Fachhochschule Westküste
Hochschule für Wirtschaft & Technik

Entwurf und Implementierung eines Fuzzing Frameworks für das verteilte Tasking Framework des Deutschen Zentrums für Luft- und Raumfahrt

Dennis Pfau

Masterarbeit im Studiengang Automatisierungstechnik

bei

Prof. Dr. Kristina Schädler

M.Sc. Moritz Ulmer

Automatisierungstechnik

29.09.2017

11. Fachsemester

Inhaltsverzeichnis

Verzeichnis der Abbildungen

Verzeichnis der Tabellen

Verzeichnis der Abkürzungen

Abkürzung	Bedeutung
DLR	Deutsches Zentrum für Luft- und Raumfahrt
SAGE	Scalable Automated Guided Execution
IoT	Internet of Things
ScOSA	Scalable On-Board Computing for Space Avionics
OBC-NG	On-board Computer - Next Generation
COTS	commercial oft-the-shelf
HPN	High Performance Nodes
TCP/IP	Transmission Control Protocol/Internet Protocol
ID	Identifikator
SQL	Structured Query Language
BLOB	Binary Large Object
ATON	Autonomous Terain-based Optical Navigation

1 Einleitung

An Missionen in der Raumfahrt werden immer höhere Anforderungen gestellt, was dazu führt, dass der Umfang und die Komplexität der in der Raumfahrt verwendeten Software stetig steigt. Diese Software kann Fehler aufweisen und stellt eine, sich mit dem Umfang der Software verstärkende Fehlerquelle dar. Aus diesem Grunde werden verstärkte Anstrengungen unternommen, Softwarefehler durch den Einsatz geeigneter Testmethoden zu erkennen. Eine solche Methode stellt das Fuzzing dar. Hierbei wird versucht, Softwarefehler durch die Manipulation von Daten zu detektieren.

In dieser Masterarbeit soll untersucht werden, wie Fuzzing auf das, vom Deutschen Zentrum für Luft- und Raumfahrt (DLR) entwickelten Tasking Framework angewendet werden kann. Dieses wird in Rahmen des ScOSA Projektes als Middleware für eine neue Generation von On-Board Computern entwickelt. Die Masterarbeit beginnt mit der Recherche zu verschiedenen Ansätzen zum Fuzzing und den Bestandteilen von Fuzzing Frameworks. Anschließend wird das Tasking Framework des ScOSA-Projektes erörtert. Dabei wird ebenfalls der neue On-Board Rechner vorgestellt.

Das Testen der Anwendung erfolgt durch das in der Masterarbeit entworfene Fuzzing Framework. Dieses erlaubt das Erstellen der Master-Slave-Software, welche aus zwei Komponenten besteht. Die Slave Software ist mit der zu testenden Anwendung gekoppelt und übernimmt das Fuzzing der Daten. Sie soll beim Testen zusammen mit der Anwendung auf dem On-Board Rechner ausgeführt werden. Die Master Software wird separat auf einem Desktop Rechner ausgeführt. Sie steuert die Slave Software und Speichert die Ergebnisse in einer SQL-Datenbank ab.

Die beim Fuzzing benötigte Manipulation der Daten erfolgt Anhand von Regeln, welche im Fuzzing Framework definiert sind. Jede Regel entspricht dabei einer anderen Anweisung, für die Art der Manipulation. Zusätzlich sind Funktionen vorhanden, welche die zu testende Anwendung überwachen. Sie helfen aufgetretene Fehler zu erkennen und zurückzuverfolgen.

Um die Funktionalität des Fuzzing Frameworks zu beweisen, wird dieses auf mehrere Beispielanwendungen Angewendet.

2 Theoretische Grundlagen zum Fuzzing

Bei Fuzzing handelt es sich um eine Technik zum Testen von Software. Dabei werden manipulierte Daten von außen an die zu testende Software gesendet, um ein fehlerhaftes Verhalten hervorzurufen. Tritt beim Fuzzing ein Fehler auf, wird anschließend versucht, die Ursache des Fehlers zu ermitteln und diesen zu beseitigen. Der Name Fuzzing ist von rauschenden „fuzzy“ Telefonleitungen abgleiten, welche bei frühen Modemanwendung zu Abstürzen von UNIX Anwendungen führten [1].

In diesem Kapitel werden die theoretischen Grundlagen zum Fuzzing behandelt. Beginnend mit einer Unterteilung von Softwaretests, folgt ein Überblick über die Entwicklung von Fuzzing. Anschließend wird der grundlegende Aufbau eines Fuzzing Frameworks betrachtet und unterschiedliche Arten von Fuzzing Frameworks klassifiziert. Zum Schluss folgt ein Ansatz für die Bewertung eines Fuzzing Frameworks sowie eine Betrachtung derer Anwendung.

2.1 Unterteilung von Softwaretests

Einen integralen Teil der Entwicklung jeder Software stellt das Testen dar. Dabei wird überprüft, ob die Software den geforderten Ansprüchen entspricht. Eine Herangehensweise an das Testen von Software stellt die Unterteilung von Testmethoden in White-Box-, Black-Box- und Grey-Box-Tests dar [2, pp. 3-19] [3, pp. 79-97]. Als Kriterium dient hier, welche Informationen über der zu testenden Software vorliegen.

2.1.1 White-Box-Test

Unter White-Box-Tests werden alle Testmethoden zusammengefasst, die Zugang zu sämtlichen Informationen der zu testenden Software haben. Dazu zählt der Zugang zum Quellcode der Software. White-Box-Tests erlauben deshalb das direkte Testen am Quellcode mittels „Code Auditing“ [3, pp. 79-97]. Ebenfalls ist das Testen einzelner Teile der Software mittels Unittest möglich. Häufig fällt der Begriff Strukturelles Testen im Zusammenhang mit White-Box-Tests, da direkt die Struktur des Quellcodes untersucht werden kann.

2.1.2 Black-Box-Test

Als Black-Box-Tests werden die Testmethoden bezeichnet, die keine Informationen zum inneren Aufbau der Software haben. Als Informationen stehen nur Ein- und Ausgabedaten der Software zur Verfügung, die von außen sichtbar sind. Black-Box-Tests werden häufig

als Funktionale Test bezeichnet [3, pp. 79-97], da mit ihnen von außen die Funktionalität der Software beobachtet wird. Zu den Black-Box Testverfahren zählt das Fuzzing, welches die Software durch die Eingabe manipulierter Daten von außen testet.

2.1.3 Grey-Box-Test

Hat eine Testmethode einen beschränkten Zugang zum inneren Aufbau der Software, wird dies als Grey-Box-Test bezeichnet [3, pp. 79-97]. Dabei werden Ansätze von Black-Box- und White-Box-Tests miteinander kombiniert. Sie testen die Software meist wie Black-Box-Test über Eingabedaten von außen, nutzen dabei jedoch Kenntnisse zum inneren Aufbau der Software.

2.2 Geschichte des Fuzzing

Fuzzing geht zurück auf die Veröffentlichung eines Artikels von Barton Miller [1]. Dieser hatte in Rahmen eines Projektes an der University of Wisconsin-Madison untersucht, wie robust UNIX-Anwendungen gegen zufällige Eingaben sind. Dabei stürzten 24% der 88 untersuchten Anwendungen ab [1]. Fünf Jahre später wiederholte er diesen Versuch und erweiterte ihn auf X-Windows- und Netzwerkanwendungen. Dabei konnte er zum Teil 40% der Anwendungen zum Absturz bringen [4].

Ari Takanen weist darauf hin, dass es bereits in den 1980 Jahren Anwendungen gab, die als Fuzzing bezeichnet werden können [3, pp. 22-24]. Wie der Abbildung 2.1 zu entnehmen ist, wird dabei „The Monkey“ als Beispiel genannt. Dabei handelt es sich um eine von Steven Capps entwickelte Anwendung für den Apple Macintosh, welche zufällige Eingaben in das System injiziert.

Zwischen 1999 und 2001 folgte daraufhin das PROTOS Projekt der Oulu University Secure Programming Group [5]. Ziel des Projektes war das Testen einer Vielzahl von Netzwerkprotokollen wie HTTP, SNMP und DNS. Dazu wurden Testwerkzeuge für die einzelnen Netzwerkprotokolle entwickelt und deren Vertreibern zur Verfügung gestellt. Nach der Behebung gefundener Fehler wurden die Werkzeuge anschließend veröffentlicht [3, pp. 22-24] [6].

Das PROTOS Projekt unterscheidet sich von den ersten Versuchen von Berton Miller dahingehend, dass beim Erstellen der manipulierten Eingabedaten die Datenstruktur der Netzwerkprotokolle berücksichtigt wird. Dieser Ansatz wird als „grammar based Fuzzing“. [7] bezeichnet und erlaubt ein effizienteres und zielgerichtetes Testen der Netzwerkprotokolle.

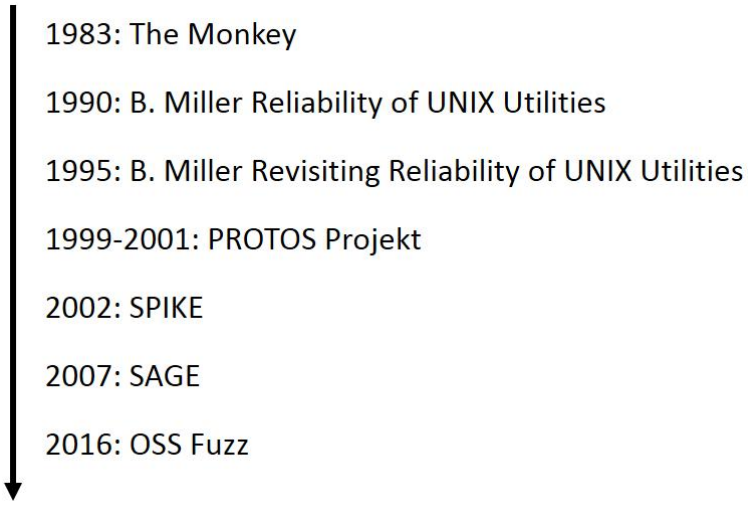

Abbildung 2.1: Zeitstrahl der Geschichte vom Fuzzing
Quelle: Eigene Zeichnung

Einen ähnlichen Ansatz verfolgte das 2002 von Dave Aitel veröffentlichte Fuzzing Framework SPIKE [8]. Die Besonderheit von SPIKE ist die Verwendung eines „block-based“ Ansatzes zum Modellieren von Netzwerkprotokollen. Aus diesen Modellen erzeugt SPIKE anschließend Eingabedaten zum Testen.

Den nächsten Schritt bildet das 2007 von Microsoft entwickelte Werkzeug SAGE (Scalable Automated Guided Execution). Es verwendet dabei einen Ansatz, welcher als „whitebox fuzz testing“ bezeichnet wird [9]. SAGE hat dabei Zugang zum Quellcode der zu testenden Software. Damit wird bei der Ausführung der Software überwacht, welche Codepfade durchlaufen wurden. Daraus zieht SAGE Rückschlüsse, welche Eingabedaten zum Durchlaufen bestimmter Codepfade führen. Aus den Rückschlüssen werden anschließend neue Eingabedaten erstellt und das Programm erneut ausgeführt und überwacht. Ziel ist es, durch das Wiederholen in einem Optimierungsprozess, Eingabedaten zu erzeugen, welche mit hoher Wahrscheinlichkeit Fehler finden.

Im Jahr 2016 startete Google das Projekt OSS-Fuzz – Continiuos Fuzzing for Open Source Software [10]. Ziel des Projektes ist es, eine Online-Plattform für das Testen von Open-Source Software mittels Fuzzing bereitzustellen. Der Entwickler der Software definiert dazu lediglich Schnittstellen, an denen die Dateneingabe erfolgen soll. Das Testen selbst erfolgt dabei auf Servern von Google.

2.3 Bestandteile eines Fuzzing Framework

Zur allgemeinen Beschreibung eines Fuzzing Frameworks kann dieses in drei Bestandteile aufgeteilt werden. [6] Diese sind der Testgenerator zum Erstellen der Eingabedaten, das Interface für die Eingabe der Daten und die Überwachung der zu testenden Software.

2.3.1 Testgenerator für die manipulierten Eingabedaten

Fuzzing testet eine Software durch die Eingabe manipulierter Daten. Für die Erzeugung dieser manipulierten Daten ist der Testgenerator zuständig. Die Testgeneratoren einzelner Fuzzing Frameworks unterscheiden sich im Ansatz und der Komplexität, mit der die Daten erstellt werden [3, pp. 137-166]. Dabei kann auch ein einzelnes Fuzzing Framework unterschiedliche Ansätze verfolgen [6]. In Kapitel 2.4 werden unterschiedliche Arten von Fuzzing vorgestellt, welche sich primär durch den Ansatz bei der Datenerstellung unterscheiden.

2.3.2 Interface für die Dateneingabe

Nach dem der Testgenerator manipulierte Eingabedaten erstellt hat, müssen diese der zu testenden Software zugeführt werden. Dies erfolgt über das Interface des Fuzzing Frameworks [3, pp. 137-166]. Da das Interface die Schnittstelle zu der zu testenden Software darstellt, muss diese meist für jede zu testende Software speziell erstellt werden. Ein Interface kann dabei aus einer einfachen Datei bestehen, die von der zu testenden Software eingelesen wird [1]. Andere Anwendungen können komplexere Interfaces erfordern. So kann das Testen von Netzwerkprotokollen die Eingabe von manipulierten Daten in ein simuliertes Netzwerk aus Server und Client erfordern [7].

2.3.3 Überwachung des zu testenden Systems

Das Ziel von Fuzzing ist das Erzeugen von Fehlern in der zu testenden Software. Deshalb muss das Fuzzing Framework in der Lage sein, aufgetretene Fehler zu erkennen. [3, pp. 167-197]. Die einfachste Form der Überwachung ist die Überprüfung, ob die Software nach Eingabe der Daten mit einem Fehler beendet wurde. Eine umfangreichere Überwachung ist mit Hilfe von Debuggern möglich, welche die zu testende Software bei deren Ausführung überwachen. Eine weitere wichtige Aufgabe der Überwachung ist die Rückverfolgung aufgetretener Fehler. Kann zurückverfolgt werden, an welcher Stelle der Software der Fehler auftritt, vereinfacht dies die spätere Behebung oder macht diese erst möglich.

2.4 Klassifizierung von Fuzzing Frameworks

Seit den ersten Anfängen des Fuzzing unter Miller [1], wurden viele unterschiedliche Arten von Fuzzing Frameworks entwickelt. Nach Takanen kann eine Klassifizierung durch zwei Kriterien erfolgen [3, pp. 137-166] [6]. Das erste Kriterium unterscheidet die Frameworks nach der Herkunft der manipulierten Eingabedaten. Im zweiten Kriterium wird dann entschieden, wie „Intelligent“ [6] die Daten anschließend manipulierte werden.

2.4.1 Herkunft der manipulierten Daten

Ein Kriterium für die Unterscheidung von Fuzzing Frameworks ist die Herkunft der zu manipulierenden Daten. Es werden zwei Herkünfte unterschieden [2, pp. 33-44]. Ein Teil der Fuzzing Frameworks verwendet korrekte Eingabedaten der zu testenden Software als Grundlage. Diese werden anschließend vom Fuzzing Framework manipuliert und als Eingabedaten verwendet. Dabei kann es sich um Daten wie JPEG-Bilder oder im Netzwerk mitgeschnittene Netzwerkprotokolle handeln.

Der zweite Teil der Fuzzing Frameworks erzeugt eigenständig komplett neue Eingabedaten. Die Erzeugung erfolgt dabei nach zuvor im Framework definierten Regeln und erzeugt meist Eingabedaten eines bestimmten Dateityps oder Protokolls.

2.4.2 Ansätze für die Manipulation der Eingabedaten

Das zweite Kriterium zur Einordnung der Fuzzing Frameworks ist der Aufwand für die Manipulation der Eingabedaten. Dies wird auch als „Intelligenz“ [6] bezeichnet, mit der das Fuzzing Framework arbeitet.

2.4.2.1 Rein zufällige Manipulation

Die am wenigsten intelligenten Fuzzing Frameworks nutzen zufällige Manipulationen. Sie besitzen keine Informationen über die Art oder Struktur der Eingabedaten und manipulieren diese durch zufällige Werteänderungen an zufälligen Stellen. Ein Beispiel für diesen Ansatz sind die 1990 von Miller verwendeten Eingabedaten zum Testen von UNIX Anwendungen [1].

2.4.2.2 Manipulation nach Regeln

Die nächste Kategorie von Fuzzing Framework verwendet einen Grundsatz von Regeln, welche angeben wie die Eingabedaten zu manipulieren sind. Die Regeln können sich dabei vom Aufbau eines Netzwerkprotokolls oder der Struktur eines Datentyps ableiten. Sie legen dabei fest, welche Teile der Eingabedaten wie zu manipulieren sind und wie deren Struktur auszusehen hat. Als Beispiel für eine Manipulation nach Regeln ist SPIKE [8].

2.4.2.3 Verwendung von Optimierungsverfahren

Einen weiteren Schritt Richtung „Intelligenten“ Fuzzing Framework stellt die Verwendung von Optimierungsverfahren dar. In Gegensatz zur Manipulation der Eingabedaten nach einfachen Regeln aus Kapitel 2.4.2.2, erhält das Fuzzing Framework eine Rückmeldung von der zu testenden Software. Diese Rückmeldung wird anschließend von einem Optimierungsalgorithmus verwendet, um neue Eingabedaten zu erzeugen. Vom Autor Takanen wird dafür ein Evolutionärer Algorithmus vorgestellt [3, pp. 201-219]. Als Rückgabewert wird hierbei der Prozentsatz der zu testenden Software zurückgegeben, welcher nach der Eingabe der manipulierten Eingabedatei ausgeführt wurde. Ziel ist es, einen möglichst großen Teil des Programmcodes auszuführen.

2.4.2.4 Verwendung des Quellcodes

Nach der Unterteilung von Softwaretests in Kapitel 2.1 gehört Fuzzing zu den Black-Box-Tests. Informationen über die zu testenden Softwares erhalten sie somit nur von außen. Es existieren jedoch Fuzzing Frameworks, die mit diesem Prinzip brechen. Somit hat das Framework Einblick in die innere Struktur der zu testenden Software. Dieser Ansatz wird als Grey-Box-Fuzzing bezeichnet, da damit Black-Box und White-Box Ansätze gemischt werden. Beim vollen Zugang zum Quellcode wird ebenfalls der Begriff White-Box-Fuzzing verwendet [9]. Ein Beispiel ist das SAGE Projekt von Microsoft. SAGE verwendet dabei den Zugang zum Quellcode, um beim Einlesen der manipulierten Eingangsdaten das Verhalten der Software zu überwachen. Dabei werden sämtliche durchlaufende Codepfade aufgezeichnet. Nach der Auswertung der aufgezeichneten Codepfade werden anschließend neue Eingabedaten erstellt. Ziel ist es erneut, einen möglichst großen Teil des Programmcodes auszuführen. Durch den Zugang zum Quellcode können die Datenwerte zusätzlich dahingehend geändert werden, dass sie mit größerer Wahrscheinlichkeit einen Fehler aufdecken.

2.5 Bewertung eines Fuzzing Frameworks

Das Ziel eines Fuzzing Framework ist das Auffinden von Fehlern in Software. Deren Effektivität hängt von einer Vielzahl von Faktoren ab, welche sich abhängig von der zu testenden Software ändern können. Zwei dieser Faktoren sind die Abdeckung des Programmcodes und die Fehlererkennung pro Testdurchlauf.

2.5.1 Abdeckung des Programmcodes

Das Testen mit Fuzzing erfolgt über die Eingabe manipulierter Daten in die zu testende Software. Die Fehler werden dadurch aufgedeckt, dass die Eingabedaten ein anormales Verhalten der Software, bei deren Bearbeitung auslöst. Da die Eingabe der Daten von außen in die Software erfolgt, werden nur die Teile des Programmcodes getestet, welche von den Eingabedaten erreicht werden [2, pp. 61-69]. Das optimale Ziel jedes Fuzzing Frameworks ist eine möglichst vollständige Abdeckung des Programmcodes. Einige Ansätze für das Fuzzing, wie der in Kapitel 2.4.2.3 vorgestellte Evolutionäre Algorithmus, optimieren die Eingabedaten dahingehend, eine möglichst große Abdeckung des Programmcodes zu erreichen.

2.5.2 Fehlererkennung pro Testdurchlauf

Fuzzing entdeckt Fehler durch die Eingabe manipulierter Daten. Um die Anzahl der endeckten Fehler zu erhöhen, wird dieser Vorgang mit unterschiedlichen Eingabedaten wiederholt. Ein Ansatz ist deshalb, die Anzahl der Durchläufe mit unterschiedlichen Eingabedaten zu erhöhen. Die alleinige Erhöhung der Durchläufe ist jedoch nicht ausreichend. Dies wird am Beispiel der if-Bedingung „if(x==10) then“ deutlich [9]. Diese Bedingung ist nur dann wahr, wenn der 32-Bit Datenwert den Wert „10“ besitzt. Somit beträgt die Wahrscheinlichkeit, den Programmcode beim Fuzzing mit zufälligen Datenwerten auszuführen, nur 1 zu 2^{32}. Deshalb versuchen viele Fuzzing Frameworks die Datenwerte nicht zufällig, sondern wie in Kapitel 2.4.2 aufgeführt, „intelligent“ zu ändern. Das Ziel ist dabei, mit dem intelligenten Ansatz Eingabedaten zu erzeugen, welche eine große Programmcodeabdeckung gewährleisten und eine hohe Wahrscheinlichkeit haben, einen Fehler der zu testenden Software aufzudecken.

2.6 Anwendung von Fuzzing

Wie zuvor erläutert dient Fuzzing dem Aufspüren von Fehlern in einer Software. Dabei besitzt Fuzzing Vor- und Nachteile, welche sich primär aus dem Black-Box-Ansatz ableiten lassen. Abhängig davon haben sich die Anwendungsfelder von Fuzzing entwickelt.

2.6.1 Warum wird Fuzzing verwendet

Fuzzing ist eine Methode zum Testen von Software, welche sich als einfach und effektiv erwiesen hat [6]. Dies wird bei den ersten Tests mit Fuzzing deutliche [1], welche mit einfachen und zufällige erzeugten Eingabedaten eine Vielzahl von Fehlern in UNIX-Anwendun-

gen erzeugten. Ein Vorteil ist der Black-Box Ansatz. Das bedeutet, dass von der zu testenden Software lediglich die Schnittstellen für die Kommunikation nach außen bekannt sein müssen. Es sind keine Kenntnisse zum Inneren der Software erforderlich. Der Quellcode wird nicht benötigt. Wie der Klassifikation verschiedener Fuzzing Frameworks aus Kapitel 2.4 zu entnehmen ist, erlaubt Fuzzing dennoch die Verwendung von Kenntnissen zum inneren der Software, um die Fehlererkennung zu verbessern. Weitere Vorteile des Black-Box Ansatzes sind die Wiederverwendbarkeit und die Automatisierbarkeit von Fuzzing Frameworks [2, pp. 3-19]. Da nur die Schnittstellen der zu testenden Software benötigt werden, kann ein einzelnes Fuzzing Framework unterschiedliche Softwares testen, solange die Schnittstellen identisch sind. Dies ist zum Beispiel der Fall, wenn unterschiedliche Programme MP4-Dateien einlesen. Die Erzeugung von manipulierten Eingabedaten und die anschließende Überwachung der zu testenden Software erfolgt anschließend meist automatisch vom Fuzzing Framework.

Die Nachteile von Fuzzing sind die Abdeckung des Programmcodes und die mit steigendem Umfang der zu testenden Software abnehmende Wahrscheinlichkeit, einen Fehler zu finden. In Kapitel 2.5 wird ausführlich auf diese Punkte eingegangen.

2.6.2 Fuzzing in der Softwaresicherheit

Fuzzing kann dann zum Testen einer Software verwendet werden, wenn dieser von außen Daten zugeführt werden. Besonders Parser, welche eingelesene Daten weiterverarbeiten, sind zum Testen mit Fuzzing geeignet. Beispiele dafür sind Softwares, welche Kommandozeilenbefehle oder Bild- und Videodateien einlesen. Ein großes Anwendungsfeld nimmt das Testen von Softwares ein, die über Netzwerke Daten austauschen. Da diese Netzwerke öffentlich zugänglich sein können, kann ein Fehler der Software von außen ausgenutzt werden, um absichtlich ein schädliches Verhalten auszulösen. Fuzzing ist, in Verbindung mit dem Black-Box-Ansatz, ideal für das Aufspüren dieser Fehler. Deshalb wird Fuzzing sowohl von der „black-hat community“ [6], als auch von Softwaretestern, zum Aufspüren dieser Fehler verwendet. Viele Unternehmen nutzen Fuzzing deshalb, um die Sicherheit ihrer Software zu verbessern [9].

Mit der steigenden Digitalisierung der Gesellschaft nimmt die Bedeutung von Softwaresicherheit zu. Durch das Internet of Things (IoT) sind immer mehr Geräte an das Internet angeschlossen. Deshalb wird Fuzzing in den letzten Jahren auch zum Testen von Industrienetzwerken [11] und Autos [12] verwendet.

3 Das ScOSA Projekt des DLR

Das in dieser Masterarbeit entworfene Fuzzing Framework soll im Rahmen des Scalable On-Board Computing for Space Avionics (ScOSA) Projektes des DLR angewendet werden. Ziel des Projektes ist die Entwicklung eines neuen On-Board Computers für Satelliten. Bestandteile des Projektes sind sowohl die Entwicklung der neuen Hardware des On-Board Computers, als auch entsprechender Software. Es baut dabei auf den Erkenntnissen des zuvor erfolgreich abgeschlossenen On-Board Computer - Next Generation(OBC-NG) Projektes auf. [13]

Die Besonderheit des ScOSA Projektes ist die Verwendung von commercial off-the-shelf (COTS) Hardwarekomponenten, die nicht speziell für die Raumfahrt entwickelt wurden. Der Vorteil von COTS Hardware ist deren stärkere Rechenleistung, welche die Realisierung eines leistungsstärkeren On-Board Computers erlaubt. [13] Der Nachteil dieser Hardware ist deren höhere Fehleranfälligkeit unter den Umweltbedingungen des Weltraums. Um diese zu kompensieren, wird unter anderem das Tasking Framework des DLR verwendet. Dies erlaubt eine Fehlerkorrektur auf Software Ebene.

In diesem Kapitel wird die fürs Fuzzing Framework relevante Hard- und Software des ScOSA Projektes vorgestellt.

3.1 Hard- und Software des ScOSA-Boards

Als Hardwareplattform für das ScOSA-Projekt wird das ScOSA-Board verwendet. Dabei handelt es sich um ein Demo-Board, welches die Hardware des zu entwickelnden On-Board Computers simuliert. Jedes ScOSA Board besitzt drei voneinander unabhängige High Performance Nodes(HPN). Jeder HPN stellt eine Recheneinheit dar und besteht aus einer ARM cortex A9 CPU als Hauptprozessor und einem FPGA als Koprozessor. Die Kommunikation der HPN nach außen erfolgt über SpaceWire [14] und Ethernet. Die HPNs des ScOSA Boards sind identisch mit den HPNs des späteren On-Board Computers.

Als Betriebssystem verwendet das ScOSA-Board wahlweise eines der beiden Betriebssysteme RTEMS [15] und Linux. RTEMS ist ein Open-Source Echtzeitbetriebssystem, welches für Embedded-Anwendungen entwickelt wurde. Das zweite Betriebssystem ist eine mit Hilfe des Yocto Projektes [16] erstellte Linux Distribution. Die vom ScOSA-Project verwendete Distribution richtet sich nach der Beispiel-Distribution Poky [17]. Der Vorteil von Linux gegenüber RTEMS ist die mögliche Verwendung von umfangreicheren Programmbibliotheken.

3.2 Aufbau des Tasking Frameworks

Das Tasking Framework wurde vom DLR als Middleware zwischen Betriebssystem und Anwendung entwickelt. [18] Im Rahmen des ScOSA-Projektes kommt es als Middleware auf dem On-Board Computer zum Einsatz. Für die Verwendung des Tasking Frameworks wird die Anwendung bei deren Entwicklung in einzelne Komponenten aufgeteilt. Dabei wird die Speicherung der Daten von deren Verarbeitung getrennt. Die Verarbeitung der Daten erfolgt in den Tasks, während die Datenspeicherung in den TaskChannels erfolgt. Das Tasking Framework übernimmt anschließend die Verwaltung der einzelnen Tasks, wie das Starten der Tasks und deren Datenaustausch untereinander.

Für das ScOSA-Projekt wurde das Tasking Framework so erweitert, dass die einzelnen Tasks auf verschieden HPNs des On-Board Computers parallel ausgeführt werden können. Somit kann die Anwendung automatisch auf die Rechenressourcen mehrerer HPNs zurückgreifen, ohne dass dies bei der Entwicklung der Anwendung berücksichtigt werden muss. Dies erlaubt die redundante Ausführung einer Anwendung.

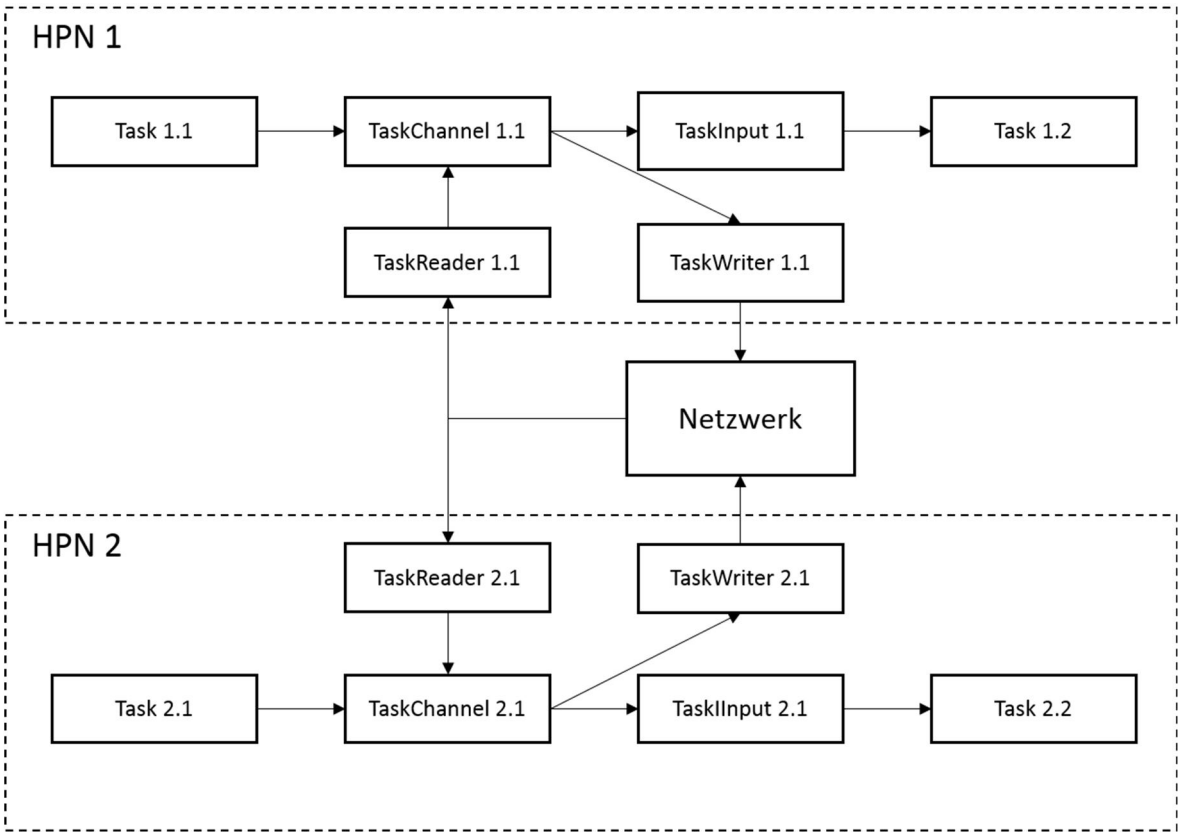

Abbildung 3.1: Datenflussdiagramm des Tasking Frameworks

Quelle: Eigene Zeichnung

Abbildung 3.1 zeigt das Datenflussdiagramm zwischen den einzelnen Komponenten des Tasking Frameworks. Zu erkennen sind die Komponenten, welche auf einem einzelnen HPN ausgeführt werden. Dazu ist anzumerken, dass die einzelnen Komponenten auf beiden HPNs identisch sind. Task 1.2 und Task 2.2 sind ein und derselbe Task, welcher einmal auf HPN 1 und einmal auf HPN 2 ausgeführt wird. Abbildung 3.1 zeigt darüber hinaus die Kommunikation zwischen den beiden HPNs mittels TaskWriter und TaskReader. Dabei ist TaskWriter für das Senden von Nachrichten über das Netzwerk zuständig, während TaskReader die Nachrichten empfängt. Das Tasking Framework ist in C++ programmiert.

3.2.1 TaskChannel

Der TaskChannel übernimmt im Tasking Framework die Aufgabe der Datenspeicherung. Dafür stellt der TaskChannel den anderen Komponenten ein Interface zur Verfügung, welches das Ablegen und Abrufen von Datenobjekten erlaubt. Die Speicherung einzelner Objekte erfolgt in einer Queue nach dem First In - First Out Prinzip. Bei den einzelnen Objekten kann es sich um komplexe Datenstrukturen handeln, jedoch müssen alle abgespeicherten Objekte vom gleichen Datentyp sein. Wird ein neues Objekt in der Queue gespeichert, informiert der TaskChannel automatisch die mit ihm verbundenen TaskWriter und TaskInput.

Das Tasking Framework besitzt noch eine zweite Art von TaskChannel. Sie werden als TaskEvent bezeichnet. Diese TaskChannel besitzen einen Timer, welcher nach einer bestimmten Zeit ein Ereignis auslöst. Dadurch kann in periodischen Zeitabstand ein Signal an einen TaskInput gesendet werden.

3.2.2 TaskInput

Der TaskInput ist die Verbindungskomponente zwischen einem TaskChannel und einem Task. Wird ein neues Datenobjekt im verbundenen TaskChannel gespeichert, wird der TaskInput darüber Informiert. Dieser sendet anschließen ein Signal an den mit ihm verbunden Task, um diesen zu starten.

3.2.3 TaskWriter und TaskReader

Im Tasking Framework übernehmen der TaskWriter und der TaskReader die Aufgabe des Datenaustausches zwischen den einzelnen HPNs des ScOSA-Boards. Der TaskWriter wartet dabei auf das Signal vom TaskChannel, dass ein neues Datenobjekt abgespeichert wurde. Hat er das Signal erhalten, liest er das Datenobjekt aus dem TaskChannel, um dieses anschließend über das Netzwerk an die anderen HPNs zu schicken.

Für den Empfang des Datenobjektes ist der TaskReader zuständig. Bei der Kommunikation zwischen mehreren HPNs ist er damit das Gegenstück zum TaskWriter. Das empfangene

Datenobjekt wird einem in einem oder mehreren TaskChanneln gespeichert, welche mit ihm verbunden sind.

3.2.4 Task

In den Tasks befindet sich der auszuführende Programmcode der Anwendung, welcher für die Verarbeitung der Daten zuständig ist. Jeder Task ist mit einem oder mehreren TaskInputs verbunden, welche den Task starten. Die Daten für die Verarbeitung erhält der Task von dem TaskChannel, welche mit seinen TaskInputs verbunden sind. Ein Task wird nur dann von einem TaskInputs gestartet, wenn dieser zuvor ein Signal vom verbunden TaskChannel bekommen hat. Somit ist garantiert, dass in diesem TaskChannel ein neues Datenobjekt ist, welches dem Task zur Verfügung steht.

4 Master-Slave-Software des Fuzzing Frameworks

Das vom DLR entwickelte Tasking Framework soll als Middleware für eine Vielzahl von Anwendung zum Einsatz kommen. Das Fuzzing Framework soll es möglich machen, diese mit Hilfe von Fuzzing auf Fehler zu testen. Das Testen erfolgt dabei mittels der Master-Slave-Software, welche im Rahmen diese Masterarbeit entwickelt wurde.

In diesem Kapitel wird die Master-Slave Software vorgestellt. Im ersten Teil werden die Qualitätsziele und Rahmenbedingungen für das Fuzzing Framework aufgelistet. Anschließend wird die darauf aufbauende Master-Slave-Software vorgestellt. Dabei werden einzelne Komponenten der Software näher erläutert. Dies sind jeweils die Master und Slave Komponente und das Interface zum Tasking Framework. Im letzten Teil wird auf die SQL-Datenbank eingegangen.

Der Quellcode des Fuzzing Framework befindet sich im digitalen Anhang B1.

4.1 Qualitätsziele und Rahmenbedingungen des Fuzzing Frameworks

Das Ziel vom Fuzzing Framework ist das Testen von Anwendungen, welche mit dem Tasking Framework des DLR realisiert wurden. Das Festlegen von Qualitätszielen und Rahmenbedingungen erlaubt es, das zu entwickelnde Fuzzing Framework genauer zu definieren. In Tabelle 4.1 sind die Qualitätsziele für das Fuzzing Framework aufgelistet. Sie geben Auskunft über die Schwerpunkte, welche für die Entwicklung festgelegt wurden.

Tabelle 4.1: Auflistung der Qualitätsziele für das Tasking Framework

Qualitätsziel	Erläuterung
Allgemeine Anwendbarkeit	Das Fuzzing Framework soll zum Testen aller Anwendungen geeignet sein, welche mit Hilfe des Tasking Frameworks realisiert werden. Dies ist entscheidend, da das Tasking Framework als Middleware für eine Vielzahl von unterschiedlichen Anwendungen verwendet wird.
Geringer Eingriff in das Tasking Framework	Das Testen der Anwendung soll unter möglichst realen Bedingungen stattfinden. Deshalb soll die Funktionalität des Tasking Frameworks, neben der beabsichtigten Injektion von Fehlern, nicht beeinfluss werden.
Geringer Aufwand	Der Aufwand für die Erstellung eines Testes für eine spezielle Anwendung soll so gering wie möglich sein. Der Großteil des Fuzzing Framework soll so beschaffen sein, dass er

	ohne Anpassung von verschieden Anwendungen verwendet werden kann.
Hoher Automatisierungsgrad	Die Durchführung des Testens soll automatisch geschehen. Der Benutzer soll lediglich den Test erstellen und anschließend die Ergebnisse auswerten.

Zusätzlich zu den Qualitätszielen, existieren noch Rahmenbedingungen für das Fuzzing Framework. Diese sind in Tabelle 4.2 aufgelistet. Aus den Rahmenbedingungen geht hervor, welche Vorgaben das Fuzzing Framework erfüllen muss. Sie grenzen darüber hinaus den Umfang des Fuzzing Frameworks ein.

Tabelle 4.2: Auflistung der Rahmenbedingungen für die Masterarbeit

Rahmenbedingung	Erläuterung
Verwendung des ScOSA Boards	Die zu testende Anwendung soll während des Testes auf dem ScOSA Board (siehe Kapitel 3.1) ausgeführt werden. Dies soll das Testen unter möglichst realen Bedingungen ermöglichen. Durch die Verwendung des ScOSA Boards sind die zur Verfügung stehenden Hardwareressourcen begrenzt. Ebenfalls hat dies Einfluss auf die zu programmierende Software. Das Betriebssystem und die Programmiersprache sind mit Linux und C++ vorgegeben. Eine weitere Einschränkung für den späteren Programmcode ist die statische Verwendung von Arbeitsspeicher.
Auswertung der durchgeführten Tests	Zur Auswertung der durchgeführten Tests sollen die beim Testen durchgeführten Manipulationen gespeichert werden. Die Speicherung muss sicherstellen, dass bei einem aufgetretenen Fehler die verursachende Manipulation ermittelt werden kann. Eine Visualisierung aufgetretenen Fehler ist nicht Teil dieser Masterarbeit.

4.2 Master-Slave-Software

Nach der Auflistung von Qualitätszielen und Rahmenbedingungen, folgt der Entwurf eines Fuzzing Frameworks, welches diese erfüllt. Darüber hinaus müssen die in Kapitel 2.3 aufgeführten Bestandteile eines Fuzzing Frameworks vorhanden sein. Das Ergebnis ist die Master-Slave-Software.

4.2.1 Architektur der Master-Slave-Software

Den größten Einfluss auf die Architektur der Master-Slave-Software hat die Rahmenbedingung, dass beim Testen das ScOSA-Board verwendet werden soll. Die zu testende Anwendung wird auf diesem ausgeführt. Deshalb muss die für das Testen entwickelte Software, ebenfalls auf dem ScOSA-Board ausgeführt werden. Dieses verfügt jedoch nur über beschränkte Hard- und Softwareressourcen.

Um die Beschränkung der Ressourcen zu umgehen, wird die Master-Slave-Software in zwei Komponenten aufgeteilt. Wie Abbildung 4.1 zu entnehmen ist, sind dies die Slave-Software und die Master-Software. Die Slave-Software wird dabei auf den HPNs des ScOSA Boards ausgeführt. Sie steht dabei über ein Interface in Verbindung mit dem Tasking Framework und somit mit der zu testenden Anwendung. Über das Interface führt sie die Interaktionen aus, welche für das Testen mittels Fuzzing erforderlich sind.

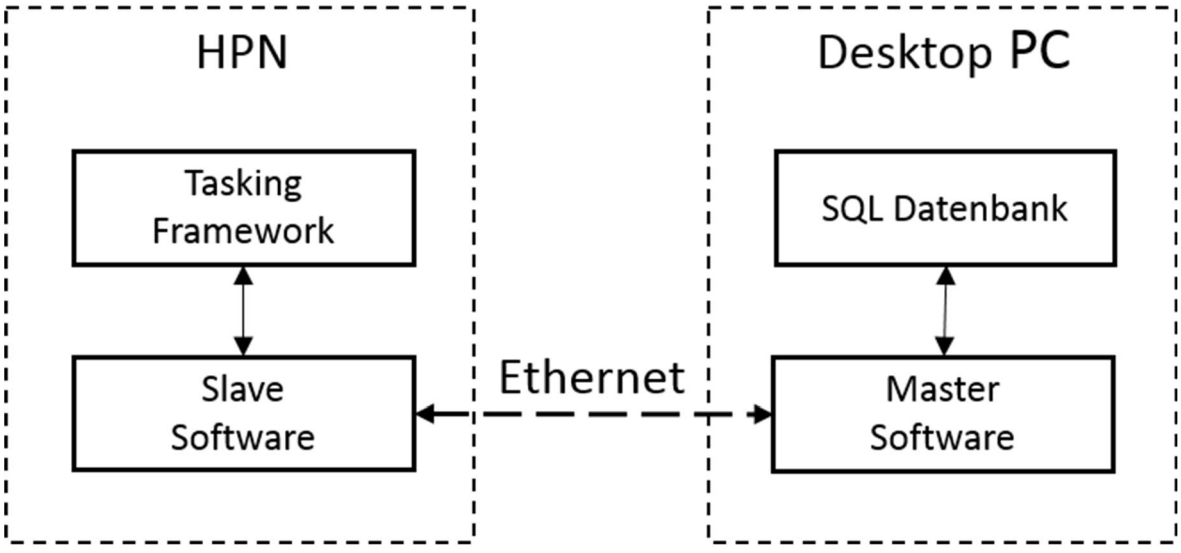

Abbildung 4.1: Datenflussdiagramm der Master-Slave-Software
Quelle: Eigene Zeichnung

Die zweite Komponente ist die Master Software. Sie wird auf einem Desktop PC ausgeführt. Als Desktop PC kann ein Computer mit stärkeren Hardwareressourcen als das ScOSA-

Board verwendet werden. Ebenfalls sind die Verwendung von Linux als Betriebssystem und die statische Nutzung des Arbeitsspeichers nicht mehr vorgegeben.

Die Master-Slave-Software nutzt die zusätzlichen Ressourcen darüber hinaus, um die zweite Rahmenbedingung aus Kapitel 4.1 zu erfüllen. Diese verlangt die Speicherung aller durchgeführten Manipulationen. Dafür wird auf dem Desktop-PC eine SQL-Datenbank verwendet. Der Zugriff auf die Datenbank erfolgt mittels der Master Software, welche die durchgeführten Manipulationen von der Slave-Software erhält. Die Master Software nutzt die Daten darüber hinaus, um Regeln für die Manipulation zu erstellen. Diese werden anschließend an die Slave Software übermittelt. Auf diese Weise steuert die Master Software die Slave Software und das Testen. Die Auftrennung in zwei Komponenten entspricht ebenfalls dem Qualitätsziel, nach dem der Eingriff in das Tasking Framework minimiert werden soll. Die Master Software verbraucht keine Hardwareressourcen des ScOSA-Boards oder stört anderweitig die Funktionalität des Tasking Frameworks.

Alle drei HPNs und der Desktop-PC sind mittels Ethernet miteinander verbunden. Die Verbindung erfolgt über einen Ethernet-Switch. Der Datenaustausch zwischen der Master-Software und den Slave-Softwares erfolgt dabei über Transmission Control Protocol/Internet Protocol (TCP/IP). Dabei sind die einzelnen Slave-Softwares die Clients, welche sich mit dem Server der Master-Software verbinden.

4.2.2 Funktionsweise der Master-Slave Software

Ziel der Master-Slave-Software ist das Testen einer mit dem Tasking Framework realisierten Anwendung mittels Fuzzing. Dazu werden die in den TaskChanneln gespeicherten Datenobjekte manipuliert, bevor diese von den Tasks verarbeitetet werden. Die Manipulation erfolgt dabei durch die Slave Software. Diese ist über das in Kapitel 4.3 beschrieben Interface mit dem Tasking Framework verbunden.

Wie Abbildung 4.2 zu entnehmen ist, führt die Master-Slave-Software im ersten Schritt ein „Profiling Run“ durch. Dabei wird ein kompletter Programmdurchlauf der zu testenden Anwendung durchgeführt, ohne die Datenobjekte zu manipulieren. Versucht ein Task auf das Datenobjekt eines TaskChannel zuzugreifen, schickt die Slave Software eine Kopie des Objektes zur Master Software. Dort wird diese in der SQL-Datenbank gespeichert. Nach dem Programmdurchlauf sind der Master-Slave-Software somit alle Datenobjekte bekannt, welche von den Tasks verarbeitet wurden.

Nach Beendigung des „Profiling Run“, erstellt die Master Software Regeln für die Manipulation von Datenobjekten. Das Erstellen von Regeln wird in Kapitel 5.1 behandelt. Die Regeln werden anschließend zur Slave Software gesendet.

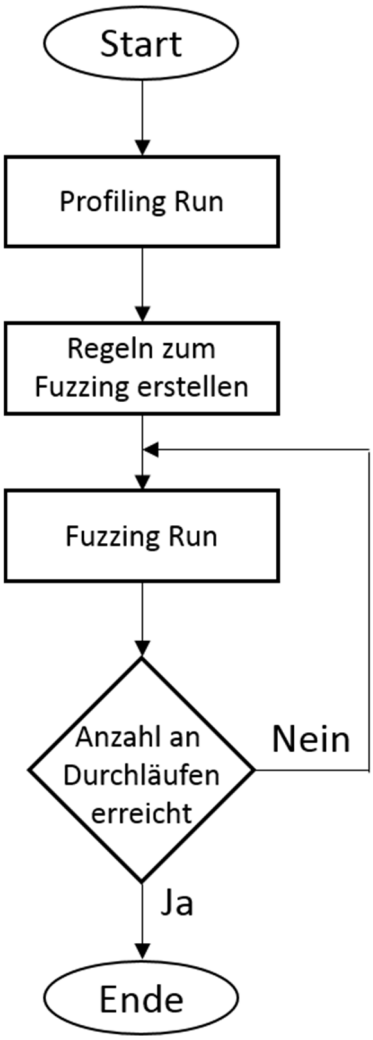

Abbildung 4.2: Flussdiagramm zur Funktionsweise der Master-Slave-Software
Quelle: Eigene Zeichnung

Im letzten Schritt erfolgt das eigentliche Testen der Anwendung im „Fuzzing Run“. Greift ein Task auf das Datenobjekt eines TaskChannels zu, führt die Slave-Software zwei Aktionen aus. In der ersten Aktion wird die Regeln ausgewählt, mit der die Daten manipuliert werden sollen. Anschließend wird das noch nicht manipulierte Datenobjekt zusammen mit der Regel zur Master-Software gesendet und dort in der SQL-Datenbank gespeichert. Dadurch ist sichergestellt, dass die durchgeführten Manipulationen für eine spätere Auswertung der Ergebnisse gespeichert sind. In der zweiten Aktion wird das Datenobjekt nach der ausgewählten Regel manipuliert. Der Task erhält daraufhin das manipulierten Datenobjekt und führt diesen aus.

Im „Fuzzing Run“ wird erneut ein kompletter Programmdurchlauf der zu testenden Anwendung durchgeführt. Wurde ein „Fuzzing Run“ beendet, wird dieser in einer Schleife neu

gestartet. Wie oft ein Neustart erfolgen soll, wird vom Benutzer festgelegt. Ein einzelner „Fuzzing Run“ ist dann beendet, wenn die Anwendung vollständig und fehlerfrei ausgeführt wurde, oder wenn sie wegen eines Fehlers vorzeitig beendet wurde.

4.3 Interface zwischen Slave Software und Tasking Framework

Wie in Kapitel 2.3.2 erläutert, ist das Interface ein Bestandteil eines Fuzzing Frameworks. Es wird für die Eingabe von manipulierten Daten in die zu testende Anwendung verwendet. Das Interface zwischen der Master-Slave-Software und dem Tasking Framework ist Teil der Slave Software. Neben der Eingabe von manipulierten Datenobjekten, wird es zum Auslesen nicht manipulierter Datenobjekte verwendet.

Das Interface besteht aus zwei Teilen. Im ersten Teil wird die Slave Software des Fuzzing Frameworks mit dem Task des Tasking Frameworks aus Kapitel 3.2.3 verbunden. Dies erlaubt den Datenaustausch zwischen den beiden Frameworks. Der zweite Teil beschäftigt sich mit der Serialisierung und Deserialisierung der in den TaskChannel gespeicherten Daten.

4.3.1 Anbindung der Slave Software an das Tasking Framework

Fuzzing testet eine Anwendung durch die Eingabe von manipulierten Daten. Im Tasking Framework werden die Daten in den TaskChannel gespeichert und von den Tasks verarbeitet. Eine weitere Form der Datenspeicherung, wie zum Beispiel in Form einer Datei, ist im Rahmen des ScOSA-Projektes nicht vorgesehen. Die Master-Slave-Software benötigt deshalb Zugang zu den in den TaskChanneln gespeicherten Daten. Der Zugang muss darüber hinaus erfolgen, während die zu testende Anwendung ausgeführt wird.

Die Lösung ist eine direkte Anbindung der Slave Software an das Tasking Framework. Jeder Task besitzt führ jeden TaskChannel, mit dem er über die TaskInputs verbunden ist, einen FifoReader. Dieser ist an einen TaskChannel gebunden und wird vom Task verwendet, um Datenobjekte aus diesem zu entnehmen. Für die Anbindung des Fuzzing Frameworks wird der FifoReader durch eine abgeleitete Version ersetzt. Sie wird als FuzzingReader bezeichnet und besitzt gegenüber dem FifoReader eine Verbindung zur Slave Software. Das Ziel ist es, die in den TaskChannel gespeicherten Datenobjekte immer dann zu manipulieren, wenn der Task über den FuzzingReader auf diesen zugreift. Abbildung 4.3 zeigt die Reihenfolge, in welcher das Datenobjekt zwischen den einzelnen Komponenten ausgetauscht wird. Der FuzzingReader entnimmt das Datenobjekt aus dem TaskChannel und

sendet ihn zur Slave Software. Diese manipuliert das Datenobjekt mittels Fuzzing und sendet es zurück. Anschließend wird das manipulierte Datenobjekt zum Task weitergereicht, welcher dieses verarbeitet.

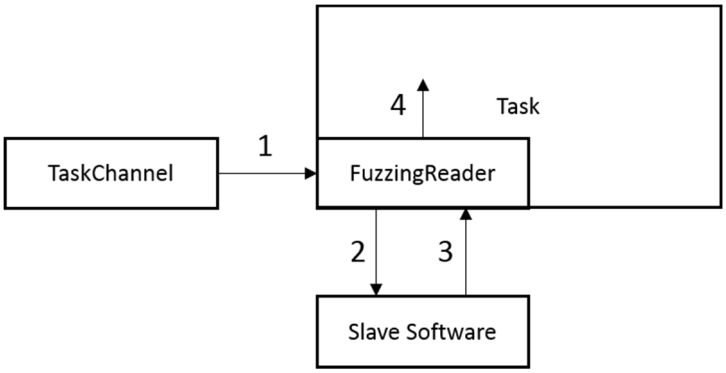

Abbildung 4.3: Datenflussdiagramm zum Interface zwischen Tasking Framework und Slave Software
Quelle: Eigene Zeichnung

Wie in Kapitel 3.2.3 beschrieben, entnimmt neben dem FifoReader auch der TaskWriter Datenobjekte aus dem TaskChannel. Das Fuzzing Framework hat keinen Zugang zu dem Datenaustausch zwischen den einzelnen HPNs und führt an dieser Stelle keine Manipulationen durch. Somit werden die Datenobjekte nur dann manipuliert, wenn ein Task unmittelbar auf diese zugreift. Dies entspricht dem Qualitätsziel aus Kapitel 4.1, wonach der Eingriff ins Tasking Framework zu minimiert ist. Zusätzlich kann dadurch gesteuert werden, welche Datenobjekte manipuliert werden. Die Datenobjekte eines TaskChannels werden nur dann manipuliert, wenn dieser mit einem FuzzingReader verbunden ist.

4.3.2 Serialisierung und Deserialisierung von Datenobjekten

Der TaskChannel speichert einzelne Datenobjekte, wie in Kapitel 3.2.1 beschrieben, in einer Queue. Alle Objekte eines einzelnen TaskChannels sind vom gleichen Datentyp. Eine mit dem Tasking Framework realisierte Anwendung kann jedoch in verschieden TaskChanneln unterschiedliche Datentypen speichern. Die Master-Slave-Software muss in der Lage sein, die verschieden Datenobjekte mittel Fuzzing zu manipulieren. Zusätzlich muss das Datenobjekt über TCP/IP von der Slave Software zur Master Software gesendet werden.

Um dies zu ermöglichen, werden die Datenobjekte in einen seriellen Byte-Stream umgewandelt. Dazu wird das Datenobjekt um zwei Methoden für die Serialisierung und Deserialisierung erweitert. Diese müssen vom Benutzer der Master-Slave-Software für jedes Datenobjekt erstellt werden. Der Aufruf der Methoden erfolgt durch den FuzzingReader. Sendet

dieser ein Datenobjekt zu der Slave Software, wird dieses zuvor in einen Byte-Stream umgewandelt. Die Slave Software erhält anschließend diesen Byte-Stream. Innerhalb der Master-Slave-Software existiert das Datenobjekt ausschließlich in Form dieses Byte-Streams. Schickt die Slave Software den manipulierten Byte-Stream zurück zum FuzzingReader, wandelt dieser den Stream wieder in ein Datenobjekt um.

Damit die Master-Slave-Software mit dem Byte-Streams verschiedener Datenobjekte arbeiten kann, erfolgt deren Serialisierung nach einem festen Protokoll. Dies ist in Abbildung 4.4 dargestellt. Die ersten vier Bytes des Protokolls enthalten die Größe des Datenobjektes in Bytes. Die darauffolgenden zwei Bytes weisen dem Objekt einen eindeutigen Identifikator(ID) zu. Dieser wird benötigt, falls das zu serialisierende Datenobjekt eine komplexe Datenstruktur ist, welche eine Vielzahl an einfachen Datenobjekten wie int und float enthält. Ist dies der Fall, werden die einfachen Datenobjekte nach Abbildung 4.4 einzeln serialisiert Anschließend werden die einzelnen Byte-Streams zu einem Byte-Stream zusammengefasst. Aus DataID kann bei der Deserialisierung zurückverfolgt werden, um welches einfache Datenobjekt es sich handelt. So kann die komplexe Datenstruktur wieder zusammengesetzt werden.

Abbildung 4.4: Protokoll für die Serialisierung eines Datenobjektes
Quelle: Eigene Zeichnung

Das nächste Byte gibt Auskunft über den Datentyp des Datenobjektes. In Tabelle A.1 aus Anhang A1 sind die Datentypen mit ihrem Bytewert aufgeführt. Den Schluss des Byte-Streams bildet der eigentliche Datenwert. Die Größe des Datenwertes ist abhängig vom Datentyp.

Für die Serialisierung wird die Open modUlar sofTware PlatfOrm for SpacecrafT (OUPOST) Programmbibliothek des DLR verwendet [19]. Sie bietet eine Vielzahl von Funktionen wie die Serialisierung und Deserialisierung von Daten. Der Vorteil der Programmbibliothek ist, dass viele Probleme bei der Serialisierung, wie die Byte-Reihenfolge, von dieser automatisch gelöst werden. Das Problem der Byte-Reihenfolge tritt auf, wenn die Prozessoren des ScOSA-Board und des Destop-PC verschiedene Prozessorarchitekturen verwenden.

Unabhängig vom Typ des Datenobjektes, werden diese durch die beschriebene Serialisierung in einer einheitlichen Form an die Slave Software übermittelt. Dadurch wird das Qua-

litätsziel der allgemeinen Anwendbarkeit erfüllt, weil das Fuzzing Framework so mit verschieden Datenobjekten arbeiten kann. Die einheitliche Form der übermittelten Datenobjekte hat darüber hinaus den Vorteil, dass eine Anpassung der Master-Slave-Software an den Datentyp nicht notwendig ist. Der Benutzer muss lediglich die Serialisierung des Datenobjektes im Interface anpassen. Dies erfüllt das Qualitätsziel nach einem geringen Aufwand für die Anpassung des Frameworks an eine spezielle Anwendung.

4.4 Slave Software

Der Fuzzing Slave ist die Komponente der Master-Slave-Software, die auf dem ScOSA-Board ausgeführt wird. Sie ist direkt mit dem Tasking Framework verbunden und ist für die Manipulation der Daten zuständig. Die Manipulation erfolgt dabei nach Regeln, welche sie von der Master Software erhält. Mit dieser ist sie über TCP/IP verbunden.

Wie Abbildung 4.5 zu entnehmen ist, bildet die Slave Klasse den Kern der Slave Software. Sie steuert alle Funktionalitäten der Slave Software. Des Weiteren übernimmt sie die Datenübertragung zum FuzzingReader des Tasking Frameworks, welcher direkt mit ihr Verbunden ist. Teil der Slave Klasse ist die TCPClient Klasse. Sie enthalt alle Funktionalitäten für den Datenaustausch mit der Mastersoftware. In Kapitel 4.6 wird die TCP-Verbindung zwischen der Master- und Slave Software genauer beschrieben.

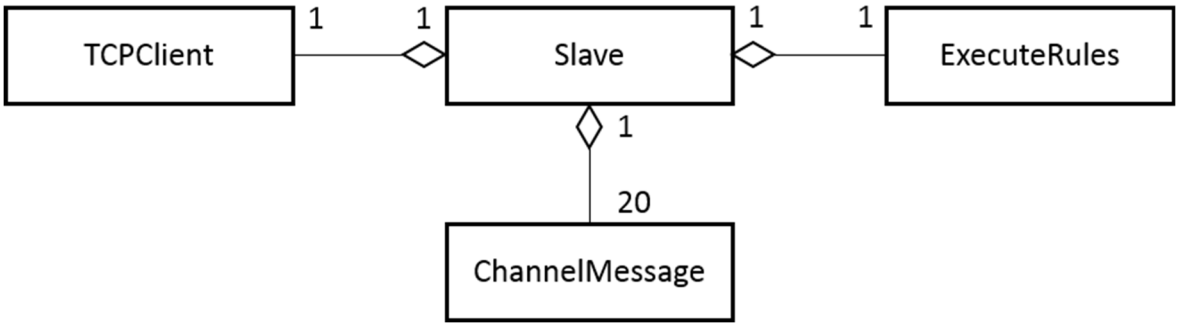

Abbildung 4.5: Klassendiagramm der Slave Software ohne Attribute und Methoden
Quelle: Eigene Zeichnung

Von der ChannelMessage Klasse enthält die Slave Klasse 20 Instanzen. Jede der Instanzen repräsentiert einen FuzzingReader, welcher mit der Slave Software verbunden ist. Pro HPN existiert nur eine Instanz der Slave Klasse, welche für die Manipulation aller FuzzingReader des Tasking Frameworks zuständig ist. Die feste Anzahl an Instanzen ist auf die statische Nutzung des Arbeitsspeichers zurückzuführen. In Rahmen der Masterarbeit

wurde die maximale Anzahl mit 20 festgelegt. Der Benutzer kann diese jedoch ändern. Die Aufgabe der ChannelMessage ist es, die Regeln für die Manipulation zu speichern. Jede Instanz enthält dabei die Regeln für den FuzzingReader, den er repräsentiert. Pro ChannelMessage können 20 Regeln gespeichert werden. Die Manipulation der Datenobjekte findet in der ExecuteRules Klasse statt. Das Erstellen der Regeln und deren Anwendung auf die Datenobjekte wird in Kapitel 5 behandelt.

4.5 Master Software

Die Master Software bildet das Gegenstück zur Slave Software und wird auf einem Destop-PC ausgeführt. Ihre Aufgabe ist es, die Regeln für die Manipulation der Datenobjekte zu erstellen. Diese sendet sie anschließend zu der Slave Software, mit der sie über TCP/IP verbunden ist. Die zweite Aufgabe der Master Software ist das Abspeichern der Datenobjekte in einer SQL-Datenbank.

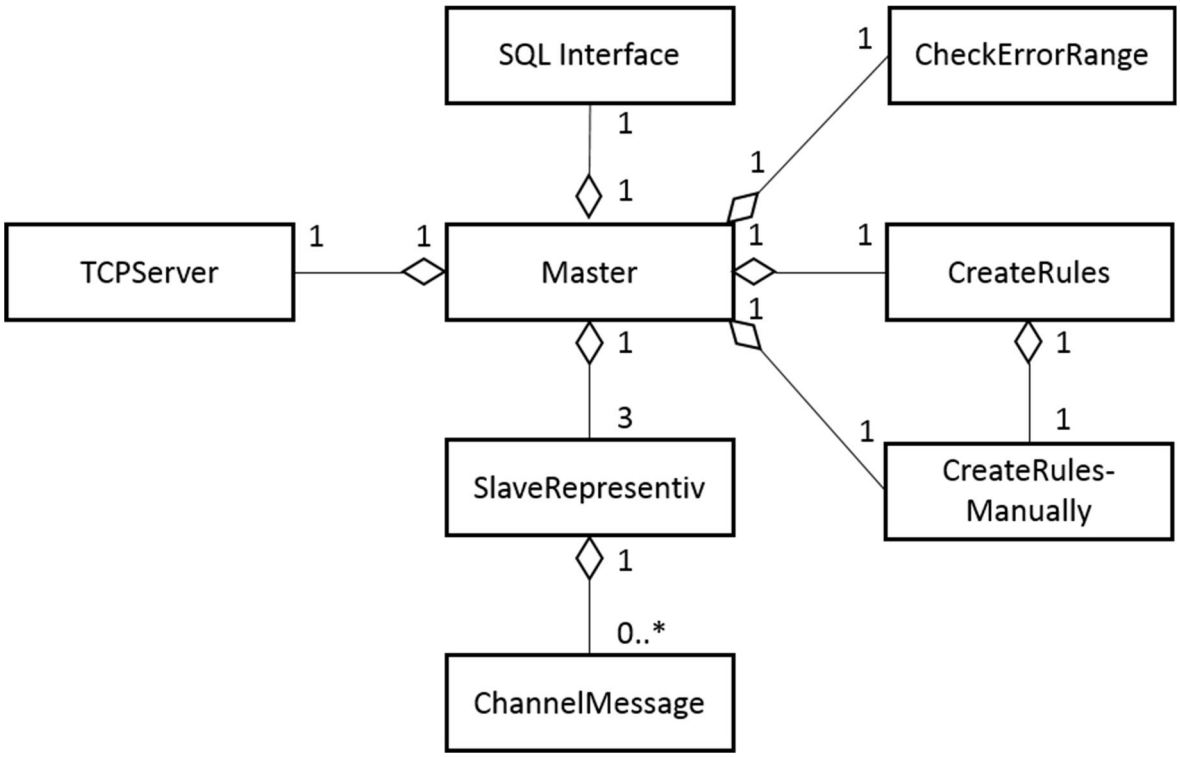

Abbildung 4.6: Klassendiagramm der Master Software ohne Attribute und Methoden
Quelle: Eigene Zeichnung

Den Kern der Master Software bildet die Master Klasse, welche deren Funktionalität steuert. Wie in Abbildung 4.6 zu erkennen ist, sind alle anderen Klassen Bestandteil der Master Klasse. Der Datenaustausch zur Slave Software erfolgt mittels der TCPServer Klasse. Eine

genauere Beschreibung TCP/IP Verbindung erfolgt in Kapitel 4.6. Da jeder HPN eine Slave Software besitzt, kann die Master Software bis zu drei Verbindungen gleichzeitig besitzen. Jede der verbunden Slave Softwares wird durch eine Instanz der SlaveRepresentiv Klasse repräsentiert. In ihr sind die von der Master Software benötigten Information zur Slave Software enthalten. Sie enthält darüber hinaus mehrere Instanzen der ChannelMessage Klasse. Da eine dynamische Nutzung des Arbeitsspeichers erlaubt ist, muss eine feste Anzahl nicht definiert werden. Die ChannelMessage Klasse ist identisch mit der ChannelMessage Klasse der Slave Software.

Das Erstellen der Regeln erfolgt in den Klassen CreateRules, CreateRulesManually und CheckErrorRange. Es erfolgt auf Basis der Datenobjekte, welche die Master Software von der Slave Software erhalten hat. Zusätzlich können einzelne Regeln manuell definiert werden. In Kapitel 5 wird dies ausführlich behandelt. Das Speichern der Datenobjekte erfolgt in einer SQL Datenbank auf dem Desktop Rechner. Der Zugriff auf die Datenbank erfolgt über die SQL-Interface Klasse der Master Software. Kapitel 4.7 befasst sich ausführlich mit der Datenbank und dem Interface.

4.6 TCP/IP-Verbindung zwischen Master- und Slave Software

Die Master-Slave-Software besteht aus zwei Komponenten, welche sich auf verschiedener Hardware befinden. Die Slave Software auf den HPNs und die Master Software auf dem Desktop PC. Um ein Testen mittels Fuzzing zu ermöglichen, müssen die beiden Komponenten miteinander kommunizieren. Die Verbindung der Hardwarekomponenten erfolgt über Ethernet. Als Übertragungsprotokoll wurde TCP/IP gewählt. Bei TCP handelt es sich um ein verbindungorientiertes Protokoll für den Datenaustausch zweier Netzwerkteilnehmer [20]. Es wurde für die Master-Slave-Software ausgewählt, weil es eine fehlerfreie Übertragung der Daten erlaubt. Die Datenübertragung erfolgt bei TCP in Form eine Byte-Streams. Deshalb wurde ein eigenes Übertragungsprotokoll entwickelt, um Datenstrukturen seriell zu übertragen.

4.6.1 Verbindungsaufbau zwischen Server und Client

Bei der TCP-Verbindung der Master-Slave-Software sind die drei Slave Softwares die Clients, welche sich mit dem Server der Master Software verbinden. Die Verbindung erfolgt mittels Sockets des Linux Betriebssysteme. Für die Verwaltung mehrere Clientverbindungen verwendet der Server die „select“ Funktion. Ist der Verbindungsaufbau zwischen Server und Client erfolgt, bleibt die Verbindung während des gesamten Testvorgangs dauerhaft

bestehen. Um ein Blockieren der Anwendung durch die Sockets zu verhindern, werden Timeouts verwendet.

4.6.2 Übertragungsprotokoll für die serielle Datenübertragung mittels Byte-Stream

Nach dem Aufbau einer TCP Verbindung, wird diese zum gegenseitigen Datenaustausch zwischen Master- und Slave Software verwendet. Die Datenübertragung mittels TCP erfolgt über einen Byte-Stream. Aus diesem Grund ist ein Protokoll notwendig, mit dem die Datenobjekte für die Übertragung serialisiert und deserialisiert werden können. Dies erfolgt über das Protokoll aus Abbildung 4.7.

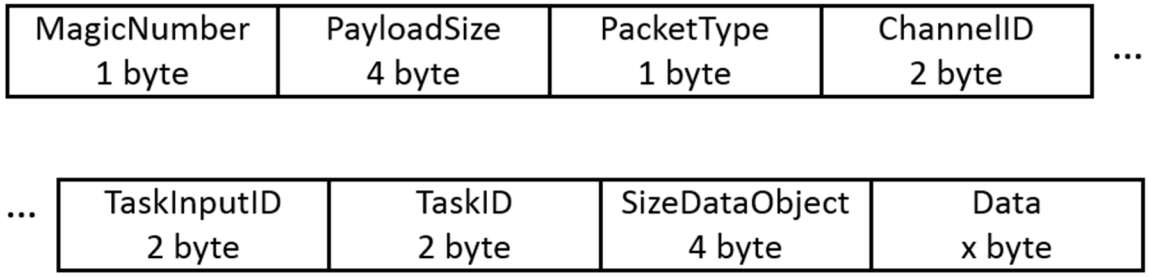

Abbildung 4.7: Protokoll für die serielle Datenübertragung mittels Byte-Stream
Quelle: Eigene Zeichnung

Das erste Byte des Protokolls bildet die MagicNumber mit dem Bytewert von 42. Sie kennzeichnet den Anfang des übertragenen Datenobjektes. Anschließend folgt die PayloadSize. Sie gibt an, wieviel Byte des Byte-Stream zu einem Datenobjekt gehören. Dies ist notwendig, um verschiedene seriell übertragen Datenobjekte voneinander zu trennen. Die PayloadSize wird darüber hinaus verwendet, um getrennt übertragene Teile des Datenobjektes wieder zusammenzuführen. Aus der PacketType kann entnommen werden, welche Art von Daten in dem Datenobjekt enthalten sind. Das Byte wird darüber hinaus Verwendet um Statusmeldungen zu übertragen. Eine Auflistung der Bytewerte und deren Bedeutung finde sich in Tabelle A.2 aus Anhang A1.

Die nächsten sechs Byte der ChannelID, TaskInputID und TaskID enthalten Inforationen zum FuzzingReader, von dem das übertragene Datenobjekt stammt. Wird kein Datenobjekt übertragen, enthalten diese 6 Byte keine Informationen. Mit SizeDataObject enthält das Protokoll eine weitere Größenangabe. Sie wird verwendet, wenn die Slave Software ein Datenobjekt samt der darauf angewendeten Regel überträgt. SizeDataObject enthält dann die Größe des Datenobjektes und erlaubt so die Trennung zwischen Datenobjekt und Regel. Den Schluss bilden die zu übertragen Daten in Data. Deren Größe ist je nach Art der übertragenen Daten unterschiedlich.

4.7 SQL-Datenbank

Eine Rahmenbedingung aus Kapitel 4.7 ist das Abspeichern der, beim Testen durchgeführten Manipulationen. Um die Rahmenbedingung einzuhalten, speichert die Master-Slave-Software alle Datenobjekte, welche sie vom Tasking Framework erhalten hat. Die Speicherung soll dauerhaft erfolgen, sodass die Daten auch nach der Durchführung des Testes zur Verfügung stehen. Aus diesem Grunde, wird eine SQL-Datenbank für die Speicherung verwendet.

Der Vorteile einer SQL-Datenbank ist, dass sie die Structured Query Language(SQL) als Datenbanksprache verwendet. Dies erlaubt den Zugriff auf die Datenbank mittels SQL-Befehlen. Dadurch können verschiede Anwendungen auf die Daten zugreifen. Ein Beispiel für eine weitere Anwendung ist die Visualisierung von Fehlern, die nicht in dieser Masterarbeit behandelt wird. Für die Erstellung der Datenbank wird die SQLite Programmbibliothek verwendet [21]. SQLite erzeugt die Datenbank in Form einer einfachen Datei mit der Endung „.db“. Die Anbindung an die Master-Software erfolgt mittels der SQLite Programmierschnittstelle für C++. Der Programmcode befindet sich dabei in der SQL-Interface Klasse.

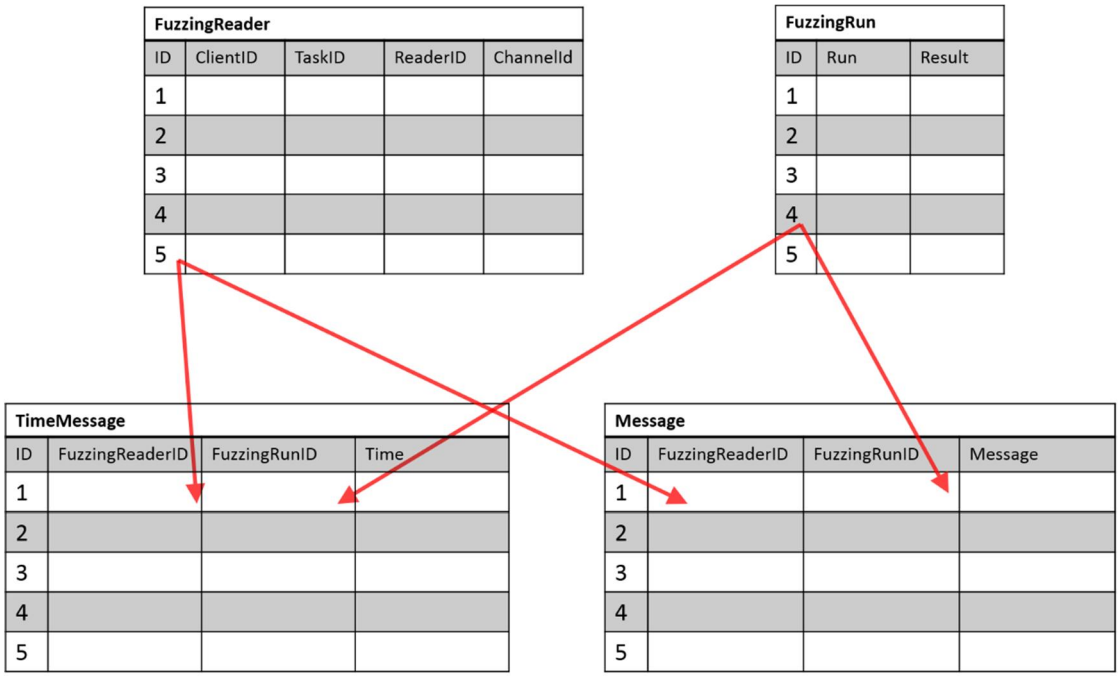

Abbildung 4.8: Aufbau der SQL Datenbank für die Speicherung der manipulierten Datenobjekte
Quelle: Eigene Zeichnung

Abbildung 4.8 zeigt den Aufbau der SQL Datenbank, welche von der Master-Software verwendet wird. Sie enthält die vier Tabellen FuzzingReader, Message, TimeMessage und

FuzzingRun. Die Speicherung der Datenobjekte erfolgt in der Tabelle Message. Dabei werden die von der Slave Software empfangenen Datenobjekte als Binary Large Object(BLOB) gespeichert. Neben den empfangenen Datenobjekten enthält die Tabelle Message Zuweisungen zu jeweils einer Zeile der beiden Tabellen FuzzingReader und FuzzingRun. In diesen sind Informationen zu den Datenobjekten gespeichert.

Neben den Datenobjekten, welche in der Tabelle Message gespeichert werden, erhält die Master Software Zeitmessungen von der Slave Software. Diese sind Teil der Detektierung von Fehlern aus Kapitel 5.3. Die Tabelle TimeMessage speichert die Zeitmessung als Integer Wert ab. Analog zur Tabelle Message werden die Zuweisungen zu jeweils einer Zeile der Tabellen FuzzingReader und FuzzingRun gespeichert.

In der Tabelle FuzzingReader sind alle Informationen zu den FuzzingReader des Tasking Frameworks aus Kapitel 4.3 gespeichert. In der Tabelle sind alle FuzzingReader aufgelistet, welche mit der Slave Software auf einem der drei HPNs verbunden sind und dieser ein Datenobjekt übermittelt haben. Durch die Zuweisung einer Zeile zu einem Datenobjekt der Tabelle Message kann zurückverfolgt werden, woher dieses kommt.

Die letzte Tabelle FuzzingRun enthält die Ergebnisse der einzelnen Testdurchläufe der Master-Slave-Software. Diese werden in Kapitel 4.2.2 erläutert. Ist der Testdurchlauf noch nicht abgeschlossen, wird der aktuelle Status abgespeichert. Jedem Datenobjekt aus der Tabelle Message wird der Testdurchlauf zugewiesen, in dem dieses gespeichert wurde.

Damit der Nutzer nach den Testdurchläufen auf die Daten in der Datenbank zugreifen kann, wird die Open-Source Software SQLite Studio [22] verwendet. Die Anwendung erlaubt das Exportieren der Daten als csv-Datei. Da die empfangenen Datenobjekte in serialisierte Form und als BLOB in der Tabelle Message vorliegen, kann SQLite Studio diese nicht auslesen. Deshalb besitz das Fuzzing Framework mit der Klasse OutputResults eine Parser, der die Daten als csv-Datei ausgibt.

5 Manipulation der Datenobjekte durch das Fuzzing Frameworks

Die Manipulation von Datenobjekten des Tasking Frameworks des DLRs soll nach, im Fuzzing Framework definierten Regeln erfolgen. Sie stellen Anweisungen an die Slave Software dar, wie diese die einzelnen Datenobjekte zu manipulieren hat. Nach der Manipulation schickt die Slave Software die angewandten Regeln zusammen mit der Kopie des unmanipulierten Datenobjektes zurück zur Master Software. Auf diese Weise wird die durchgeführte Manipulation dokumentiert. Zusätzlich übernehmen drei Funktionen die Aufgabe der Fehlerüberwachung.

In diesem Kapitel werden die Regeln des Fuzzing Frameworks vorgestellt. Im ersten Teil wird die Erstellung der Regeln durch die Master Software behandelt. Dazu zählt die Auflistung aller Regeln sowie deren Aufbau. Der zweite Teil stellt die Ausführung der Regeln durch die Slave Software vor. Am Ende werden die Funktionen für die Fehlerüberwachung behandelt.

Nach der Klassifizierung verschiedener Fuzzing Frameworks in Kapitel 2.4, entspricht der Ansatz der Manipulation dem regelbasierten Fuzzing. Für die Manipulation werden korrekte Eingabedaten verwendet.

5.1 Erstellen der Regeln zur Manipulation

Das Erstellen der Regeln erfolgt im Fuzzing Framework durch die Master Software. Wie in Kapitel 4.2.2 erläutert, erhält diese beim Profiling Run eine Kopie von jedem Datenobjekt, welches bei dem Programmdurchlauf vom TaskChannel an die Tasks weitergeleitet wird. Nach Abschluss des Fuzzing Runs entnimmt die Master Software die erhaltenen Datenobjekte aus der SQL-Datenbank, um für diese die Regeln zu erstellen.

Die entnommenen Datenobjekte liegen dabei in Form eines Byte-Streams vor. Die Umwandlung eines Datenobjektes in einen Byte-Stream wird in Kapitel 4.3.2 beschrieben. Jeder Byte-Stream enthält den Wert und den Typ des Datenobjektes. Aus diesen Informationen wählt die Master Software die Regeln aus, welche auf dieses Datenobjekt angewendet werden sollen. Nachdem die Master Software die Regeln erstellt hat, werden diese an die Slave Softwares geschickt.

5.1.1 Aufbau der Regeln

Damit die Regeln von der Master Software an die Slave Software übertragen werden können, müssen diese die Form eines Byte-Stream aufweisen. Aus diesem Grund sind die Regeln als Byte-Stream aufgebaut. Alle Regeln nutzen dafür das in Abbildung 5.1 dargestellte Protokoll. Die ersten vier Bytes enthalten die Größe des Byte-Stream der einzelnen Regel. Darauf folgt die Bytegröße ClientID. Diese gibt an, für welche Slave Software die Regel bestimmt ist. Über die FuzzingReaderID erfolgt die Zuordnung der Regel zu dem entsprechenden FuzzingReader. Die RuleID gibt an, um welche Regel es sich handelt. Eine Auflistung der einzelnen Regeln mit ihren RuleIDs findet sich in Tabelle A.3 des Anhanges.

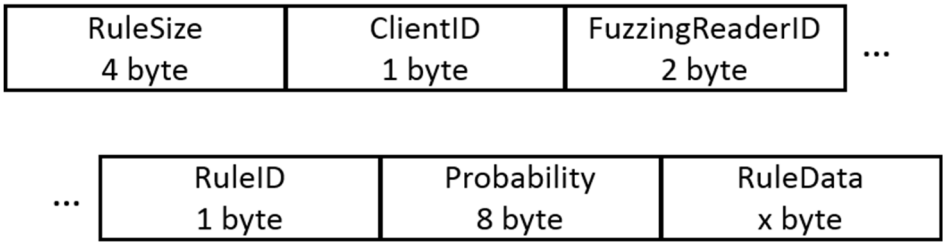

Abbildung 5.1: Protokoll für die Übertragung von Regeln
Quelle: Eigene Zeichnung

In Probability wird die Wahrscheinlichkeit angegeben, dass diese Regel ausgeführt wird. Die acht Byte enthalten dabei den Wert eines serialisierten Datenobjekts vom Typ „double“. Eine genauere Beschreibung der Wahrscheinlichkeit und wie diese die Ausführung der Regeln beeinflusst, findet sich in Kapitel 5.1.4. Am Ende des Byte-Streams befinden sich die Daten zu den einzelnen Regeln. Dabei handelt es sich um Flags und Parameter, welche beeinflussen, wie die Regel die Manipulation vornimmt.

5.1.2 Regeln unabhängig vom Datenobjekt

Es existieren zwei Regeln für die Manipulation von Datenobjekten, welche unabhängig von deren Wert und Typ angewendet werden. Sie sind damit auf alle Datenobjekte des Tasking Frameworks anwendbar. Die beiden Regeln sind in Tabelle 5.1 aufgelistet.

Tabelle 5.1: Auflistung der Regeln unabhängig vom Wert und Typ des Datenobjektes

Regel	Erläuterung
dropDataobject	Diese Regel gibt der Slave Software die Anweisung, das komplette Datenobjekt aus dem TaskChannel zu löschen. Stattdessen wird das nächste Datenobjekt an den Task weitergereicht. Ist

	kein weiteres Datenobjekt im TaskChannel enthalten, wird der NULL pointer als Referenz weitergereicht. Das Ziel der Regel ist, zu testen, wie sich das Tasking Framework verhält, wenn einzelne Datenobjekte verloren gehen.
changeRandomBit	Die Regel „changeRandomBit“ ändert ein zufälliges Bit des Datenobjektes. Der Nutzer des Fuzzing Frameworks kann dabei festlegen, wie viele Bits eines Datenobjektes geändert werden. Das Ziel ist eine möglichst zufällige Änderung des Datenobjektes. Die Regel ist darüber hinaus geeignet für die Manipulation großer serialisierte Datenblöcke. Dies können zum Beispiel Bilder sein. Nach Ausführung der Regel, wird diese um die Information erweitert, welche Bits geändert wurden.

5.1.3 Regeln abhängig vom Typ und Wert des Datenobjektes

Zusätzlich existieren Regeln, welche von dem Typ und dem Wert des Datenobjektes abhängen. Diese ändern den Wert des Datenobjektes nicht zufällig durch das Ändern von Bits. Stattdessen wird der Wert gezielt geändert. Ein Beispiel dafür ist die Änderung des Vorzeichens oder das Löschen sämtlicher Nachkommastellen. Diese Regeln sind jedoch nicht auf alle Datentypen anwendbar. In Tabelle A.4 aus Anhang A2 befindet sich die Zuweisung, welche Regeln auf einen bestimmten Datentyp angewendet werden können.

Bei den Datentypen handelt es sich um einfache Datentypen wie int oder float. Ist das zu manipulierende Datenobjekt eine komplexe Struktur aus mehreren einfachen Datenobjekten, erstellt die Master Software für jedes der einfachen Datenobjekte Regeln. Die Regeln selbst sind in Tabelle 5.2 aufgelistet.

Tabelle 5.2: Auflistung der Regeln abhängig vom Typ und Wert des Datenobjektes

Regel	Erläuterung
changeValueZero	Durch diese Regel wird der Wert des Datenobjektes auf den Wert Null geändert. Durch diese Regel wird getestet, wie der Programmcode in den Tasks auf Nullwerte reagiert.
changeValueAbs	Mit „changeValueAbs“ wird der Wert des Datenobjektes durch einen neuen Wert aus einem angegebenen Wertebereich ersetzt. Die Regel besitzt zwei Parameter, mit denen der Nutzer den minimalen und maximalen Wert des Wertebereiches angeben kann. Standardmäßig sind

	dies der minimale und maximale Wert des Datentyps. Aus diesem Wertebereich wird bei der Manipulation ein zufällig neuer Wert gewählt. Dieser ersetzt den alten Wert. Durch die Begrenzung des Wertebereiches, erlaubt diese Regel eine gezielte Manipulation des Datenobjektes. Nach der Durchführung der Manipulation wird der neue Datenwert der angewendeten Regel hinzugefügt. Auf diese Weise wird die durchgeführte Manipulation dokumentiert.
changeValueRel	Diese Regel ändert den Wert des Datenobjekts abhängig vom aktuellen Wert. Dafür gibt der Nutzer einen minimalen und maximalen Prozentsatz an, um den der Wert geändert werden soll. Die Manipulation erfolgt durch einen zufällig gewählten Prozentsatz dieses Wertes. Diese Regel wird nur dann erstellt, wenn der Wert des Datenobjektes ungleich Null ist. Als minimalen und maximalen Prozentwert verwendet die Regel standardmäßig -100 und 100. Der Prozentsatz vom Wert des Datenobjektes wird zu diesem hinzugefügt und ergibt so den neuen Wert. Zur Dokumentation wird der neue Wert zur Regel hinzugefügt. Der Vorteil dieser Regel ist, dass sie durch die relative Änderung den aktuellen Wert des Datenobjektes berücksichtigt.
changeSign	Diese Regel ändert das Vorzeichen vom Wert des Datenobjektes. Sie wird aus diesem Grund nur für Datentypen erstellt, die ein änderbares Vorzeichen haben.
changeFractionalPart	Die Regel „changeFractionalPart“ ändert die Nachkommastelle von Datenobjekten. Sie wird deshalb nur auf die Datentypen angewendet, die Gleitkommazahlen darstellen können. Der Nutzer kann durch ein Flag drei Optionen wählen. Die Erste entfernt sämtliche Nachkommastellen ohne zu runden. Bei der Zweiten wird eine zufällig gene-

	rierte Nachkommastelle hinzugefügt, wenn keine vorhanden ist. Die letzte Option entfernt vorhandene Nachkommastellen oder fügt Zufällige hinzu, wenn keine vorhanden sind. Standartmäßig wird die letzte Option angewendet.

5.1.4 Einstellung der Regeln durch den Nutzer

Die Master Software erstellt automatisch für jedes Datenobjekt sämtliche anwendbaren Regeln. Erlauben der Typ und der Wert des Datenobjektes die Anwendung einer Regel, wird diese erstellt. Bei der Erstellung verwendet die Master Software für alle Datenobjekte die gleichen Flags und Parameter. Der Nutzer definiert diese einmalig mittels set-Methoden in der Master Klasse der Masters Software. Neben den Parametern und Flags kann der Nutzer für jede Regel die Wahrscheinlichkeit festlegen, dass diese ausgeführt wird. Des Weiteren ist es möglich, einzelne Regeln zu deaktivieren, sodass diese von der Master Software nicht erstellt werden. Ändert der Nutzer nicht die Einstellung, haben alle Regel eine Wahrscheinlichkeit von 0.9 und sind aktiv. Die Erstellung erfolgt in der CreateRules Klasse. Das automatische Erstellen der Regeln entspricht dem Qualitätsziel des Automatisierungsgerades aus Kapitel 4.1.

Zusätzlich besitzt der Nutzer des Fuzzing Frameworks die Möglichkeit, die Regel „changeValueAbs“ und „changeValueRel“ manuell für einzelne Datenobjekte zu erstellen. Dazu erstellt er eine Instanz der Klasse CreateRulesManually. Dieser fügt er mit der entsprechenden add-Methode die einzelnen Regeln hinzu. Anschließend wird die Referenz von CreateRulesManually an die Master-Klasse übergeben. Die manuell erstellten Regeln werden dann zu den automatisch erstellten Regeln der Klasse CreateRules hinzugefügt. Wurde eine Regel für ein Datenobjekt manuell erstellt, wird dieselbe Regel für dieses Datenobjekt nicht mehr automatisch erstellt.

5.2 Ausführung der Regeln zur Manipulation der Datenobjekte

Nach der Erstellung der Regeln durch die Master Software, werden diese an die Slave Software geschickt. Hier werden die Regeln auf die Instanzen der Klasse ChannelMessage verteilt und gespeichert. Anschließend besitzt jede Instanz die Regeln für den FuzzingReader, den sie repräsentieren.

5.2.1 Ausführung der Regel

Wird die Slave Software von einem FuzzingReader des Tasking Frameworks aufgerufen, erhält sie das zu manipulierende Datenobjekt. Darüber hinaus werden Information zur Identifizierung des FuzzingReaders übermittelt. Anhand dieser Information wählt die Slave Software die entsprechende Instanz von ChannelMessage und entnimmt ihr die Regeln. Diese werden zusammen mit dem Datenobjekt an die Klasse ExecuteRules übergeben. Hier werden im ersten Schritt die Regeln ausgewählt, welche für die Manipulation verwendet werden.

Die Klasse ExecuteRules besitzt für jede Regel eine Methode, welche das Datenobjekt manipuliert. Für jede ausgewählte Regel wird die entsprechende Methode aufgerufen und das Datenobjekt zusammen mit den in der Regel gespeicherten Parametern übergeben. Nach der Ausführung aller Regeln, gibt ExecuteRules das manipulierte Datenobjekt zusammen mit den ausgeführten Regeln zurück.

Die ausgeführten Regeln werden anschließend zusammen mit einer Kopie des noch unmanipulierten Datenobjektes an die Master Software geschickt. Auf diese Weise wird die durchgeführte Manipulation dokumentiert. Anschließend gibt die Slave Software das manipulierte Datenobjekt an den FuzzingReader zurück. Da die Benachrichtigung der Master Software vor der Rückgabe des manipulierten Datenobjektes erfolgt, ist die Dokumentation sichergestellt, falls es aufgrund des manipulierten Datenobjektes zu einem Absturz des Tasking Frameworks kommt.

5.2.2 Auswahl einer Regel

Welche Regeln für die Manipulation verwendet werden sollen, entscheidet die ExecuteRules Klasse anhand von Wahrscheinlichkeiten. Jede Regel besitzt nach Kapitel 5.1.1 einen Wert vom Typ „double“, der die Wahrscheinlichkeit repräsentiert. Dieser wird als „probability“ bezeichnet. Dabei enthält der Wert von „probability“ zwei Informationen. Diese sind das Vorzeichen und der Betrag des Wertes. Der Betrag hat einen Wert zwischen 0 und 1.

Die ExecuteRules Klasse besitzt zwei Vorgehensweisen, wie sie mit den Wahrscheinlichkeiten der Regeln umgeht. Das Vorzeichen entscheidet darüber, welche Vorgehensweise gewählt wird. Dazu werden die Regeln je nach Vorzeichen in zwei Gruppen aufgeteilt.

Die erste Gruppe stellen die Regeln mit den positiven Vorzeichen dar. Aus diesen wird nur eine Regel für die Manipulation verwendet. Im ersten Schritt wird zufällig eine Regel ausgewählt. Alle Regeln besitzen die gleiche Wahrscheinlichkeit ausgewählt zu werden. Im zweiten Schritt, wird diese mit der Wahrscheinlichkeit des Betrages von „probability“ aus-

geführt. Wurde die Regel nicht ausgeführt, wird erneut eine Regel ausgewählt und der Vorgang wiederholt. Dies geschieht solange, bis eine Regel ausgeführt wird. Nach der Ausführung werden keine weiteren Regeln ausgewählt.

Regeln mit einem negativen Vorzeichen bilden die zweite Gruppe. Aus dieser Gruppe wird jede Regel ein einziges Mal mit der Wahrscheinlichkeit des Betrages von „probability“ ausgeführt. Damit ist die Ausführung von mehreren Regeln für die Manipulation des Datenobjektes möglich. Durch die negative Wahrscheinlichkeit kann darüber hinaus sichergestellt werden, dass eine Regel immer ausgeführt wird. Dies ist der Fall, wenn der Wert von „probability“ den Wert -1 hat. Dies ist für die Regeln erforderlich, die für die Fehlerüberwachung verwendet werden.

5.3 Funktionen zum detektieren aufgetretener Fehler

Eine weitere Funktionalität des Fuzzing Frameworks ist das Detektieren von Fehlern, welche durch die manipulierten Datenobjekte erzeugt werden. Das Fuzzing Framework detektiert dazu drei Arten von Fehlern. Als erstes können die Werte der Datenobjekte dahingehend überprüft werden, ob sie einen bestimmten Wertebereich einhalten. Als zweites wird das zu testende Programm mit dem Debugger GDB [23] ausgeführt, um Programmabstürze zu dokumentieren. Als letzte Funktion erlaubt das Fuzzing Framework, die Zeit für die Ausführung eines Tasks zu messen.

5.3.1 Überprüfung des Wertebereiches

Die Überprüfung des Wertebereiches bestimmter Datenobjekte erfolgt mittels der speziellen Regel „checkErrorRange“. Diese überprüft, ob der Wert des Datenobjektes im angegeben Wertebereich liegt. Als Parameter definiert der Nutzer den minimalen und maximalen Wert des Wertebereiches. Darüber hinaus kann er festlegen, ob der Task bei einer Verletzung des Wertebereiches weiterhin gestartet werden soll. Für ein einzelnes Datenobjekt können mehrere Regeln erstellt werden.

Die Erstellung der Regel erfolgt ausschließlich manuell durch den Benutzer. Dieser fügt die Regeln mittels add-Methode einem Objekt vom Typ CheckErrorRange hinzu. Dieses Objekt wird anschließend an die Master Klasse der Master Software übergeben. Diese leitet die Regeln zur Überprüfung zusammen mit den anderen Regeln an die Slave Software weiter. Die Ausführung der Regel erfolgt wie die anderen Regeln in der ExecuteRules Klasse.

5.3.2 Dokumentation von Programmabstürzen

Die zweite Art von Fehlern stellen Programmabstürze der zu testen Software dar. Um diese zu detektieren, wird das zu testende Programm mit dem Debugger GDB ausgeführt. Dieser liefert bei einem Programmabsturz Information über die Art und den Ort des Fehlers. Zur Auswertung werden diese Informationen in einer log-file gespeichert.

5.3.3 Zeitmessung in den Tasks

Eine weiter Funktion zum Detektieren von Fehlern stellt die Messung von Zeiten innerhalb der Tasks dar. Das Ziel dieser Zeitmessungen ist es, den Einfluss der manipulierten Datenobjekte auf die Zeit zu messen, die ein Task führ die Verarbeitung der Daten benötigt. Dies erlaubt die Detektierung von Fehlern, die weder zu einem Programmabsturz oder einer Verletzung des Wertebereiches führen, jedoch die Bearbeitungszeit eines Task stark erhöhen. Die Überwachung der Bearbeitungszeit ist von besonderer Bedeutung, wenn die Bearbeitungszeit der Tasks vorgegeben Grenzwerte nicht überschreiten darf.

Die Messungen innerhalb der Tasks erfolgen über zwei Methoden der FuzzingReader aus Kapitel 4.3.1. Gemessen wird die Zeit in Millisekunden, die zwischen dem Aufruf der beiden Methoden vergangen ist. Die gemessene Zeit wird anschließend automatisch zur Master Software gesendet und in der SQL-Datenbank gespeichert.

6 Anwendung des Fuzzing Framework

Das Ziel des Fuzzing Framework ist das Testen von Anwendungen, die mit dem Tasking Framework realisiert wurden. Dafür wird die Slave Software des Fuzzing Frameworks mit den zu testenden Anwendungen verbunden. Anschließend wird die Anwendung wiederholt ausgeführt und mittels Fuzzing getestet. Dabei entstehen Daten, welche die Slave Software an die Master Software schickt. Diese beinhaltenen Informationen zu Programmabstürzen, Überschreitung von angegeben Wertebereichen und die Bearbeitungszeit der Tasks. Aus diesen kann geschlussfolgert werden, ob die zu testende Software Fehler aufweist.

In diesem Kapitel wird das Fuzzing Framework zum Testen von drei Beispielanwendungen verwendet. Für jede der drei Anwendungen werden unterschiedliche Regeln zur Manipulation der Daten verwendet. Diese sollen zeigen, wie das Fuzzing Framework zum Testen verwendet werden kann und welche Fehler erkannt werden können. Im ersten Teil werden einfache Beispiele mittels Fuzzing getestet. Dabei werden unter anderem Wertebereichsverletzungen erzeugt. Der zweite Teil handelt vom Testen einer Anwendung, die auf drei HPNs ausgeführt wird. Getestet wird hier die Bearbeitungszeit. Im letzten Teil wird eine Anwendung zur Bilderkennung getestet. Das Testen erfolgt durch das zufällige Ändern einzelner Bits und dem Löschen vollständiger Datenobjekte.

6.1 Beispiele zur Erkennung von Laufzeitfehlern durch Wertebereichsverletzungen

Die erste Anwendung des Fuzzing Frameworks erfolgt auf eine Beispielanwendung, die für diesen Zweck erstellt wurde. Das Ziel ist es, den Wert eines Datenobjektes vom Typ int32_t zu manipulieren. Dadurch sollen Laufzeitfehler erzeugt werden, die zu einer Verletzung eines überwachten Wertebereiches führen. Zudem wird ein Programmabsturz durch die Division durch Null erzeugt. Anschließend werden die bei den Fuzzing-Durchläufen erzeugten Daten präsentiert und ausgewertet. Der Quellcode der Beispielanwendung befindet sich im digitalen Anhang B2.1.

Wie Abbildung 6.1 zu entnehmen ist, besteht die Beispielanwendung aus zwei Tasks mit jeweils einem TaskInput und TaskChannel. Diese sind in einer Reihe verbunden, sodass der erste Task Daten in den zweiten TaskChannel schreibt. Somit erzeugt er die Eingabedaten des zweiten Tasks. Die Reihenschaltung der beiden Tasks erlaubt es, die Datenobjekte aus dem ersten TaskChannel zu manipulieren und dadurch Fehler im ersten Task zu

erzeugen. Führen die Fehler nicht zu einem Programmabsturz, schreibt dieser anschließend ein Datenobjekt in den zweiten TaskChannel. Die Datenobjekte des zweiten TaskChannels können dann überwacht werden, um Wertebereichsverletzungen festzustellen. Der zweite Task entnimmt dazu die Datenobjekte, bearbeitet diese jedoch nicht.

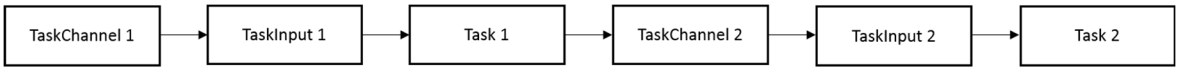

Abbildung 6.1: Datenflussdiagramm der Beispielanwendung
Quelle: Eigene Zeichnung

Wie in Kapitel 4.3.1 beschrieben, wurde eine Instanz der Slave Software mit beiden Tasks der Beispielanwendung verbunden. Die Beispielanwendung wurde während der Versuchsdurchführung auf einem einzelnen HPN des ScOSA-Bords ausgeführt.

6.1.1 Verletzung des Wertebereiches durch Overflow

Bei der ersten Versuchsdurchführung soll durch die Manipulation des Datenobjektes von Typ int32_t ein Überlauf eines internen Datenobjektes des ersten Tasks erzeugt werden. Dieser übergelaufene Wert wird anschließend in den zweiten TaskChannel geschrieben und überprüft. Tabelle 6.1 zeigt die Ergebnisse der ersten Versuchsdurchführung aus Anhang B2.2. In der Tabelle besitzt jede Zeile die Daten eines Datenobjektes. An der RunID ist zu erkennen, dass neben dem „Profiling Run“ zehn „Fuzzing Run“-Durchläufe durchgeführt wurden. Die Spalten TaskID, ChannelID und ReaderID enthalten Informationen, von welchem Task und TaskChannel die Daten stammen. Die Daten vom ersten Task besitzen für alle IDs den Wert 1 und die des zweiten Tasks den Wert 4.

Werden pro „Fuzzing Run“ mehrere Datenobjekte mit denselben IDs von der Slave Software zur Master Software gesendet, werden diese durch unterschiedliche Werte des Zählers von Message ID unterschieden. DataID erlaubt, wie in Kapitel 4.3.2 erläutert, die Unterscheidung mehrerer Datenwerte eines Datenobjektes.

OldValue enthält den ursprünglichen Datenwert des Datenobjektes bevor dieser manipuliert wird. An der RuleID ist zu erkennen, welche Regel zur Manipulation des Datenwertes angewendet wurde. Die Zuweisung zwischen Regel und RuleID befindet sich in Tabelle A.3 des Anhangs A1. Der neue, manipulierte Wert ist in NewValue dargestellt.

Wie Tabelle 6.1 zu entnehmen ist, beträgt der Datenwert des ersten TaskChannels in allen Durchläufen 30000. Dieser Wert wird anschließend mit der Regel „changeValueRel“ manipuliert. Die Regel wurde für alle Durchläufe so parametriert, dass sie den Wert um 0 bis 200 Prozent des ursprünglichen Datenwertes erhöht. Der Wert der Wahrscheinlichkeit

wurde mit -1.0 festgelegt. Damit wird die Regel immer ausgeführt. Das Ergebnis der Manipulation kann der Spalte NewValue entnommen werden.

Tabelle 6.1: Ergebnisse zur Verletzung der Wertebereiche durch Overflow

RunID	MessageID	TaskID	ChannelID	ReaderID	DataID	OldValue	RuleId	NewValue
0	0	1	1	1	1	30000		
1	0	1	1	1	1	30000	5	52895
2	0	1	1	1	1	30000	5	30116
3	0	1	1	1	1	30000	5	71280
4	0	1	1	1	1	30000	5	64200
5	0	1	1	1	1	30000	5	68611
6	0	1	1	1	1	30000	5	44532
7	0	1	1	1	1	30000	5	88637
8	0	1	1	1	1	30000	5	42370
9	0	1	1	1	1	30000	5	67911
10	0	1	1	1	1	30000	5	66738
RunID	MessageID	TaskID	ChannelID	ReaderID	DataID	OldValue	RuleId	NewValue
0	0	4	4	4	1	60000		
1	0	4	4	4	1	17359	8	1
2	0	4	4	4	1	60116	8	0
3	0	4	4	4	1	35744	8	0
4	0	4	4	4	1	28664	8	1
5	0	4	4	4	1	33075	8	0
6	0	4	4	4	1	8996	8	1
7	0	4	4	4	1	53101	8	0
8	0	4	4	4	1	6834	8	1
9	0	4	4	4	1	32375	8	0
10	0	4	4	4	1	31202	8	0

Der erste Task hat die Aufgabe, das manipulierte Datenobjekt aus dem ersten TaskChannel zu entnehmen. Im Task wird der manipulierte Datenwert zu einem Datenobjekt vom Typ uint16_t addiert, welches mit seinem ungefähren Mittelwert von 30000 initialisiert wurde. Der Wert des Datenobjektes wird anschließend in den zweiten TaskChannel gespeichert. Der zweite Task entnimmt anschließend das Datenobjekt. Dabei wird das Datenobjekt erneut durch eine Regel manipuliert. An der RuleID von 8 ist zu erkennen, dass es die Regel „checkErrorRange“ ist. Diese überprüft, ob sich der Wert des Datenobjektes im Wertebereich zwischen 30000 und 65535 befindet. Ist dies der Fall, gibt sie den Wert 0 zurück, welcher in der Spalte von NewValue gespeichert wird. Liegt der Wert außerhalb des Wertebereiches, ist der Wert 1.

Da das Datenobjekt vom Typ uint16_t des ersten Tasks einen maximalen Datenwert von 65535 darstellen kann und bereits mit 30000 initialisiert wurde, kam es bei der Summenbildung zum numerischen Überlauf. Von den zehn durgeführten „Fuzzing Run"-Durchläufen war dies mit Ausnahme des zweiten Durchlaufes der Fall. Dies hat zur Folge, dass der Datenwert des Objektes einen Wert kleiner 30000 annehmen kann und so den Wertebereich verletzt. Diese Verletzung des Wertebereiches wurde vom Fuzzing Framework erkannt, wie zum Beispiel beim achte Durchlauf. Es zeigt sich jedoch auch, dass kein Fehler erkannt wird, wenn sich der übergelaufene Wert innerhalb des überwachten Wertebereiches befindet. Dies ist unter anderem im fünften Durchlauf der Fall.

6.1.2 Verletzung des Wertebereiches durch Underflow

Das Ziel der zweiten Versuchsdurchführung ist die Erzeugung eines Unterlaufes im ersten Task. Wie in der ersten Versuchsdurchführung soll der dabei entstandene Wert überprüft werden. Die Ergebnisse der zweiten Versuchsdurchführung aus Anhang B2.3 sind in Tabelle 6.2 aufgelistet. Eine Erläuterung zu den einzelnen Spalten und welche Daten sie enthalten findet sich in Kapitel 6.1.1.

Tabelle 6.2: Ergebnisse zum Verletzen des Wertebereiches durch Underflow

RunID	MessageID	TaskID	ChannelID	ChannelReaderID	DataID	OldValue	RuleId	NewValue
0	0	2	2	2	1	10000		
1	0	2	2	2	1	10000		
2	0	2	2	2	1	10000	6	changeSign
3	0	2	2	2	1	10000	6	changeSign
4	0	2	2	2	1	10000	6	changeSign
5	0	2	2	2	1	10000		
6	0	2	2	2	1	10000		
RunID	MessageID	TaskID	ChannelID	ChannelReaderID	DataID	OldValue	RuleId	NewValue
0	0	5	5	5	1	15000		
1	0	5	5	5	1	15000	8	0
2	0	5	5	5	1	60536	8	1
3	0	5	5	5	1	60536	8	1
4	0	5	5	5	1	60536	8	1
5	0	5	5	5	1	15000	8	0
6	0	5	5	5	1	15000	8	0

Wie der Tabelle 6.2 zu entnehmen ist, wurden fünf „Fuzzing Run"-Durchläufe durchgeführt. Der unmanipulierte Wert des Datenobjektes aus dem ersten TaskChannel beträgt 10000. Dieser wird bei der Entnahme durch den ersten Task bei drei Durchläufen mit der Regel „changeSign" manipuliert. Die Regel wurde so parametriert, dass sie mit einer Wahrscheinlichkeit von 0,5 ausgeführt wird. Im ersten Task wird der Wert dann zu einem mit 5000

initialisiertem Datenobjekt vom Typ uint16_t addiert. Dies hat für alle manipulierten Datenobjekte zur Folge, dass es bei der Addition zu einem numerischen Unterlauf kommt. Die Ergebnisse der Addition sind an den Werten des zweiten TaskChannels sichtbar. Die Regel „checkErrorRange“ überprüft im zweiten Versuchsdurchlauf den Wertebereich von 0 bis 20000. Diese erkennt die Unterläufe als Wertebereichsverletzungen.

6.1.3 Programmabsturz durch Division durch Null

In der letzten Versuchsdurchführung soll durch die Manipulation eines Datenwertes ein Programmabsturz hervorgerufen werden. Dazu wird der Wert des Datenobjektes aus dem ersten TaskChannel mit der Regel „changeValueZero“ zu Null manipuliert. Im ersten Task wird anschließend ein mit 5000 initialisiertes Datenobjekt von Typ uint16_t durch diesen Wert geteilt. Dies sollte bei zu Null manipulierten Werten zum Laufzeitfehler „Divide Zero“ führen und einen Programmabsturz verursachen.

Tabelle 6.3: Ergebnisse zum Programmabsturz durch Division durch Null

RunID	MessageID	TaskID	ChannelID	ChannelReaderID	DataID	OldValue	RuleId	NewValue
0	0	3	3	3	1	10000		
1	0	3	3	3	1	10000	3	0
RunID	MessageID	TaskID	ChannelID	ChannelReaderID	DataID	OldValue	RuleId	NewValue
0	0	6	6	6	1	0		

Der Tabelle 6.3 ist zu entnehmen, dass im ersten „Fuzzing Run“-Durchlauf eine Manipulation des Datenobjektes des ersten TaskChannels erfolgt ist. Diese Daten sind gleichzeitig die letzten, welche die Master Software von der Slave Software erhalten hat. Daran zeigt sich, dass die Master-Slave-Software immer die letzte Manipulation speichert, bevor das Datenobjekt an den Task weitergereicht wird. Die gespeicherten Daten befinden sich im Anhang B2.4. Dadurch ist der Nutzer über die letzte Aktion des Fuzzing Frameworks vor einem Programmabsturz informiert. Er kennt darüber hinaus die Eingabedaten, mit denen der Task arbeitet.

```
#2  0x00012898 in TheDivideZeroTask::execute (this=0xbefbc950)
    at examples/FuzzingErrorExamples/FuzzingClient/FuzzingErrorExamplesTasks.hpp:212
      s = 0xbefb4910
      x1 = 0
      returnvalue = <error reading variable returnvalue (Division by zero)>
      ptr = 0xbefb4a68
```

Abbildung 6.2: Ausschnitt der GDB Fehlermeldung nach dem Programmabsturz durch Division durch Null

Abbildung 6.2 enthält einen Ausschnitt der GDB [23] Fehlermeldung nach dem Programmabsturz. Die vollständige Fehlermeldung findet sich in Abbildung A.1 des Anhangs A3. Dem Ausschnitt ist zu entnehmen, dass der Fehler „Division by zero“ auftrat. Ebenfalls ist zu entnehmen, dass dieser in der „execute“ Funktion des Tasks „TheDivideZeroTask“ aufgetreten ist. Dies zeigt, dass sich ein aufgetretener Fehler zurückverfolgen lasst.

6.2 Messung der Bearbeitungszeit von Tasks

Die zweite Anwendung ist eine vom DLR erstellte Beispielanwendung für die Nutzung des Tasking Frameworks auf allen drei HPNs des ScOSA-Boards. Das Ziel der in Abbildung 6.3 dargestellten Anwendung ist, einen Zahlenwert vom Task 1.1 an alle anderen Tasks zu senden. Dies soll die Funktionsweise des verteilten Tasking Frameworks des DLR aufzuzeigen. Der unveränderte Quellcode des DLR findet sich im Anhang B3.1.

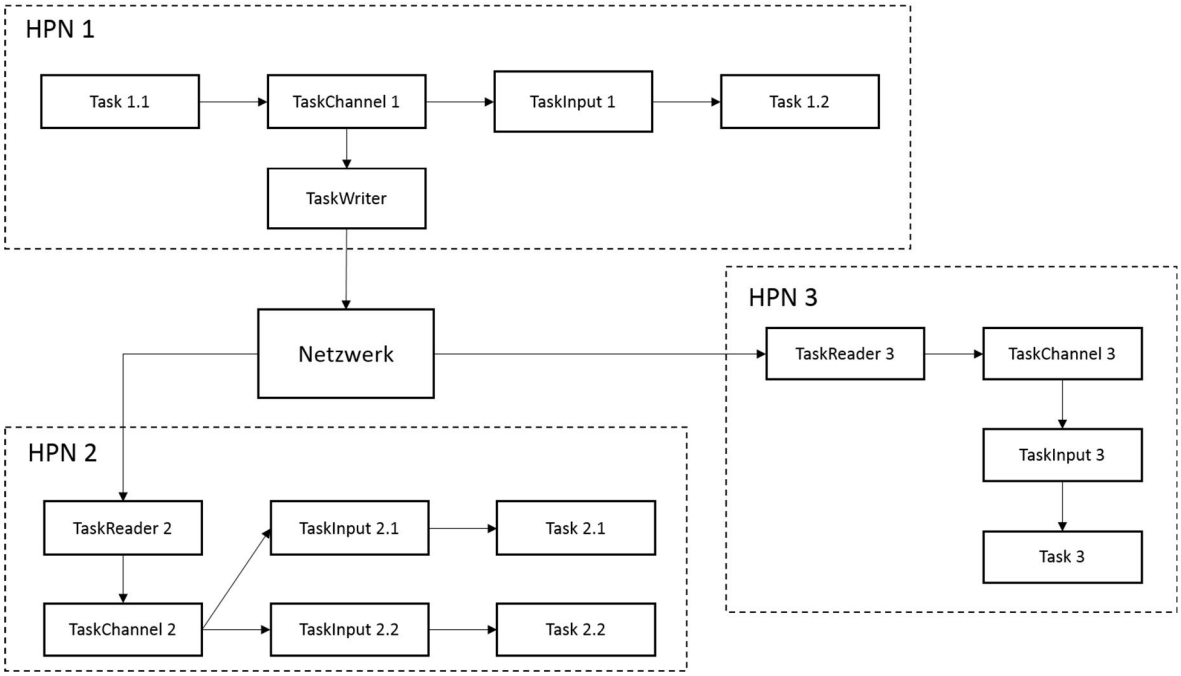

Abbildung 6.3: Datenflussdiagramm der Beispielanwendung für das verteilte Tasking Framework
Quelle: Eigene Zeichnung

Die Beispielanwendung wurde gewählt, um die Nutzung des Fuzzing Frameworks auf mehreren HPNs zu demonstrieren. Darüber hinaus soll die Messung der Bearbeitungszeit der Tasks aufgezeigt werden. Für diesen Zweck wurden drei Änderungen an der Beispielanwendung vorgenommen. Die erste Änderung betrifft den Typ des Datenobjektes, der für die

Übertragung des Zahlenwertes verwendet wird. Dieser wurde von uint8_t zu uint32_t geändert. Als zweite Änderung wurden die FifoReader der empfangenen Tasks wie in Kapitel 4.3.1 beschrieben durch FuzzingReader ersetzt. Dadurch wird die Slave Software mit der Beispielanwendung verbunden.

Die letzte Änderung betrifft die Tasks, welche den Zahlenwert empfangen. Diese führen nach der Änderung die Funktion „usleep(1000)“ in einer „for-Schleife“ aus. Die Anzahl der Wiederholungen entspricht dabei dem empfangenen Zahlenwert. Dadurch ist die Bearbeitungszeit des Tasks abhängig vom Zahlenwert. Es wird erwartet, dass die Bearbeitungszeit dem Zahlenwert in Millisekunden entspricht. Der für den Testdurchlauf verwendete Quellcode befindet sich im Anhang B3.2.

Um dies zu zeigen, wurden 100 Durchlaufe der Beispielanwendung durchgeführt. Als Zahlenwert ist dabei der konstante Wert von 100 übertragen worden. Das Fuzzing Framework manipuliert dabei den Zahlenwert vom TaskChannel 2, wenn dieser vom Task 2.1 eingelesen wird. Die Manipulation erfolgt dabei mittels der Regel „changeValueRel“, welche den ursprünglichen Zahlenwert um einen zufälligen Wert zwischen 0 und 300 Prozent erhöht.

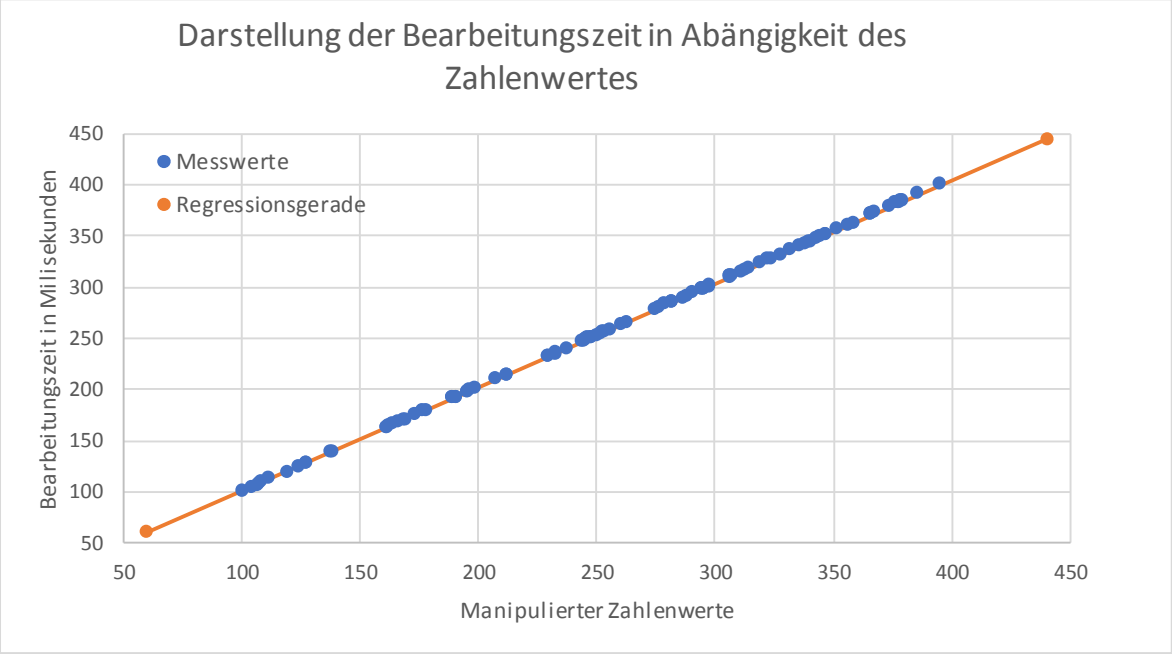

Abbildung 6.4: Darstellung der Bearbeitungszeit eines Tasks in Abhängigkeit des Zahlenwertes
Quelle: Eigenes Diagramm

Abbildung 6.4 zeigt die Gegenüberstellung vom manipulierten Zahlenwert des TaskChannels und der Bearbeitungszeit des Tasks. Wie erwartet, ist ein linearer Zusammenhang zwischen den beiden Werten zu erkennen. Die lineare Regressionsgerade hat die Funktion

y=1,0133x +0,0013. An 1,0133 ist zu erkennen, dass der Zahlenwert der Bearbeitungszeit in Millisekunden entspricht. Der Offset von 0,0013 kommt daher, dass das Betriebssystem neben dem Task Framework noch andere Prozesses im Hintergrund ausführt. Die Berechnung der Regressionsgeraden findet sich in Anhang A4. Damit ist gezeigt, dass das Fuzzing Framework in der Lage ist, eine Änderung der Bearbeitungszeit in Abhängigkeit von manipulierten Datenobjekten zu erkennen.

6.3 Testen einer Anwendung zur Bilderkennung

Das letzte Anwendungsbeispiel ist eine vom DLR erstellte Anwendung, welche die Funktionsweise des Tasking Frameworks Anhand eines „divide and conquer“ Algorithmus zur Bilderkennung demonstriert. Die Anwendung wird in zwei Testdurchläufen mittels Fuzzing Framework getestet. Im ersten Durchlauf werden die Datenobjekte zufällig manipuliert. Im zweiten Durchlauf werden einzelne Datenobjekte komplett gelöscht. In beiden Testdurchläufen traten durch die Manipulation Zugriffsverletzungen auf. Im dritten Testdurchlauf wird ein Belastungstest des Fuzzing Framework durchgeführt.

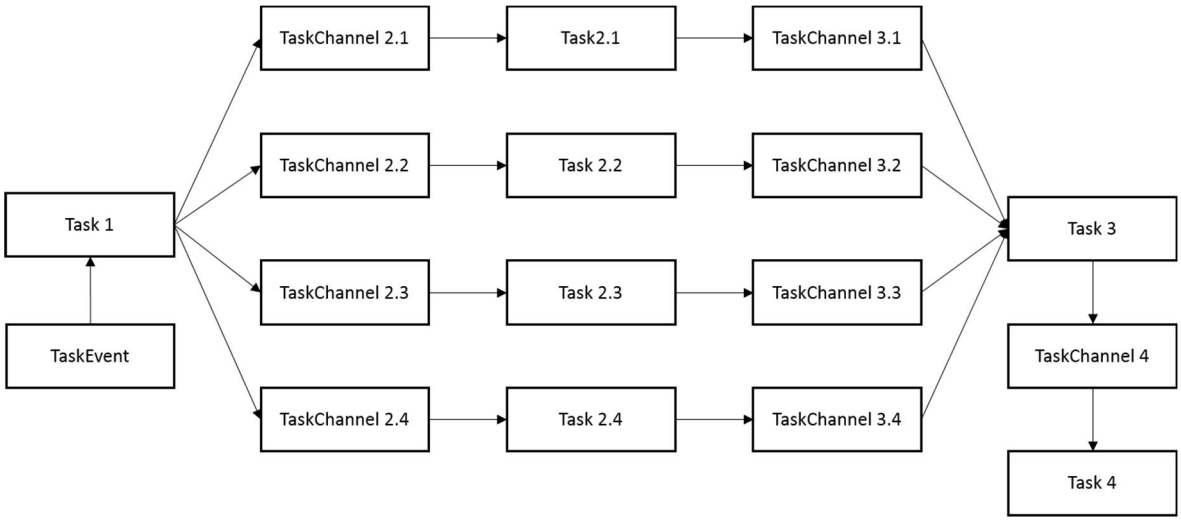

Abbildung 6.5: Datenflussdiagramm der Image Tracking Anwendung.
Die TaskInputs sind wegen der Übersichtlichkeit nicht enthalten.
Quelle: Eigene Zeichnung

Das Ziel der Beispielanwendung ist die Erkennung eines geometrischen Objektes in einem Farbbild. Dafür liest der Task 1 aus Abbildung 6.5 das Farbbild ein und wandelt dies in ein Graustufenbild um. Anschließend wird dieses in vier Bildausschnitte zerlegt und an die vier

Task 2 aufgeteilt. Hier wird in den Bildern mittels Canny-Algorithmus nach dem geometrischen Objekt gesucht. Die Ergebnisse der Suche werden als Bildkoordinate an den Task 3 weitergereicht, der diese zusammenfasst und das Ergebnis ausgibt

Da die Beispielanwendung mit einer älteren Version des Tasking erstellt wurde, ist die ursprüngliche Version nicht mit dem Fuzzing Framework kompatibel. Deshalb wurden die Komponenten des Tasking Frameworks durch deren neuere Version ersetzt. Dadurch ändert sich weder der Aufbau der Anwendung noch deren Funktionalität. Zusätzlich wurde die Anwendung um den Task 4 erweitert, welcher vom Task 3 benachrichtigt wird, ob ein geometrisches Objekt erkannt wurde. Dadurch kann das Ergebnis der Bilderkennung vom Fuzzing Framework überwacht werden. Der Quellcode der alten Version befindet sich im digitalen Anhang B4.1. Der für die Testdurchläufe verwendete Quellcode befindet sich im Anhang B4.2.

6.3.1 Zugriffsverletzungen durch zufällige Werteänderung

Im ersten Testdurchlauf soll die Beispielanwendung durch die zufällige Manipulation der Datenobjekte in den TaskChannel getestet werden. Die Manipulation erfolgt durch die Regel „changeRandomBit“, welche mit einer Wahrscheinlichkeit von 0.1 parametriert wurde. Diese ändert den Wert eines einzelnen Bits des Datenobjektes. Der erste Testdurchlauf umfasst den „Profiling Run“ und vier „Fuzzing Runs“. Die Beispielanwendung liest jedes Mal acht Beispielbilder ein und wertet diese aus.

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0xb6469450 (LWP 790)]
0x000155a4 in areaMarker (inputImgName=<optimized out>,
    finalMarkImg=0xb6468d48 "./examples/FuzzingImageTracking/Debug/landingZone_final7.ppm",
topEdge=427700, lowEdge=224253895)
    at examples/FuzzingImageTracking/Client/algorithm/ObjectRecognition.hpp:191
191       examples/FuzzingImageTracking/Client/algorithm/ObjectRecognition.hpp: No such file or directory.
#0  0x000155a4 in areaMarker (inputImgName=<optimized out>,
    finalMarkImg=0xb6468d48  "./examples/FuzzingImageTracking/Debug/landingZone_final7.ppm",  topEdge=427700, lowEdge=224253895)
    at examples/FuzzingImageTracking/Client/algorithm/ObjectRecognition.hpp:191
#1  0x00015d10 in LandingAreaMarkTaskFuzzy::execute (
    this=0x8eb1c0 <landAreaMarkTK>)
    at examples/FuzzingImageTracking/Client/TasksImgTrack.hpp:311
```

Abbildung 6.6: Ausschnitt der GDB Fehlermeldung nach dem Programmabsturz durch zufällige Manipulation Information zum Fehlerursprung sind rot markiert. Datenwerte

Während des Testdurchlaufes trat im dritten „Fuzzing Run“ eine Zugriffverletzung auf, die zum Programmabsturz führte. Abbildung 6.6 enthält einen Ausschnitt der GDB Fehlermeldung. Die vollständige Fehlermeldung findet sich in Abbildung A.2 des Anhangs A5. Dem Ausschnitt ist zu entnehmen, dass der Fehler in der Funktion „areaMarker“ in Zeile 191 der Datei ObjectRecognition.hpp auftrat. Desweiten ist die Information zum ausführenden Task enthalten, in dem der Fehler auftrat. Dies ist der „LandingAreaMarkTaskFuzzy“, welcher dem Task 3 aus Abbildung 6.5 entspricht. Im digitalen Anhang B4.3 befindet sich die Tabelle mit den Daten, welche die Master Software von der Slave Software erhalten hat. Dieser ist zu entnehmen, dass der Fehler bei der Bearbeitung des ersten Beispielbildes im dritten „Fuzzing Run“ erfolgte. Um den aufgetreten Fehler weiter zu untersuchen, wird der Quellcode der Daten ObjectRecognition.hpp um Zeile 191 betrachtet. Der gesamte Quellcode der Funktion „areaMarker“ ist in Abbildung A.4 des Anhangs einzusehen. In Abbildung 6.7 ist ein Ausschnitt um Zeile 191 dargestellt.

```
// mark the position
for(int i = -5; i < 5; i++)
{
    rgb_image[locY+locX+i].Red = rgb_image[locY+locX+i].Green = rgb_image[locY+locX+i].Blue = 0;
}
```

Abbildung 6.7: Ausschnitt des Quellcodes, der zu einem Absturz bei der zufälligen Manipulation führte

Demnach trat die Zugriffsverletzung beim Zugriff auf das Array „rgb_image“ auf. Dessen Index hängt von den Werten „locY“ und „locX“ ab, welche aus den Werten von „topEdge“ und „lowEdge“ aus Abbildung 6.6 berechnet werden. Deren Werte berechnen sich direkt aus den Ergebnissen der zweiten Tasks und können somit bei der zufälligen Manipulation geändert worden sein. Aus den Daten der Master Software aus dem digitalen Anhang ist ersichtlich, dass im letzten aufgenommenen Durchlauf zwei zufällige Manipulationen der Ergebnisse durchgeführt wurden.

6.3.2 Löschen eines Datenobjektes aus dem TaskChannel

Der zweite Testdurchlauf soll die Beispielanwendung durch das Löschen von Datenobjekten aus den TaskChanneln testen. Das Löschen erfolgt mittels der Regel „dropDataobject“. Diese ist mit der Wahrscheinlichkeit von 0,1 parametriert. Will der Task das Datenobjekt entnehmen, erhält er statt der Referenz zum Datenobjekt einen Null pointer.

Wie den Daten der Master Software aus dem digitalen Anhang B4.4 zu entnehmen ist, trat ein Programmabsturz im ersten „Fuzzing Run“ bei der Bearbeitung des ersten Bildes auf. Dem Ausschnitt der GDB Fehlermeldung aus Abbildung 6.8 ist zu entnehmen, dass der Fehler in der „gaussian_smooth“ Funktion auftrat. Der genaue Ort des Fehlers ist Zeile 481

der Datei canny_edge.cpp. Die vollständige Fehlermeldung befindet sich in Abbildung A.3 des Anhangs. Des Weiteren ist die Information enthalten, dass der Fehler im Task „ImgChunkProcessingTask“ auftrat.

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0xb6469450 (LWP 774)]
gaussian_smooth (image=image@entry=0x0, rows=85128, rows@entry=600,
    cols=cols@entry=600, sigma=sigma@entry=0.00999999978,
    smoothedim=smoothedim@entry=0xb6468d60)
    at examples/FuzzingImageTracking/Client/algorithm/canny_edge.cpp:481

                                                    .
                                                    .
                                                    .

#3  0x00015a5c in ImgChunkProcessingTaskFuzzy::execute (
    this=0x61c2a0 <imgChunkProcTK_0>)
    at examples/FuzzingImageTracking/Client/TasksImgTrack.hpp:170
```

Abbildung 6.8: Ausschnitt der GDB Fehlermeldung nach dem Programmabsturz durch das Löschen des Datenobjektes

Zur weiteren Untersuchung des Fehlers wird der an die Funktion „gaugassian_smooth“ übergeben Wert des Pointers „image“ betrachtet. Die GDB Fehlermeldung zeigt, dass dieser auf NULL zeigt. Dies entspricht der Referenz des Datenobjektes, welche der Task vom TaskChannel erhalten hat, nachdem das Datenobjekt gelöscht wurde.

6.3.3 Belastungstest für das Fuzzing Framework

In einem dritten Testdurchlauf wird ein Belastungstest für das Fuzzing Framework durchgeführt. Die dritte Beispielanwendung ist dafür geeignet, weil sie Farbbilder als Datenobjekte in den TaskChanneln speichert. Deshalb müssen während des Testdurchlaufes große Datenmengen von der Slave Software an die Master Software gesendet und anschließend gespeichert werden.

Für den Belastungstest wurde die Beispielanwendung mit 99 „Fuzzing Runs“ ausgeführt. Zusammen mit dem „Profiling Run“ ergeben sich damit 100 Durchläufe. Damit in den Durchläufen keine Fehler durch die Manipulation der Datenobjekte auftreten, sind alle Regeln mit einer Wahrscheinlichkeit von 0.0 parametriert worden.

Der Belastungstest konnte erfolgreich beendet werden. Dabei wurden 11,6 Gigabyte Daten in der SQL-Datenbank gespeichert. Aus diesem Grund ist im digitalen Anhang B4.5 nur die csv-Datei mit den Ergebnissen enthalten.

7 Zusammenfassung

Das Ziel dieser Master Thesis ist es, zu untersuchen, ob und wie Fuzzing zum Testen des Tasking Frameworks verwendet werden kann. Dafür werden im Kapitel 2 die theoretischen Grundlagen zu Fuzzing erläutert. Dabei wird unter anderem auf die verschiedenen Bestanteile des Fuzzing Frameworks eingegangen. Ein zweiter Schwerpunkt liegt in der Klassifizierung verschiedener Fuzzing-Ansätze. Das Tasking Framework des DLR wird in Kapitel 3 vorgestellt. Dies geschieht im Rahmen des ScOSA-Projektes, in welches auch diese Master Arbeit eingebunden ist.

In Kapitel 4 wird die Master-Slave-Software vorgestellt, welche den Kern des Fuzzing Frameworks bildet. Für diese werden zu Beginn Rahmenbedingungen und Qualitätsziele definiert. Anschließend folgt die Architektur und Funktionsweise der Software. Das wichtigste Merkmal ist die Aufteilung der Software in zwei Komponenten. Die Slave Software wird mit der zu testenden Anwendung auf dem ScOSA-Board ausgeführt. Über eine TCP/IP Verbindung ist diese mit der Master Software verbunden, welche separat auf einem Desktop-Rechner läuft. Kapitel 4 beschreibt darüber hinaus die Anbindung der Slave Software an die zu testende Anwendung. Ebenfalls wird die Speicherung von Ergebnissen durch die Master Software in einer SQLite Datenbank erläutert.

Das fünfte Kapitel befasst sich mit der Manipulation der Daten. Diese erfolgt durch Regeln, welche die Master Software erstellt und von der Slave Software ausgeführt werden. Jede der insgesamt sieben Regeln entspricht dabei einer Anweisung zur Manipulation. Im Kapitel werden die verschiedenen Regeln des Fuzzing Frameworks erläutert und wie diese aufgebaut sind. Ebenfalls wird die Ausführung der Regeln betrachtet. Den Schluss des Kapitels bilden die Funktionen zur Erkennung von Fehlern, die durch das Fuzzing verursacht wurden. Dies ist zum einen die Ausführung der Anwendung mit dem GDB Debugger, um bei einem Programmabsturz eine Fehlermeldung zu erhalten. Des Weiteren erlaubt eine Funktion die Überprüfung eines Wertebereiches für die Werte bestimmter Datenobjekte. Die dritte Funktion erlaubt die Aufnahme von Bearbeitungszeit der zu testenden Anwendung.

Im sechsten Kapitel wird das Fuzzing Framework zum Testen dreier Beispielanwendung verwendet. Das erste Beispiel demonstriert die Manipulation eines einzelnen Datenobjektes und die daraus resultierende Verletzung eines überwachten Wertebereiches. Die zweite Beispielanwendung wird auf mehreren HPNs des ScOSA-Boards ausgeführt. Sie zeigt die Anwendung des Fuzzing Frameworks auf das verteilte Tasking Framework des DLR. Dafür wurden die Bearbeitungszeiten verschiedener Tasks gemessen. Als Ergebnis wird der Zusammenhang zwischen manipulierten Datenwerten und der Bearbeitungszeit eines Tasks

dargestellt. Die dritte Beispielanwendung demonstriert die Funktionsweise des Tasking Frameworks anhand eines „divide and conquer“-Algorithmus zur Bilderkennung. Das Testen der Anwendung erfolgte separat durch zwei Regeln. Mit beiden konnte ein Absturz der Anwendung aufgrund einer Zugriffsverletzung erzeugt werden. Anhand der vorhandenen Daten wurde anschließend demonstriert, wie aufgetretene Fehler untersucht werden können. Zusätzlich sind in einem Belastungstest 11,6 Gigabyte an Daten von der Slave Software an die Master Software übertragen worden.

8 Fazit

Das Ziel dieser Master Thesis war es, zu untersuchen, ob und wie Fuzzing zum Testen des Tasking Frameworks verwendet werden kann. Dafür wurde eine Regelbasiertes Fuzzing Framework entworfen, welches korrekte Eingabedaten manipuliert. Zum Nachweis der Funktionalität wurde verschiedene Testdurchläufe an drei Beispielanwendungen durchgeführt. Aus den dabei aufgenommenen Daten konnte anschließend die Zeile des Quellcodes ermittelt werden, in welcher der Fehler auftrat. Darüber hinaus ließ sich der Task bestimmen, bei dessen Ausführung es zum Programmabsturz kam. Aus den Daten ist außerdem ersichtlich, welche Eingabedaten der Task vor dem Absturz von den TaskChanneln erhalten hat.

Damit ist die Zweite der in der Masterarbeit definierten Rahmenbedingungen erfüllt. Nach dieser muss das Fuzzing Framework ausreichend Daten speichern, um einen aufgetretenen Fehler zurückzuverfolgen. Die erste Rahmenbedingung legt fest, dass die zu testenden Anwendungen bei den Tests auf dem ScOSA-Board ausgeführt werden. Dies war bei allen drei Beispielanwendungen der Fall, womit die Rahmenbedingung erfüllt ist. Darüber hinaus sind Qualitätsziele definiert worden, welche erfüllt wurden. So zeigt die dritte Beispielanwendung, welche mit einer alten Version des Tasking Framework realisiert wurde, die allgemeine Anwendbarkeit des Fuzzing Frameworks. Die Aufteilung der Software in eine Master - und Slave Komponente, zusammen mit dem Interface, minimiert den Eingriff in das Tasking Framework und vereinfacht die Handhabung. Die Automatisierung wird durch das automatische Erstellen der Regeln erreicht.

Zusätzlich zu der Erkennung von Programmabstürzen ist das Fuzzing Framework in der Lage, die Wertebereiche bestimmter Datenobjekte zu überwachen. Ebenfalls ist eine Messung der Bearbeitungszeiten eines Tasks möglich. Somit verfügt das Fuzzing Framework über zwei zusätzliche Funktionen, um das Verhalten der zu testenden Anwendung zu überwachen.

Zusammengefasst ergibt sich das Ergebnis, dass das in Rahmen dieser Masterarbeit vorgestellte Fuzzing Framework geeignet ist, um mit dem verteilten Tasking Framework realisierte Anwendungen mittels Fuzzing zu testen.

9 Ausblick

Das verteilte Tasking Framework wurde vom DLR als Middleware für verschieden Anwendungen entwickelt. Das im Rahmen der Masterarbeit entwickelte Fuzzing Framework erlaubt es, zukünftig erstellte Anwendung während oder nach der Entwicklung zu testen. Anhand der Beispielanwendung zur Bilderkennung wurde darüber hinaus gezeigt, dass ebenfalls eine rückwirkende Anwendung möglich ist.

Ein Ziel weiterer Entwicklungen am Fuzzing Framework sollten sein, die Nutzerfreundlichkeit zu verbessern. Ein erster Ansatz dafür ist ein automatischer Neustart der zu testenden Anwendung nach einem Programmabsturz. Das Fuzzing Framework archiviert nach jedem Absturz die Daten und Fehlermeldungen. Der Nutzer erhält anschließend eine Auflistung aller Fehler und Daten, die im Testzeitraum aufgetreten sind. Ein zweiter Ansatz ist die Weiterentwicklung der Fehler- und Datendarstellung. Dies soll die Rückverfolgung von Fehlern durch den Nutzer vereinfachen.

Ein weiteres Ziel ist die Erweiterung der Funktionalität des Fuzzing Framework. Dies beinhalt die Implementierung neuer Regeln für die Manipulation. Ein Ansatz dafür sind Regeln, welche auf den Trend von Werten der Datenobjekte über mehrere Durchläufe eingehen. Eine zusätzliche Funktionalität für die Fehlererkennung ist die Überwachung von verschiedenen Modi der Tasks während des Programmdurchlaufes.

Als erste Anwendung auf ein großes Projekt bietet sich das Autonomous Terain-based Optical Navigation(ATON) Projekt des DLR an. Dieses soll die neuste Version des verteilten Tasking Framewoks nutzen. Ein Schwerpunkt von ATON liegt in der Bildverarbeitung. Damit gibt es eine Ähnlichkeit zu der Beispielanwendung aus Kapitel 6.3. Ein weiter Vorteil der Bildverarbeitung von ATON ist, dass große Mengen an Daten anfallen. Darüber hinaus müssen bestimmte Bilder mit einer Frequenz von 100 Hz verarbeitet werden. Dies stellt in dieser Hinsicht hohe Anforderungen an das Fuzzing Framework und zeigt damit dessen Leistungsfähigkeit.

Ein weiteres Anwendungsbereich ist die Nutzung des Fuzzing Framework zu ausschließlichen Überwachung der Anwendung. Das Framework sammelt dabei ausschließlich Daten zur Anwendung, führt aber keine Manipulationen aus

Literaturverzeichnis

[1] B. P. Miller, L. Fredriksen und B. So, „An Empirical Study of the Reliability of UNIX Utilities," *Commun. ACM,* Bd. 33, pp. 32-44, #dec# 1990.

[2] M. Greene, A. Amini und P. Sutton, Fuzzing: Brute Force Vulnerability Discovery, Addison-Wesley, 2007.

[3] A. Takanen, J. D. Demott und C. Miller, Fuzzing for software security testing and quality assurance, Artech House, 2008.

[4] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan und J. Steidl, „Fuzz revisited: A re-examination of the reliability of UNIX utilities and services," 1995.

[5] U. of Oulu, „PROTOS - Security Testing of Protocol Implementations," [Online]. Available: https://www.ee.oulu.fi/research/ouspg/Protos. [Zugriff am 01 05 2017].

[6] R. McNally, K. Yiu, D. Grove und D. Gerhardy, „Fuzzing: the state of the art," 2012.

[7] R. Kaksonen, „A functional method for assessing protocol implementation security. Publication 448, VTT Electronics," *Telecommunication Systems, Kaitov{\"a}yl{\"a},* Bd. 1, 2001.

[8] D. Aitel, „The advantages of block-based protocol analysis for security testing," *Immunity Inc., February,* Bd. 105, p. 106, 2002.

[9] P. Godefroid, M. Y. Levin, D. A. Molnar und others, „Automated Whitebox Fuzz Testing.," in *NDSS*, 2008.

[10] Google, „OSS-Fuzz - Continuous Fuzzing for Open Source Software," 2016. [Online]. Available: https://github.com/google/oss-fuzz. [Zugriff am 01 05 2017].

[11] A. G. Voyiatzis, K. Katsigiannis und S. Koubias, „A Modbus/TCP Fuzzer for Testing Internetworked Industrial Systems," in *{PROCEEDINGS OF 2015 IEEE 20TH CONFERENCE ON EMERGING TECHNOLOGIES \& FACTORY AUTOMATION (ETFA)}*, 2015.

[12] H. Lee, K. Choi, K. Chung, J. Kim und K. Yim, „Fuzzing CAN Packets into Automobiles," in *{2015 IEEE 29th International Conference on Advanced Information Networking and Applications (IEEE AINA 2015)}*, 2015.

[13] D. Lüdtke, K. Westerdorff, K. Stohlmann, A. Börner, O. Maibaum, T. Peng, B. Weps, G. Fey und A. Gerndt, „OBC-NG: Towards a reconfigurable on-board computing architecture for spacecraft," in *2014 IEEE Aerospace Conference*, 2014.

[14] ESA, „ESA SpaceWire," 2016. [Online]. Available: http://spacewire.esa.int/content/Home/HomeIntro.php. [Zugriff am 01 06 2017].

[15] RTEMS, „RTEMS Real Time Operating System(RTOS)," 2017. [Online]. Available: https://www.rtems.org/. [Zugriff am 01 06 2017].

[16] L. Foundation, „Yocto Project," 2017. [Online]. Available: https://www.yoctoproject.org/. [Zugriff am 06 06 2017].

[17] L. Foundation, „Poky," 2017. [Online]. Available: https://www.yoctoproject.org/tools-resources/projects/poky. [Zugriff am 06 06 2017].

[18] O. Maibaum, D. Lüdtke und A. Gerndt, „Tasking Framework: Parallelization of Computations in Onboard Control Systems," in *ITG/GI Fachgruppentreffen Betriebssysteme*, 2013.

[19] „Open modUlar sofTware PlatfOrm for SpacecrafT (OUPOST)," 2017. [Online]. Available: https://github.com/DLR-RY/outpost-core. [Zugriff am 04 07 2017].

[20] A. S. Tanenbaum, „Computernetzwerke, 3. Auflage," München, Pearson Studium, 2000, pp. 53-55.

[21] „SQLite," 2017. [Online]. Available: https://www.sqlite.org/. [Zugriff am 29 06 2017].

[22] „SQLite Studio," 2017. [Online]. Available: https://sqlitestudio.pl/index.rvt. [Zugriff am 25 09 2017].

[23] „GDB, the GNU Debugger," Free Software Foundation, 2017. [Online]. Available: https://www.gnu.org/software/gdb/. [Zugriff am 31 07 2017].

[24] L. Papula, „Mathematische formelsammlung," Bd. 11, Springer, 2014, pp. 307-310.

Anhang A

A1 IDs der Protokolle

Tabelle A.1: Auflistung der Datentypen bei der Serialisierung von Datenobjekten

Bytewert von DataTyp	Datentyp
1	int32_t
2	uint8_t
3	uint16_t
4	uint32_t
5	uint64_t
6	float
7	double
8	uint8_t *

Tabelle A.2: Auflistung der PacketTyp Ids für die serielle Übertragung mittels TCP

Bytewert von PacketTyp	Bedeutung
1	profilingRun
2	returnRules
3	fuzzingRun
4	fuzzingRunDone
5	error
6	finished
7	Error

Tabelle A.3: Auflistung der einzelnen Regel und deren RuleID

Bytewert von RuleID	Regel
1	dropDataobject
2	changeRandomBit
3	changeValueZero
4	changeValueAbs
5	changeValueRel
6	changeSign
7	changeFractionalPart
8	checkErrorRange

A2 Anwendbarkeit von Regeln abhängig vom Datentyp

Tabelle A.4: Auflistung welche Regeln auf welche Datentypen angewandt werden können. Das „x“ markiert, dass die Anwendung möglich ist.

Regel\ Datentyp	int (32bit)	uint8_t	uint16_t	uint32_t	uint64_t	float	double	uint8_t*
dropDataobject	x	x	x	x	x	x	x	x
changeRandomBit	x	x	x	x	x	x	x	x
changeValueZero	x	x	x	x	x	x	x	
changeValueAbs	x	x	x	x	x	x	x	
changeValueRel	x	x	x	x	x	x	x	
changeSign	x					x	x	
changeFractionalPart						x	x	
checkErrorRange	x	x	x	x	x	x	x	

A3 GDB Fehlermeldungen und Quellcode der ersten Beispielanwendung

```
Starting program: /home/root/FuzzingErrorExamplesClient
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/libthread_db.so.1".
[New Thread 0xb6c69450 (LWP 880)]
[New Thread 0xb6469450 (LWP 881)]

Program received signal SIGFPE, Arithmetic exception.
[Switching to Thread 0xb6469450 (LWP 881)]
0xb6fa0838 in raise (sig=8)
   at /usr/src/debug/glibc/2.23-r0/git/sysdeps/unix/sysv/linux/pt-raise.c:35
35          return INLINE_SYSCALL (tgkill, 3, pid, THREAD_GETMEM (THREAD_SELF, tid),
#0  0xb6fa0838 in raise (sig=8)
   at /usr/src/debug/glibc/2.23-r0/git/sysdeps/unix/sysv/linux/pt-raise.c:35
      _sys_result = 0
      pid = 879
#1  0xb6db312c in __aeabi_ldiv0 () from /lib/libgcc_s.so.1
No symbol table info available.
#2  0x00012898 in TheDivideZeroTask::execute (this=0xbefbc950)
   at examples/FuzzingErrorExamples/FuzzingClient/FuzzingErrorExamplesTasks.hpp:212
      s = 0xbefb4910
      x1 = 0
      returnvalue = <error reading variable returnvalue (Division by zero)>
      ptr = 0xbefb4a68
#3  0x00014300 in Tasking::Scheduler::callExecution (data=data@entry=0x30fb8)
   at ScOSA_Tasking/src/linux/scheduler.cpp:109
      task = 0xbefbc950
      event = <optimized out>
#4  0x00014cb4 in Tasking::executionThread (management=0x30fb8)
   at ScOSA_Tasking/src/linux/scheduler.cpp:61
      __PRETTY_FUNCTION__ = "void* Tasking::executionThread(void*)"
      data = 0x30fb8
#5  0xb6f94f00 in start_thread (arg=0xb6469450)
   at /usr/src/debug/glibc/2.23-r0/git/nptl/pthread_create.c:335
Quit
```

Abbildung A.1: Fehlerausgabe von GDB nach Programmabsturz durch Division durch Null

```
Starting program: /home/root/fuzzingImageTrackingClient
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/libthread_db.so.1".
[New Thread 0xb6c69450 (LWP 789)]
[New Thread 0xb6469450 (LWP 790)]

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0xb6469450 (LWP 790)]
0x000155a4 in areaMarker (inputImgName=<optimized out>,
   finalMarkImg=0xb6468d48     "./examples/FuzzingImageTracking/Debug/landingZone_final7.ppm",     to-
pEdge=427700, lowEdge=224253895)
   at examples/FuzzingImageTracking/Client/algorithm/ObjectRecognition.hpp:191
191       examples/FuzzingImageTracking/Client/algorithm/ObjectRecognition.hpp: No such file or directory.
#0  0x000155a4 in areaMarker (inputImgName=<optimized out>,
   finalMarkImg=0xb6468d48     "./examples/FuzzingImageTracking/Debug/landingZone_final7.ppm",     to-
pEdge=427700, lowEdge=224253895)
   at examples/FuzzingImageTracking/Client/algorithm/ObjectRecognition.hpp:191
#1  0x00015d10 in LandingAreaMarkTaskFuzzy::execute (
   this=0x8eb1c0 <landAreaMarkTK>)
   at examples/FuzzingImageTracking/Client/TasksImgTrack.hpp:311
#2  0x000182f0 in Tasking::Scheduler::callExecution (data=data@entry=0x10d72b8)
   at ScOSA_Tasking/src/linux/scheduler.cpp:109
#3  0x00018ca4 in Tasking::executionThread (management=0x10d72b8)
   at ScOSA_Tasking/src/linux/scheduler.cpp:61
#4  0xb6f94f00 in start_thread (arg=0xb6469450)
   at /usr/src/debug/glibc/2.23-r0/git/nptl/pthread_create.c:335
#5  0xb6d38510 in ?? () at ../sysdeps/unix/sysv/linux/arm/clone.S:89
  from /lib/libc.so.6
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
```

Abbildung A.2: Fehlerausgabe von GDB nach Programmabsturz durch zufällige Manipulation

A4 Berechnung der linearen Regressionsgeraden

Anhand der aufgenommenen Messwerte der Bearbeitungszeit des Task soll die lineare Regressionsgerade bestimmt werden. Die Gerade soll die Form y= ax+b haben, wobei y die Bearbeitungszeit ist und x der manipulierte Zahlenwert. In Formel 1 ist die Berechnung von a dargestellt.

$$a = \frac{n \cdot \sum_{i=1}^{n} x_i \cdot y_i - (\sum_{i=1}^{n} x_i) \cdot (\sum_{i=1}^{n} y_i)}{n \cdot \sum_{i=1}^{n} {x_i}^2 - (\sum_{i=1}^{n} x_i)^2} \quad (1)$$

Anschließend kann b mit Formel 2 berechnet werden.

$$b = \frac{(\sum_{i=1}^{n} {x_i}^2) \cdot (\sum_{i=1}^{n} y_i) - (\sum_{i=1}^{n} x_i) \cdot (\sum_{i=1}^{n} x_i \cdot y_i)}{n \cdot \sum_{i=1}^{n} {x_i}^2 - (\sum_{i=1}^{n} x_i)^2} \quad (2)$$

Beide Formeln stammen aus dem Papula [24]. Die für die Berechnung verwendeten Zahlenwerte und Zwischenergebnisse finden sich im digitalen Anhang B3.3. Daraus ergeben sich die Werte a= 1,0133 und b= 0,0012.

A5 GDB Fehlermeldungen und Quellcode der dritten Beispielanwendung

```
Starting program: /home/root/fuzzingImageTrackingClient
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/libthread_db.so.1".
[New Thread 0xb6c69450 (LWP 773)]
[New Thread 0xb6469450 (LWP 774)]

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0xb6469450 (LWP 774)]
gaussian_smooth (image=image@entry=0x0, rows=85128, rows@entry=600,
    cols=cols@entry=600, sigma=sigma@entry=0.00999999978,
    smoothedim=smoothedim@entry=0xb6468d60)
    at examples/FuzzingImageTracking/Client/algorithm/canny_edge.cpp:481
481         examples/FuzzingImageTracking/Client/algorithm/canny_edge.cpp: No such file or directory.
#0  gaussian_smooth (image=image@entry=0x0, rows=85128, rows@entry=600,
    cols=cols@entry=600, sigma=sigma@entry=0.00999999978,
    smoothedim=smoothedim@entry=0xb6468d60)
    at examples/FuzzingImageTracking/Client/algorithm/canny_edge.cpp:481
#1  0x00014f4c in canny (image=image@entry=0x0, rows=rows@entry=600,
    cols=cols@entry=600, sigma=sigma@entry=0.00999999978,
    tlow=tlow@entry=0.300000012, thigh=thigh@entry=0.00499999989,
    edge=0xb6468d9c, edge@entry=0xb6468d94)
    at examples/FuzzingImageTracking/Client/algorithm/canny_edge.cpp:201
#2  0x00015058 in cannyEdgeMain (gray_image=gray_image@entry=0x0,
    edge_img=0x15058 <cannyEdgeMain(unsigned char*, unsigned char*)+40> "\030\020\237\345\004",
    edge_img@entry=0x67801c <imgChunkProcTK_0+376188> '\377' <repeats 200 times>...) at examples/FuzzingImageTracking/Client/algorithm/canny_edge.cpp:158
#3  0x00015a5c in ImgChunkProcessingTaskFuzzy::execute (
    this=0x61c2a0 <imgChunkProcTK_0>)
    at examples/FuzzingImageTracking/Client/TasksImgTrack.hpp:170
#4  0x000182f0 in Tasking::Scheduler::callExecution (data=data@entry=0x10d72b8)
    at ScOSA_Tasking/src/linux/scheduler.cpp:109
#5  0x00018ca4 in Tasking::executionThread (management=0x10d72b8)
    at ScOSA_Tasking/src/linux/scheduler.cpp:61
#6  0xb6f94f00 in start_thread (arg=0xb6469450)
    at /usr/src/debug/glibc/2.23-r0/git/nptl/pthread_create.c:335
Quit
```

Abbildung A.3: Fehlerausgabe von GDB nach dem Programmabsturz durch das Löschen eines Datenobjektes

```
void areaMarker(char* inputImgName,char* finalMarkImg,int topEdge, int lowEdge){

	int locY=0,locX=0;
	static rgb_pixel_t rgb_image[IMG_SIZE_TOTAL+100];
	if(topEdge==0 && lowEdge==0){
		cout << "Nothing detected.\n";
		return;
	}
	// Read in the PPM format RGB color image file to 1D array with RGB structure
	//input_ppm((char *)rgb_image, 3*IMG_SIZE_TOTAL,inputImgName,false);
	input_ppm((char *)rgb_image, IMG_SIZE_TOTAL,inputImgName,false);


	locY = ((lowEdge/IMG_HEIGHT_TOTAL - topEdge/IMG_HEIGHT_TOTAL)/2 + to-
pEdge/IMG_HEIGHT_TOTAL)*IMG_WIDTH_TOTAL;
	if((topEdge%IMG_WIDTH - lowEdge%IMG_WIDTH) <= 0)
		locX = (topEdge%IMG_WIDTH_TOTAL - lowEdge%IMG_WIDTH_TOTAL)/2 +lo-
wEdge%IMG_WIDTH_TOTAL;
	else
		locX = (lowEdge%IMG_WIDTH_TOTAL -topEdge%IMG_WIDTH_TOTAL)/2 +to-
pEdge%IMG_WIDTH_TOTAL;

	// mark the position
	for(int i = -5; i < 5; i++)
	{
		rgb_image[locY+locX+i].Red = rgb_image[locY+locX+i].Green =
rgb_image[locY+locX+i].Blue = 0;
	}

	//marking the edges
	rgb_image[topEdge].Red = rgb_image[topEdge].Green = rgb_image[topEdge].Blue = 0;
	rgb_image[lowEdge].Red = rgb_image[lowEdge].Green = rgb_image[lowEdge].Blue = 0;


	// Write out the PMM format image
	//output_ppm((char *)rgb_image, 3*IMG_SIZE_TOTAL, finalMarkImg,false);
	output_ppm((char *)rgb_image, IMG_SIZE_TOTAL, finalMarkImg,false);

}
```

Abbildung A.4: Quellcode der zu einem Absturz bei der zufälligen Manipulation führte

Digitaler Anhang B

B1 Quellcode des Fuzzing Frameworks

B2 Das erste Anwendungsbeispiel

B2.1 Quellcode der Beispielanwendung

B2.2 Ergebnisse vom Overflow Testdurchlauf

B2.3 Ergebnisse vom Underflow Testdurchlauf

B2.4 Ergebnisse der Division durch NULL

B3 Das zweite Anwendungsbeispiel

B3.1 Unveränderter Quellcode des DLR

B3.2 Für den Testdurchlauf verwendeter Quellcode

B3.3 Ergebnisse des zweiten Anwendungsbeispiels

B4 Das dritte Anwendungsbeispiel

B4.1 Unveränderter Quellcode des DLR

B4.2 Für den Testdurchlauf verwendeter Quellcode

B4.3 Ergebnisse der zufälligen Werteänderungen

B4.4 Ergebnisse vom Löschen eines Datenobjektes

B4.5 Ergebnisse des Belastungstests

Erklärung zur selbständigen Bearbeitung

Hiermit erkläre ich, dass ich die von mir eingereichte Masterarbeit „Entwurf und Implementierung eines Fuzzing Frameworks für das verteilte Tasking Framework des Deutschen Zentrums für Luft- und Raumfahrt“ selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Heide, 29.09.2017

(Dennis Pfau)