zu stark abweichenden Ergebnissen führten. In der Dokumentation des Injektionsvorgangs sind die Signatur der Schnittstelle, an welcher ein Fehlzustand injiziert wurde sowie der Dateiname des aufrufenden Quellcode-Abschnitts und die Zeilennummer des Aufrufs enthalten. Mit diesen Informationen lassen sich nun die möglicherweise fehlerbehafteten *FTAMs* der getesteten Software untersuchen und verbessern.

5.7.1. Beispiel: Speicherreservierung in *jpeg-compressor*

Als Beispiel dient hier Experiment 24 aus der Kampagne zu *jpeg-compressor*. Darin wurde das Fehlschlagen jedes einzelnen unter der in der Kampagne definierten Arbeitslast vorkommenden Aufrufs der Speicherallokierungsfunktion `malloc` simuliert. Listing 5.6 zeigt das Protokoll des vierten Unterexperiments.

```
1  [24.05.2017 16:53:59.430228] [Experiment 24.4] [pid 17599, thread 17599] [setup] LogLevel: 3.
    ↪  Experiment begins.
2  [24.05.2017 16:53:59.430258] [Experiment 24.4] [pid 17599, thread 17599] [setup] target
    ↪  configuration:
3  target  probability   callCount     occurrence    injection mode
4  ----------------------------------------------------------------
5  malloc  0             0             4             replace
6
7  [24.05.2017 16:53:59.431155] [Experiment 24.4] [pid 17599, thread 17599] [injection] function:
    ↪  'void *malloc(unsigned long int)' in file 'jpge.cpp' in line 34:
8  0x7f26bbba4010 ---> 0
9
10 Experiment 24.4: program returned -11
11 comparison command returned 2 with output:
12 ======
13 compare: Empty input file ‘images/ship_experiment_50.jpg’ @ error/jpeg.c/JPEGErrorHandler/322.
14
15 ======
```

Listing 5.6: Experiment 24.4: Simulation fehlschlagender Speicherallokation

Hier terminiert *jpeg-compressor* mit dem Rückgabecode `-11`, der eine Speicherschutzverletzung (engl.: *segmentation fault*) signalisiert. Der Rückgabecode 2 des im Vergleichsskript verwendeten Programms `compare` signalisiert einen Fehler während des Vergleichs. Die protokollierte Fehlermeldung des Tools erläutert, dass die Eingabedatei leer ist. Bei der Ausgabedatei, die vom Vergleichs-Skript gesichert wird, handelt es sich bei Experiment 24.4 tatsächlich um eine leere Datei. Die Simulation einer fehlschlagenden Speicheranforderung hat an dieser Stelle

also zu einem unkontrollierten Abbruch der getesteten Software geführt und ein leeres Ergebnis produziert.

Listing 5.7 zeigt die Stelle des Quellcodes, auf welche der Log des Experiments verweist.

```
32 static inline void *jpge_malloc(size_t nSize)
33 {
34     return malloc(nSize);
35 }
```

Listing 5.7: Ausschnitt aus 'jpge.cpp': Ein kritischer Aufruf von `malloc`

Für den Aufruf der Wrapper-Methode `jpge_malloc` des Kodierungs-Moduls `jpge` kommen nach kurzer Suche wiederum nur zwei Stellen in Frage, die in Listing 5.8 aufgeführt sind.

```
625 void image::init()
626 {
627     m_pixels = static_cast<float *>(jpge_malloc(m_x * sizeof(float) * m_y));
628     m_dctqs = static_cast<dctq_t *>(jpge_malloc(m_x * sizeof(dctq_t) * m_y));
629 }
```

Listing 5.8: Ausschnitt aus 'jpge.cpp': Nutzung von `jpge_malloc`

Die Speicherreservierung für `m_pixels` und `m_dctqs` wird also nicht gegen ein mögliches Fehlschlagen abgesichert. Die Funktion `jpge::init` verlässt sich auf den Erfolg der Wrapper-Funktion `jpge_malloc`, welche allerdings keinerlei Fehlertoleranz implementiert. Auch die hier initialisierten Attribute werden an keiner Stelle vor ihrer Verwendung geprüft.

Die Verwendung der funktionsäquivalenten Methode `jpgd_malloc` des Dekodierungs-Moduls `jpgd` ist hingegen besser implementiert. Ein Beispiel zeigt Listing 5.9, in welchem die Dekodierung bei Misserfolg mit einem passenden Fehlercode abgebrochen wird.

```
847 mem_block *b = (mem_block *)jpgd_malloc(sizeof(mem_block) + capacity);
848 if (!b) {
849     stop_decoding(JPGD_NOTENOUGHMEM);
850 }
```

Listing 5.9: Ausschnitt aus 'jpgd.cpp': Nutzung von `jpgd_malloc` mit implementierter Fehlerbehandlung

6. Zusammenfassung und Ausblick

Zum Abschluss der Arbeit werden die Ergebnisse zusammengefasst. Nach einem Überblick über mögliche Verwendungszwecke der Software wird ein Ausblick darauf gegeben, welche Erweiterungen möglich und sinnvoll wären. Ebenfalls wird erklärt, wie sie in *AOFIT* integriert werden könnten.

6.1. Zusammenfassung

Die mit *AOFIT* implementierte Lösung ermöglicht die Injektion von Fehlzuständen an Schnittstellen zwischen verschiedenen Softwarekomponenten sowie zwischen Software und Programmbibliotheken. Über die Konfigurationsdateien sind Software-Schnittstellen sowie Fehlerinjektionskampagnen und deren Durchführung detailliert konfigurierbar. Die Lösung führt jene für die Umsetzung der Konfiguration notwendigen Schritte automatisiert durch und liefert umfangreiche Ergebnisse in Form eines Logs. Sind die Voraussetzungen für die Nutzung von *AOFIT* gegeben, so ist es flexibel für vielerlei Software einsetzbar.

Die Implementierung zeigt die Eignung aspektorientierter Programmierung für den Einsatz zur Fehlerinjektion an Software-Schnittstellen. Die Vorteile liegen in der Möglichkeit, auch an Schnittstellen innerhalb einer Software ansetzen zu können sowie in der freien Konfigurierbarkeit der Schnittstellen, die den Einsatz nicht auf bestimmte Programme oder Programmbibliotheken limitiert. Nachteile ergeben sich aus dem erhöhten Zeitbedarf beim Kompilieren und der teilweise aufwendigen Integration von AspectC++ in den Build-Prozess.

6.2. Mögliche Anwendungsfelder

Anders als bei FIG [4] oder Hovac [5] wird die Implementierung der Fehlerinjektion nicht in einer Programmbibliothek bereitgestellt, die zur Laufzeit eingeschleust wird, sondern mit Hilfe von AspectC++ direkt in die zu testende Software integriert. Dafür muss ein Zugriff auf deren Quellcode möglich sein, was etwa bei der Entwicklung eigener Software sowie bei quelloffener Software gegeben ist.

Der primäre Zweck von *AOFIT* ist der Einsatz zur Verbesserung oder Nachrüstung von *FTAMs*. In einem gewissen Maße ist jedoch auch eine Zuverlässigkeitsbewertung möglich. So könnte man vorgefertigte Konfigurationen mit passenden Experimenten für häufig verwendete Schnittstellen an vielerlei Software testen und die Ergebnisse miteinander vergleichen. Beispiele für solche Schnittstellen sind etwa Funktionen zum Reservieren von Speicher oder zur Abfrage der Zeit.

6.3. Mögliche Erweiterungen

Das entwickelte Werkzeug ist im Bezug auf die Performance, den Komfort und den Funktionsumfang noch optimier- beziehungsweise erweiterbar. Im Folgenden werden einige Ideen erläutert, die das vorgestellte Konzept erweitern und aufgrund des zeitlichen Rahmens dieser Arbeit nicht umgesetzt wurden.

6.3.1. Parallelisierung der Experimente

Da die einzelnen Experimente nicht voneinander abhängig sind, wäre es möglich, ihre Ausführung zu parallelisieren, um die Durchführung von Kampagnen zu beschleunigen. Dazu könnte der Kampagnenkonfiguration ein Attribut hinzugefügt werden, in welchem ein Wert für die maximale Anzahl paralleler Experimente hinterlegt werden kann. Das Ausführen paralleler Aufgaben kann in *AOFIT* mithilfe der Standardbibliothek von Python auf verschiedene Arten realisiert werden[1]. Es sollte bei der Umsetzung darauf geachtet werden, die Ergebnisse der einzelnen Experimente zum Schluss in einer gemeinsamen Logdatei zusammenzuführen.

6.3.2. Weitere Injektionsmodi

Statt der Manipulation eines Rückgabewertes selbst, könnte der Zeitpunkt der Bereitstellung verändert werden, wie etwa von Herscheid, Richter und Polze beschrieben [5]. Mit einem zusätzlichen Injektionsmodus, etwa *delay* genannt, könnte das Zurückkehren einer Funktion um einen konfigurierbaren Zeitwert künstlich verzögert werden. Dieser Modus würde sich gerade für die Untersuchung von Echtzeitsystemen, bei denen es auf die Einhaltung von Deadlines ankommt, als nützlich erweisen [11].

Ein weiterer Injektionsmodus könnte das sogenannte *fuzzing* [10] ermöglichen. Anders als im Modus *replace* würde dabei der zu injizierende Fehlzustand nicht in der Konfiguration des Experiments festgelegt, sondern zufällig generiert. Dies würde für jeden Datentypen, der unterstützt werden soll, eine spezielle Funktion erfordern, die zufällige Daten dieses Typs generiert. Das Attribut *seed* könnte

[1] `https://docs.python.org/3.5/library/concurrency.html`

hier wiederverwendet werden, indem es für die Initialisierung des den *fuzzing*-Funktionen zugrunde liegenden Pseudo-Zufallszahlen-Generators eingesetzt wird.

6.3.3. Erweiterte Injektion an Funktionsparametern

Die Festlegung der Parameter über ihren Typ ist bisher so implementiert, dass der erste passende Parameter als Injektionsziel gewählt wird. Dadurch ist es nicht möglich bei einer Funktion, die mehrere Parameter des gleichen Typs besitzt, Fehlzustände an einem anderen Parameter als dem Ersten zu injizieren. Als Erweiterung wäre es denkbar, die Konfiguration um eine Möglichkeit zu erweitern, die Zielparameter zum Beispiel durch Angabe ihres Indexes statisch festzulegen. Damit die Definition von Schnittstellen allein anhand von Parametertypen und damit unabhängig von spezifischen Funktionen weiterhin möglich bleibt, könnten im Fall von Parametern gleichen Typs für jeden davon Unterexperimente generiert werden.

6.3.4. Mehrere aktive Schnittstellen

Aktuell können Fehlzustände mit dem Wahrscheinlichkeits- und dem Aufrufzahltrigger zwar mehrfach pro Experiment injiziert werden, allerdings immer nur an der selben Schnittstelle. Die Parameter 2-4 der Konfigurationsschnittstelle (siehe Abschnitt 4.4) unterstützen zwar bereits die Aktivierung mehrerer Advices und die entsprechende Konfiguration ihrer Trigger. Da die umfassende Unterstützung der Injektion einzelner Fehlzustände pro Experiment eine höhere Priorität hatte, wurde diese Möglichkeit im weiteren Verlauf der Implementierung jedoch nicht weiter verfolgt, weshalb sie aktuell nicht konfigurierbar ist.

Für eine Vervollständigung dieser Implementierung müssten auch die Parameter 5 sowie 8-11 und die von ihnen konfigurierten Funktionen an die neuen Bedingungen angepasst werden.

6.3.5. Unterstützung bei Konfiguration und Auswertung

Das Erstellen der Konfiguration sowie die Analyse des Logs nach Abschluss des automatischen Prozesses geschieht aktuell vollständig manuell. An beiden Stellen würde eine gewisse Unterstützung für deutlich mehr Komfort sorgen.

Für die Definition der Schnittstellen wäre zum Beispiel eine Auflistung der Signaturen aller in der zu testenden Software implementierten sowie von externen Bibliotheken genutzen Funktionen hilfreich. Daraus könnten automatisch alle möglichen Schnittstellendefinitionen generiert werden, aus denen der Anwender dann etwa in einer grafischen Oberfläche die gewünschten auswählen könnte.

Auch das Erstellen von Kampagnen könnte mit einer grafischen Hilfssoftware deutlich vereinfacht werden. Diese könnte etwa mit Vorlagen und Auswahlmenüs

das Erstellen von Experimenten und Experimentgruppen unterstützen und daraus die entsprechende Konfigurationsdatei generieren.

Die Auswertung der Ergebnisse aus den Experimenten könnte durch eine automatische Interpretation unterstützt werden. So könnten beispielsweise Experimente, die Abstürze oder vom Referenzergebnis abweichende Ergebnisse zur Folge hatten, hervorgehoben oder zusammengefasst werden. Hilfreich wäre auch das Anzeigen des jeweiligen Quellcode-Abschnitts, in welchem die Schnittstelle benutzt wurde, in die im entsprechenden Experiment injiziert wurde.

Zusammengefasst bietet diese Arbeit mit *AOFIT* eine beispielhafte Implementierung eines Konzepts zur Injektion von Fehlzuständen an verschiedenen Arten von Software-Schnittstellen und demonstriert daran die Stärken und Schwächen eines aspektorientierten Ansatzes zur Fehlerinjektion. Wie gezeigt, ist *AOFIT* ein flexibel einsetzbares Werkzeug, auf dessen Basis noch weitere Funktionalität implementierbar ist.

Literatur

[1] J. Arlat u. a. “Dependability of COTS Microkernel-Based Systems”. In: *IEEE Transactions on Computers* 51.2 (2002), S. 138–163. ISSN: 0018-9340. DOI: 10.1109/12.980005.

[2] A. Avižienis u. a. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), S. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[3] G. Blaschke. *Ag++ manual*. Reference. Version 0.9. URL: http://aspectc.org/doc/ag-man.pdf.

[4] P. Broadwell, N. Sastry und J. Traupman. “FIG: A Prototype Tool for Online Verification of Recovery Mechanisms”. In: *Workshop on Self-Healing, Adaptive and Self-Managed Systems*. Citeseer. 2002.

[5] L. Herscheid, D. Richter und A. Polze. “Hovac: A Configurable Fault Injection Framework for Benchmarking the Dependability of C/C++ Applications”. In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, S. 1–10. DOI: 10.1109/QRS.2015.12.

[6] M. Hoffmann. “Konstruktive Zuverlässigkeit: Eine Methodik für zuverlässige Systemsoftware auf unzuverlässiger Hardware”. Diss. 2016.

[7] A. Johansson und N. Suri. “Error Propagation Profiling of Operating Systems”. In: *2005 International Conference on Dependable Systems and Networks (DSN’05)*. 2005, S. 86–95. DOI: 10.1109/DSN.2005.45.

[8] P. D. Marinescu und G. Candea. “Efficient Testing of Recovery Code Using Fault Injection”. In: *ACM Trans. Comput. Syst.* 29.4 (Dez. 2011), 11:1–11:38. ISSN: 0734-2071. DOI: 10.1145/2063509.2063511.

[9] B. P. Miller, L. Fredriksen und B. So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (Dez. 1990), S. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[10] R. Natella, D. Cotroneo und H. S. Madeira. “Assessing Dependability with Software Fault Injection: A Survey”. In: *ACM Comput. Surv.* 48.3 (Feb. 2016), 44:1–44:55. ISSN: 0360-0300. DOI: 10.1145/2841425.

[11] M. Rodríguez, A. Albinet und J. Arlat. “MAFALDA-RT: A Tool for Dependability Assessment of Real-Time Systems”. In: *Proceedings International Conference on Dependable Systems and Networks*. 2002, S. 267–272. DOI: 10.1109/DSN.2002.1028909.

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

[13] O. Spinczyk und D. Lohmann. “The design and implementation of AspectC++”. In: *Knowledge-Based Systems* 20.7 (2007). Special Issue on Techniques to Produce Intelligent Secure Software, S. 636 –651. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2007.05.004.

[14] H. Ziade, R. Ayoubi, R. Velazco u. a. “A Survey on Fault Injection Techniques”. In: *Int. Arab J. Inf. Technol.* 1.2 (2004), S. 171–186.

Abbildungsverzeichnis

2.1. Die Fehlerkette: vom Defekt zum Fehlverhalten 4

3.1. Struktur des Entwurfs 10

4.1. Struktur der Implementierung und der automatisierten Umsetzung der Konfiguration 16

5.1. Ergebnisse aus Experiment 14 33

Tabellenverzeichnis

3.1. Entwurf zur Konfiguration der Protokollierung anhand festgelegter Stufen 13

4.1. Struktur der Konfigurationsdatei einer Kampagne 18
4.2. Struktur eines Fehlerinjektionsexperiments 19

5.1. Vergleich der Kompilierzeiten 35
5.2. Vergleich der Programmgrößen 35
5.3. Stichproben zu Laufzeit und Logging aus den Experimenten mit *jpeg-compressor* 36

Listingverzeichnis

2.1. Aspekt mit Advice-Code. 7

4.1. Definition der Schnittstelle `primitive_bool_argument` 17
4.2. Definition der Schnittstelle `enum_result` 17
4.3. Beispiel für eine Fehlerinjektionskampagne (gekürzt) 20
4.4. Templates für die Trigger. 20
4.5. Beispiel eines Vektors, der zwei zu injizierende Werte speichert . . 21
4.6. Generierter Advice-Code für die Injektion an Rückgabewerten mit dem Aufrufzahltrigger 22
4.7. Aufruf des modifizierten Programms '`encoder`' für den *golden run* 23
4.8. Aufruf des modifizierten Programms '`encoder`' für Experiment 13 23
4.9. Die Template-Funktion `log` und die notwendige Spezialisierung für Referenzen vom Typ `std::vector<bool>::reference` 23
4.10. Nutzung von `enable_if` zur Maßschneiderung der Spezialisierungen der `inject`-Funktion 25

5.1. Manipulation des Parameters `buf_len` 30
5.2. Injektion an booleschem Rückgabewert 31
5.3. Injektion an einem Argument vom Typ `int` 31
5.4. Injektion eines Feldes 32
5.5. Injektion eines Objekts 32
5.6. Experiment 24.4: Simulation fehlschlagender Speicherallokation . 38
5.7. Ausschnitt aus 'jpge.cpp': Ein kritischer Aufruf von `malloc` 39
5.8. Ausschnitt aus 'jpge.cpp': Nutzung von `jpge_malloc` 39
5.9. Ausschnitt aus 'jpgd.cpp': Nutzung von `jpgd_malloc` mit implementierter Fehlerbehandlung 39

A.1. `struct`-Templates zur Ermittlung eines Argumentindexes 53
A.2. Die Template-Funktion `checkArgs` 53
A.3. Advice-Template zur Injektion an Argumenten 54

B.1. Konfiguration der Interfaces für *jpeg-compressor* 56
B.2. Konfiguration der Kampagne für *jpeg-compressor* 57
B.3. Konfiguration der Interfaces für *lzham_codec* 58
B.4. Konfiguration der Kampagne für *lzham_codec* 58

A. *AOFIT*

```
template <int I> struct ArgChecker{
    template <typename argType, class JP> static inline int work (JP &tjp) {
      int i = ArgChecker<I - 1>::template work<argType> (tjp);
    if(i == -1) {
      if(typeid(argType) == typeid(*tjp.template arg<I-1>()) || typeid(argType)
 ↪   == typeid(&(*tjp.template arg<I-1>()))) {
        return I-1;
      } else {
        return -1;
      }
    } else {
      return i; //first argindex of matching type
    }
  }
};
template <> struct ArgChecker<0> { //anchor
  template <typename argType, class JP> static inline int work (JP &tjp) {
    return -1;
  }
};
```

Listing A.1: `struct`-Templates zur Ermittlung eines Argumentindexes

```
template <typename argType, class JP> int checkArgs(JP &tjp) {
  return ArgChecker<JP::ARGS>::template work<argType>(tjp);
}
```

Listing A.2: Die Template-Funktion `checkArgs`

```
1  advice call("$signature") : before() {
2    occurrences[$id]++; //used for 'each_occurrence_once' experiment mode
3    if(activeTargets[$id]) {
4      $trigger //replaced by trigger_probabilities or trigger_callCounts
5        tjp->arg(0); //workaround taken from
     ↪  'src/AspectC++-Project/AspectC++/tests/StaticTrace/Trace.ah'
6        int index = checkArgs<$argType>(*tjp); //get index of (first) argument
     ↪  of requested type
7        if(index != -1) {
8          $argType* arg = ($argType*)tjp->arg(index);
9          if(injection_mode != "replace") {
10           inject(tjp, arg);
11         } else {
12           if (*arg != $valueVector[valueID[$id]]) {
13             log(tjp->filename(), tjp->signature(), tjp->line(), *arg,
     ↪  $valueVector[valueID[$id]]);
14             fileLogger->flush();
15             *arg = $valueVector[valueID[$id]];
16           } else {
17             log(tjp->filename(), tjp->signature(), tjp->line(), *arg,
     ↪  $valueVector[valueID[$id]], "value planned to inject equals argument");
18           }
19         }
20         $triggerReset //for callCount trigger
21       }
22     }
23   }
24 }
```

Listing A.3: Advice-Template zur Injektion an Argumenten

B. Konfigurationen

B.1. *jpeg-compressor*

```
{ "targets": [
  {
    "id": "primitive_bool_result",
    "interface": {
      "namespace": "jpge",
      "className": "jpeg_encoder",
      "funcName": "read_image",
      "resultType": "bool"
    },
    "injectAt": "result"
  },
  {
    "id": "primitive_bool_argument",
    "interface": {
      "namespace": "jpge",
      "className": "jpeg_encoder",
      "funcName": "code_block",
      "argType": "bool"
    },
    "injectAt": "argument"
  },
  {
    "id": "primitive_int_result",
    "interface": {
      "namespace": "jpgd",
      "funcName": "decode",
      "resultType": "int"
    },
    "injectAt": "result"
  },
  {
    "id": "primitive_int_argument",
    "interface": {
      "namespace": "jpge",
      "funcName": "subsample",
      "argType": "int"
    },
    "injectAt": "argument"
  },
  {
    "id": "primitive_int_argument2",
    "interface": {
      "namespace": "jpge",
      "funcName": "set_px",
      "argType": "int"
    },
    "injectAt": "argument"
  },
  {
    "id": "primitive_float_argument",
    "interface": {
      "namespace": "jpge",
      "funcName": "set_px",
      "argType": "float"
    },
    "injectAt": "argument"
  },
  {
    "id":
    ↪ "primitive_charArray_argument",
    "interface": {
      "namespace": "jpge",
      "funcName":
    ↪ "compress_image_to_jpeg_file",
      "argType": "const char*"
    },
    "injectAt": "argument"
  },

  {
    "id": "typedef_argument",
    "interface": {
      "namespace": "jpge",
      "funcName":
    ↪ "put_signed_int_bits",
      "argType": "jpge::uint"
```

```
74      },
75      "injectAt": "argument"
76    },
77    {
78      "id": "enum_result",
79      "interface": {
80        "namespace": "jpgd",
81        "resultType":
      ↪  "jpgd::jpgd_status"
82      },
83      "injectAt": "result"
84    },
85    {
86      "id": "enum_argument",
87      "interface": {
88        "namespace": "jpgd",
89        "funcName": "stop_decoding",
90        "argType": "jpgd::jpgd_status"
91      },
92      "injectAt": "argument"
93    },
94    {
95      "id": "object_argument",
96      "interface": {
97        "funcName": "subsample",
98        "argType": "jpge::image*"
99      },
100     "injectAt": "argument"
101   },
102   {
103     "id": "malloc",
104     "interface": {
105       "resultType": "void*",
106       "pointCutExp": "%
      ↪  ...::malloc(...)"
107     },
108     "injectAt": "result"
109   }
110 ] }
```

Listing B.1: Konfiguration der Interfaces für *jpeg-compressor*

```
1  {
2    "customIncludes": ["jpge.h", "jpgd.h"],
3    "buildCommand": ["make"],
4    "programPath": "./encoder",
5    "goldenRunParams": ["images/ship.png", "images/ship_golden_50.jpg", "50"],
6    "workloadParams": ["images/ship.png", "images/ship_experiment_50.jpg", "50"],
7    "comparisonCommand": ["images/comp_jpeg.py"],
8    "logLevel": 3,
9    "experiments": [
10     {"num": 1, "target": "primitive_bool_result", "injection_mode": "replace",
     ↪  "errorValue": "false", "callCount": 1},
11     {"num": 2, "target": "primitive_bool_result", "injection_mode": "invert",
     ↪  "callCount": 1},
12     {"num": 3, "target": "primitive_bool_argument", "injection_mode":
     ↪  "replace", "errorValue": "false", "probability": 5, "seed": 24317},
13     {"num": 4, "target": "primitive_bool_argument", "injection_mode": "invert",
     ↪  "probability": 5, "seed": 24317},
14     {"num": 5, "target": "primitive_int_result", "injection_mode": "replace",
     ↪  "errorValue": "-1", "callCount": 1},
15     {"num": 6, "target": "primitive_int_result", "injection_mode": "invert",
     ↪  "callCount": 1},
16     {"num": 7, "target": "primitive_int_result", "injection_mode": "offset+",
     ↪  "callCount": 1},
17     {"num": 8, "target": "primitive_int_result", "injection_mode": "offset-",
     ↪  "callCount": 1},
18     {"num": 9, "target": "primitive_int_argument", "injection_mode": "replace",
     ↪  "errorValue": "0", "callCount": 1},
```

```
19      {"num": 10, "target": "primitive_int_argument", "injection_mode": "invert",
    ↪  "callCount": 1},
20      {"num": 11, "target": "primitive_int_argument", "injection_mode":
    ↪  "offset+", "callCount": 1},
21      {"num": 12, "target": "primitive_int_argument", "injection_mode":
    ↪  "offset-", "errorValue": "0", "callCount": 1},
22      {"num": 13, "target": "primitive_int_argument2", "injection_mode":
    ↪  "offset-", "probability": 1, "seed": 2925},
23      {"num": 14, "target": "primitive_float_argument", "injection_mode":
    ↪  "invert", "probability": 1, "seed": 2925},
24      {"num": 15, "target": "primitive_charArray_argument", "injection_mode":
    ↪  "replace", "errorValue": "\"nil.0\"", "callCount": 1},
25      {"num": 16, "target": "primitive_charArray_argument", "injection_mode":
    ↪  "replace", "errorValue": "(const char[]){'n', 'i', 'l', '.', '1'}",
    ↪  "callCount": 1},
26      {"num": 17, "target": "typedef_argument", "injection_mode": "replace",
    ↪  "errorValue": "0", "probability": 2, "seed": 20981},
27      {"num": 18, "target": "enum_result", "injection_mode": "replace",
    ↪  "errorValue": "jpgd::JPGD_TOO_MANY_COMPONENTS", "callCount": 1},
28      {"num": 19, "target": "enum_result", "injection_mode": "replace",
    ↪  "errorValue": "jpgd::JPGD_TOO_MANY_COMPONENTS", "each_occurrence_once":
    ↪  true},
29      {"num": 20, "target": "enum_argument", "injection_mode": "replace",
    ↪  "errorValue": "jpgd::JPGD_TOO_MANY_COMPONENTS", "callCount": 1},
30      {"num": 21, "target": "object_argument", "injection_mode": "replace",
    ↪  "errorValue": "nullptr", "callCount": 1},
31      {"num": 22, "target": "object_argument", "injection_mode": "replace",
    ↪  "errorValue": "new jpge::image()", "callCount": 1},
32      {"num": 23, "target": "malloc", "injection_mode": "replace", "errorValue":
    ↪  "nullptr", "probability": 75, "seed": 4096},
33      {"num": 24, "target": "malloc", "injection_mode": "replace", "errorValue":
    ↪  "nullptr", "each_occurrence_once": true}
34    ]
35  }
```

Listing B.2: Konfiguration der Kampagne für *jpeg-compressor*

B.2. *lzham_codec*

```
{ "targets": [
  {
    "id": "init_result",
    "interface": {
      "namespace": "lzham",
      "className": "lzcompressor",
      "funcName": "init",
      "resultType": "bool"
    },
    "injectAt": "result"
  },
  {
    "id": "put_bytes_len",
    "interface": {
      "namespace": "lzham",
      "className": "lzcompressor",
      "funcName": "put_bytes",
      "argType": "uint"
    },
    "injectAt": "argument"
  },
  {
    "id": "put_bytes_buf",
    "interface": {
      "namespace": "lzham",
      "className": "lzcompressor",
      "funcName": "put_bytes",
      "argType": "const void*"
    },
    "injectAt": "argument"
  },
  {
    "id": "put_bytes_result",
    "interface": {
      "namespace": "lzham",
      "className": "lzcompressor",
      "funcName": "put_bytes",
      "resultType": "bool"
    },
    "injectAt": "result"
  }
] }
```

Listing B.3: Konfiguration der Interfaces für *lzham_codec*

```
{
  "customIncludes": ["lzham.h"],
  "buildCommand": ["make", "lzhamtest"],
  "programPath": "bin_linux/lzhamtest",
  "goldenRunParams": ["-m3", "c", "bin_linux/ac++", "bin_linux/golden"],
  "workloadParams": ["-m3", "c", "bin_linux/ac++", "bin_linux/test"],
  "comparisonCommand": ["bin_linux/comp_lzham.py"],
  "logLevel": 3,
  "experiments": [
    {"num": 1, "target": "init_result", "injection_mode":"invert",
 ↪  "each_occurrence_once": true},
    {"num": 2, "target": "put_bytes_len", "injection_mode":"offset+",
 ↪  "probability": 20, "seed": 8427},
    {"num": 3, "target": "put_bytes_buf", "injection_mode": "replace",
 ↪  "errorValue": "nullptr", "probability": 10, "seed": 8427},
    {"num": 4,"target": "put_bytes_result", "injection_mode":"invert",
 ↪  "probability": 10, "seed": 8427}
  ]
}

```

Listing B.4: Konfiguration der Kampagne für *lzham_codec*