HAW HAMBURG

BACHELOR THESIS
Eduard Strehlau

Softwaretesten unter Berücksichtigung von nichtdeterministischen Verhalten

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Eduard Strehlau

Softwaretesten unter Berücksichtigung von nichtdeterministischen Verhalten

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuende Prüferin: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. Klaus-Peter Kossakowski

Eingereicht am: 04. März 2024

Eduard Strehlau

Thema der Arbeit

Softwaretesten unter Berücksichtigung von nichtdeterministischen Verhalten

Stichworte

Nichtdeterminismus, Automatisiertes Testen, Testabdeckung, Fehlerinjektion

Kurzzusammenfassung

Diese Arbeit untersucht die Hypothese: „Softwarezustände welche durch nichtdeterministische Effekte wie z. B. Eingabe/Ausgabe oder Nebenläufigkeit erreicht werden, werden nicht ausreichend getestet.“ Dabei wurde sich auf den Aspekt der Fehlerbehandlung fokussiert. Für diesen Aspekt ließ sich die Hypothese anhand von Testabdeckungsmetriken verifizieren. Um diese Metriken zu erfassen wurden eigene neuartige Werkzeuge entwickelt, die es möglich machen, sowohl Fehlerbehandlung als auch fehlende Fehlerbehandlung in C-Programmen zu lokalisieren. Zusätzlich wurden Werkzeuge entwickelt um diese mit Testabdeckungsmetriken zu verknüpfen. Nachdem die Hypothese belegt wurde, wurde nach Lösungsansätzen gesucht und der Stand der Technik wurde zwischen Forschung und Praxis verglichen. Ein effektives Verfahren zum automatisierten Testen von Fehlerbehandlung wurde in beiden Bereichen gefunden und eine Anwendung dieser Techniken im Allgemeinfall diskutiert.

Eduard Strehlau

Title of Thesis

Software testing in consideration of nondeterministic behaviour

Keywords

Nondeterminism, Automated Testing, Code Coverage, Fault Injection

Abstract

This thesis examines the hypothesis: „Software states which are achieved by nondeterministic effects such as input/output or concurrency are not sufficiently tested.“ The focus was on the aspect of error handling. For this aspect, the hypothesis was verified using test coverage metrics. In order to capture these metrics, novel tools were developed that make it possible to localize error handling, as well as missing error handling, in C programs. In addition, tools were developed to relate these to test coverage metrics. Once the hypothesis was proven, solutions were sought and the state of the art was compared between research and practice. An effective method for automated testing of error handling was found in both areas and an application of these techniques in the general case was discussed.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

clang **GNU Compiler Collections [40] (GCC)** kompatibler C-Compiler, bietet Unterstützung für Plugins über dynamische Bibliotheken sowie eine Bibliothek für Syntaxanalyse.

errno Globaler Fehlerindikator der C-Standardbibliothek und der **Portable Operating System Interface [34] (POSIX)**-Bibliothek [25].

error-analyzer Für diese Bachelorarbeit entwickeltes **clang**-Plugin zur Untersuchung von Fehlercodepfaden.

Interaktionspunkt Interaktionen zwischen System und Umgebung, definiert in Sektion 2.2.

rr Debugger mit „Record and Replay"-Funktion (Aufzeichnen und deterministische Wiedergabe des kompletten Programablaufs)[58][51].

SEI CERT C Standard für Entwicklung sicherer, robuster Systeme in der C-Programmiersprache [61].

stderr Mechanismus für Fehler- sowie Diagnostikausgabe der C-Standardbibliothek und der **POSIX**-Bibliothek [69].

stdout Standardausgabemechanismus der C-Standardbibliothek und der **POSIX**-Bibliothek [69].

Systeminstanz Zusammenschluss eines Programms und einer Ausführungsumgebung, definiert in Sektion 2.2.

Systemtrace Aufzeichnung einer **Systeminstanz**-Ausführung, definiert in Sektion 2.2.

Umgebungsaufruf Art von **Interaktionspunkten**, definiert in Sektion 2.2.

Umgebungsereignis Art von **Interaktionspunkten**, definiert in Sektion 2.2.

Umgebungsrückgabewert Rückgabewert eines **Umgebungsaufrufs**, definiert in Sektion 2.2.

Abkürzungen

ADFI Automatic Driver Fault Injection, Forschungsprojekt zur laufzeitinstrumentierten **Software Fault Injection [50] (SFI)** [15].

API Application Programming Interface, Schnitstellendefinition gegen die Anwendungen entwickelt werden können.

AST Abstract Syntax Tree, Baumrepräsentation von Quellcode.

CI Continuous Integration, stetige Integration des Gesamtsystems, nicht nur zwecks Release. Bspw. durch automatisierte Tests die bei jeder Änderung laufen.

CVE Common Vulnerabilities and Exposures [33], ein von der MITRE Corporation gepflegter Katalog von bekannten Schwachstellen. Industriestandard.

CWE Common Weakness Enumeration [21], eine von der MITRE Corporation gepflegte Liste häufiger Arten von Softwareschwachstellen.

GCC GNU Compiler Collections [40], Standard-C-Compiler vieler Projekte wie z.B. Linux.

IDL Interface Definition Language, Sprache/Formalismus zur Beschreibung einer Softwareschnittstelle.

JSON JavaScript Object Notation, sprachunabhängiges Datenbeschreibungsformat [6].

KEV Known Exploited Vulnerabilities [40], Katalog von **Common Vulnerabilities and Exposures [33] (CVE)** für die tatsächlich vorkommende Angriffe bekannt sind.

LCOV Linux Test Project GCOV extension, Zwischenformat für Testabdeckungsmetriken [43].

POSIX Portable Operating System Interface [34], ein von mehreren Betriebsystemen unterstützter Standard. Umfasst eine C-**Application Programming Interface (API)** für Bibliotheksfunktionen und Systemaufrufe. Beinhaltet die C-Standardbibliothek.

RAII Resource Acquisition Is Initialization, Resourcenverwaltung über Objektlebenszyklus [55].

SFI Software Fault Injection, Beobachtung eines Softwaresystems unter Injektion von Fehlern [50].

1 Exposé: Was soll untersucht werden?

Softwaretesten hat sich zu einem integralen Bestandteil der (Software-)Industrie entwickelt. In modernen Softwareprojekten ist Entwicklung automatisierter Tests oft eng verzahnt mit der Entwicklung der getesteten Software selbst. Diese Tests sollen sicherstellen, dass die entwickelte Software dem erwarteten Verhalten folgt und vor Regressionen bei Änderungen schützen. Zusätzlich soll sichergestellt werden, dass es im produktiven Einsatz der Software nicht zu unerwartetem Verhalten kommt. Um die Effektivität dieser Tests, ob Unit-Tests, Integrations-Tests oder Blackbox-End-To-End-Tests, zu bewerten, werden in vielen Projekten Whitebox-Testabdeckungsmetriken eingesetzt.

Diese Arbeit stellt die Vermutung auf, dass Softwarezustände, welche im produktiven Betrieb durch nichtdeterministische Effekte wie z. B. Eingabe/Ausgabe, Netzwerke, das Verstreichen von Zeit oder Nebenläufigkeit erreicht werden, nicht ausreichend getestet sind, also nur ein kleiner Teil dieser Effekte in Tests exerziert wird.

Als Untersuchungsansatz wird von dieser Arbeit zunächst Fehlerbehandlung betrachtet. Konkret wird hier die Hypothese aufgestellt: *Codepfade in denen Fehler der Laufzeitumgebung, wie z. B. Lese-/Schreibfehler, abgebrochene Systemaufrufe, nicht genügend Arbeitsspeicher etc., behandelt werden, werden weniger durch Softwaretests abgedeckt als Codepfade, in denen die Fehler nicht behandelt werden.* Diese Fehler werden durch Test-Suites nicht provoziert, und es wird oft nur der „Sunny-Case“ überprüft. Der Ansatz dieser Arbeit wurde gewählt, da bestehende Testabdeckungsmetriken, wie z. B. Zeilenabdeckung oder Branch-Abdeckung, diesen Sachverhalt aufzeigen sollten. Für Softwarezustände, welche durch nichtdeterministische Nebenläufigkeit erzeugt werden, fehlt es in der Industrie noch an konkreten Metriken, sodass die Vermutung dieser Arbeit für diese Fälle schwieriger zu belegen ist.

2 Modell und Nomenklatur

2.1 Einleitung

Um die Ausgangsfragen besser untersuchen zu können, wird zunächst versucht, klare Begrifflichkeiten zu definieren. Wo möglich, wird mit bereits bestehenden Definitionen gearbeitet. Leider gibt es in Referenzwerken [65] und Glossaren [37] wenige Begriffe, die in dieser Arbeit direkt verwendet werden können. Beispielsweise ist es schon bei dem Begriff „Softwaresystem“ nicht eindeutig, ob von laufender Installation des Systems gesprochen wird oder etwa von dem System „in Ruhe“. Als konkretes Beispiel dazu: Ist „WordPress“ selbst ein Softwaresystem, oder ist eine „WordPress“-Instanz, wie sie die Webseite der HAW-Hamburg bereitstellt, ein Softwaresystem? In diesem Abschnitt wird eine konsistente Nutzung der Begriffe für diese Arbeit eingeführt und es werden gegebenenfalls neue Begriffe, wo nötig, definiert.

Zusätzlich wird versucht, auch ein in diesem Kontext praktisch anwendbares Modell für Determinismus von Software zu finden. Auch hier gibt es in der Literatur viele Definitionen und Ansätze wie z. B. Memorymodelle [47][11] oder Nebenläufigkeitsmodelle [32][54]. In erster Linie ist aber ein Modell gesucht, welches dabei hilft, die Ausgangsfragen dieser Arbeit zu untersuchen und einen Lösungsansatz zu entwickeln. Entwicklung oder Findung eines Modells ist in dieser Hinsicht zweitrangig in dieser Arbeit. Dennoch wird aber validiert, ob das Modell und die damit verbundenen Aussagen auf Softwaresysteme anwendbar sind. Dafür wird das Modell mit bestehenden Softwarelösungen, welche eng im Zusammenhang mit Determinismus stehen (Debugger, Schnapsschussreplikation etc.), verglichen.

2.2 Kontextabgrenzung

Zunächst wird die Problemstellung auf Softwaresysteme, welche ein einzelnes Programm umfassen, beschränkt. Als **Systeminstanz** wird in dieser Arbeit ein Zusammenschluss zwischen einer konkreten (Ausführungs-)Umgebung und einem Programm bezeichnet. Dabei kann es sich bei der Umgebung beispielsweise um einen Server handeln, auf dem das Programm installiert ist, oder um einen Test-Harnisch, in dem das Programm getestet wird. Die Sequenz der beobachtbaren Ereignisse einer konkreten Ausführung der **Systeminstanz** sowie der Zustand des Systems vor dem ersten Ereignis wird als **Systemtrace** bezeichnet. Ein **Systemtrace** ist zeitlich begrenzt, eine Begrenzung kann beispielsweise eine komplette Ausführung des Programms beinhalten (Batchprozess), ein Request/Response-Paar (Interaktives System) oder einen Test-Case etc. Dabei befindet sich das System zu Anfang eines Traces stets in einem Startzustand. Der Umgebung wird kein Zustand zugeordnet, stattdessen wird die Umgebung rein über die Interaktion mit dem System charakterisiert. Diese Interaktionen, die Zeitpunkte in einem Trace widerspiegeln, werden **Interaktionspunkte** genannt. Kernannahme ist, dass sich das System zwischen **Interaktionspunkten** komplett deterministisch verhält. D. h. die Zustandsübergänge des Systems werden einzig und allein durch die Umgebung an **Interaktionspunkten** sowie durch den Startzustand bestimmt. Sofern der Startzustand des Systems sowie das Verhalten der Umgebung an **Interaktionspunkten** eines Traces bekannt ist, ist es möglich, jeden Zustand, den das System in einem Trace einnimmt, zu rekonstruieren.

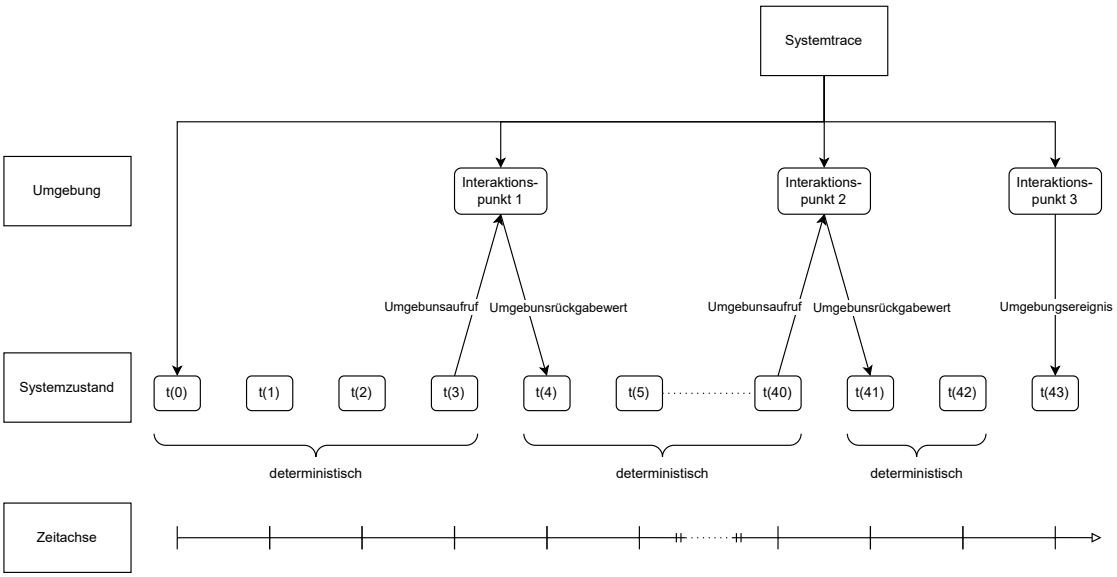

Abbildung 2.1: Visualisierung des hier beschriebenen Modells

Dieses Modell erlaubt also „Record and Replay“ einer Ausführung einer **Systeminstanz**.

In der Praxis wird dieses Modell beispielsweise von dem Debugger **rr [58][51]** umgesetzt, welcher in der Lage ist, produktive Linuxanwendungen wie z. B. Chrome und Firefox auszuführen und zu instrumentieren. Daraus lässt sich schließen, dass das Modell sich auf unmodifizierte, produktive Linuxanwendungen anwenden lässt. Die Kernannahme über das deterministische Verhalten von Programmen trifft also auch bei großen, komplexen Softwareprojekten zu.

Im Vergleich zu **rr**, bei dem dieses Modell auf einer sehr tiefen Ebene umgesetzt ist (Prozessorzustand), werden in dieser Arbeit einige weitere Annahmen gemacht.

Dazu müssen zunächst **Interaktionspunkte** weiter klassifiziert werden. **Interaktionspunkte**, welche explizit im Programm angegeben sind (durch einen Funktions- oder Methodenaufruf), werden **Umgebungsaufrufe** genannt. Zusätzlich existieren aber noch **Interaktionspunkte**, welche implizit jederzeit zwischen zwei **Umgebungsaufrufe** auftreten können. Diese werden von **rr** als „Asynchronous Events“ [51] bezeichnet und hier **Umgebungsereignisse** genannt.

Dabei handelt es sich konkret um **POSIX**-Signale und präemptive Kontextwechsel [51]. Entscheidend ist, dass diese **Umgebungsereignisse** auf Ebene der Maschinensprache

stattfinden, da sie das Programm zu jedem Zeitpunkt unterbrechen können und nicht nur zu Zeitpunkten, die ein klar definiertes Analogon im Quellcode haben. Für **rr** stellt dies kein Problem dar, denn Ziel des Projekts ist eine „Record and Replay"-Ausführung von Maschinensprache. Für diese Arbeit hingegen stellt dies jedoch ein Problem dar, denn es wird angestrebt, auf der Quellcodeebene zu operieren.

Es werden also zum Ausschluss von **Umgebungsereignissen** zusätzlich noch folgende Annahmen gemacht:

1. Ausführung eines Programms ist singlethreaded, **Umgebungsereignisse** durch präemptive Kontextwechsel fallen also weg. Diese Annahme hält für Programme, die kein Multithreading nutzen, sowie auch oft für ein Subset der Test-Suite von Programmen, die Multithreading nutzen. Gegebenenfalls lässt sich im weiteren Verlauf der Arbeit ein Multithreading-Modell finden, welches rein mit **Umgebungsaufrufen** arbeitet. Eine mögliche Skizze dafür unter der Annahme, dass ein Programm frei von Data-Races ist (nötig für C++ [47], Java [11] sowie pthreads [30] Memorymodell), sieht folgendermaßen aus: Threads interagieren nur über explizite Synchronisationspunkte (Locks, Conditionvariables, etc.), dabei handelt es sich um **Umgebungsaufrufe**. Ist ein Synchronisationspunkt erreicht, kann die Umgebung entscheiden, welcher Thread deterministisch bis zum nächsten Synchronisationspunkt ausgeführt werden soll. Unter pthreads [30] handelt es sich bei den Synchronisationspunkten um Funktionsaufrufe [30], bei den anderen Speichermodellen kommen noch atomare Datentypen hinzu, welche aber auch nur durch Funktions-/Methodenaufrufe modifiziert werden können.
2. Signale beeinflussen nur das Ergebnis von **Umgebungsaufrufen** und haben selbst keinen Effekt auf das Systemverhalten. (Das System verhält sich zwischen zwei **Umgebungsaufrufen**, zwischen denen ein Signal liegt, stets deterministisch.) Unter **POSIX** lässt sich diese Annahme machen. Signale werden in der Praxis oft durch einen Adapter behandelt, welcher diese in Ergebnisse von **Umgebungsaufrufen** umwandelt und ansonsten keinen Effekt auf die Ausführung hat [5][3][64]. Für die verbreitete Nutzung dieser Annahme spricht, dass z. B. Linux auch eine native **API** für umgebungsaufrufbasierte Signale bereitstellt [63], sowie dass Sprachimplementierungen die gleiche oder ähnliche Strategien zur Signalbehandlung nutzen [42][17].

Kurz zusammengefasst wird also angenommen, dass alle **Interaktionspunkte** explizit im Programm (durch einen Funktions- oder Methodenaufruf, hier **Umgebungsaufruf**) angegeben sind.

Die vom Programm wahrnehmbaren Effekte eines **Umgebungsaufrufs** werden **Umgebungsrückgabewerte**, oder wenn anhand von Kontext eindeutig, Rückgabewerte genannt. Dabei handelt es sich nicht nur um explizite Rückgabewerte von Funktionen/-Methoden, sondern z. B. auch um das Setzen globaler Variablen wie **errno [25]**. Es wird angenommen, dass **Umgebungsrückgabewerte** sich anhand der Dokumentation des **Umgebungsaufrufs** in Fehlerwerte und Nicht-Fehlerwerte klassifizieren lassen. Die Fehlerwerte werden als Umgebungsfehler bezeichnet.

Die Ausgangsfrage kann jetzt anhand dieser Definitionen formuliert werden:

Werden Programmpfade, welche aufgrund von Umgebungsfehlern durchlaufen werden, weniger durch Tests abgedeckt als die durchschnittliche Zeilenabdeckung für das gesamte Programm?

3 Vorgehen zur Feststellung des Ist-Zustands

3.1 Einleitung

Nachdem in der vorherigen Sektion ein Modell und eine Nomenklatur zur Untersuchung der Ausgangsfrage gefunden wurde, soll in dieser Sektion die Fragestellung beantwortet werden. Dazu wird zunächst versucht, Kriterien zu finden, welche Art von Open-Source-Softwareprojekten sich gut in Hinsicht der Frage untersuchen lassen. Schließlich wird versucht, die Frage für diese Projekte zu beantworten und daraus eine allgemeingültige Antwort herzuleiten.

3.2 Umgebungsaufrufe aus Sicht des Programms

Umgebungsaufrufe haben in den meisten Programmiersprachen keine besondere Semantik, sondern sind durch einfache Funktions- oder Methodenaufrufe, verfügbar über eine Laufzeitbibliothek, abgebildet. Für Fehlerbehandlung jener lassen sich Programmiersprachen generell in zwei Lager unterteilen, Sprachen mit und ohne Ausnahmebehandlung:

1. Programmiersprachen ohne Ausnahmebehandlung (C [36], Go [72], Rust [59] etc.)

   Fehlerfälle werden über Rückgabewerte signalisiert, im Fehlerfall gibt es keinen besonderen Kontrollfluss. Das Signalisieren des Fehlers muss explizit im Programmcode bis zum Ort der Fehlerbehandlung erfolgen.

2. Programmiersprachen mit Ausnahmebehandlung (C++ [66], Java [38], C# [8] etc.)

   Im Fehlerfall bricht der normale Programmfluss ab. Anstatt einen Rückgabewert zu liefern und den Programmcode sequentiell abzuarbeiten, springt die Ausführung zu

der im Callstack nächsten Fehlerbehandlung und bricht alle dazwischenliegenden Funktionsaufrufe ab [74]. Es existiert voreingestellt eine globale Ausnahmebehandlung, welche im Fehlerfall den Programmfluss abbricht.

In dieser Arbeit liegt der Fokus auf Sprachen ohne Ausnahmebehandlung, dies hat folgende Gründe:

1. Wird Fehlerbehandlung vergessen, geht ein Programm ohne Ausnahmebehandlung in einen nicht definierten Zustand über. Programme mit Ausnahmebehandlung werden standardmäßig beendet. In einem solchen Zustand kann es zu einer Vielzahl von ungewolltem Verhalten kommen, welche über die bloße Verfügbarkeit der Software hinausgeht: beispielsweise Sicherheitsprobleme, Korruption von persistenten Daten etc.
2. Mit Blick auf die Testabdeckung ist, da es sich um einen nicht-lokalen Effekt handelt, nicht klar, wie Ausnahmen gehandhabt werden sollen. In Sprachen ohne Ausnahmebehandlung wird hingegen explizit lokal behandelt oder weitergegeben, was dann auch immer in einer Testabdeckungsmetrik auftaucht.
3. Die explizite Handhabung von Fehlerfällen setzt gegenüber Ausnahmebehandlung eine größere Menge von Programmcode voraus, dieser bietet mehr Fläche für Programmierfehler.

Zusätzlich ist abzusehen, dass Werkzeuge zur Analyse und Lösungsansätze sprach- und laufzeitumgebungsspezifisch sind. Daher fokussiert sich diese Arbeit nur auf eine Sprache. Aus den drei genannten Alternativen (C, Go und Rust) wurde C ausgewählt. Das hat vor allem den Grund, dass C im Gegensatz zu Rust und Go kein automatisches oder laufzeitunterstütztes Ressourcenmanagement unterstützt. D. h. im Fehlerfall müssen alle von einer Funktion reservierten Ressourcen wie Speicher, Dateien, Verbindungen etc. manuell wieder freigegeben werden. Rust und Go unterstützen beide automatisches Speichermanagement, Rust unterstützt Destruktoren/**Resource Acquisition Is Initialization [55][56] (RAII)** für automatisches Ressourcenmanagement und Go bietet Laufzeitunterstützung für Ressourcenmanagement mithilfe des „defer“ Keywords [73]. Hinzu kommt, dass es bei Fehlern im Speichermanagement zu sicherheitsrelevanten Schwachstellen und Verfügbarkeitsproblemen (Speicherlecks, wachsender Speicherverbrauch) kommen kann. Wird z. B. Speicher zweimal freigegeben „double-free“ [22], kann es zu Schwachstellen kommen, welche einem Angreifer erlauben, eigenen Code auszuführen.

Für C ist es also allein schon aus einer Sicherheitsperspektive (CWE-787, CWE-416, CWE-125, CWE-476, CWE-122 in **Common Weakness Enumeration [21] (CWE)** Top 25 [20] und **CWE Known Exploited Vulnerabilities [40] (KEV)** Top 10 [19]) kritisch, dass Codepfade, welche bei Umgebungsfehlern durchlaufen werden, Speicher korrekt verwalten. Dahingehend ist es auch wichtig, dass diese getestet werden, um fundierte Aussagen über diese Programmabschnitte treffen zu können.

3.3 Projektauswahl

Es wurden jeweils von GitHub, einer der größten Kollaborationsplattformen für Open-Source-Projekte nach Nutzern [23], die Top-100 Projekte nach „Sternen“ für die Sprache C aufgelistet. (Auf GitHub können Nutzer Projekte mit Sternen markieren, um ihr Interesse an dem Projekt zu bekunden, vergleichbar mit „Abonnier“-Funktionen auf anderen sozialen Plattformen.) Eine kleine Untermenge dieser Projekte wurde für die Untersuchung in dieser Arbeit ausgewählt. Dabei war die Existenz einer Test-Suite, welche untersucht werden kann, sowie Unterstützung bzw. einfache Integration von Testabdeckungsmetriken, das Hauptkriterium [77]. Die von den Projekten genutzten Test-Suites reichen dabei von Unit-Tests bis zu End-To-End-Tests. Die zuvor genannten Probleme von C (Memory-Safety, manuelles Memory-Management) wurden von vielen Test-Suites mitgetestet. Dazu werden Tools wie Valgrind [76] und Compiler-Sanitizer [2] von den Projekten eingesetzt, um sicherzustellen, dass bei der Ausführung der Tests keine Speicherkorruption oder Lecks auftreten.

Projekt	Beschreibung	GitHub-Ranking nach Sprache
cURL	Datentransfer-Bibliothek/Kommandozeilenprogramm, genutzt in 20 Milliarden Installationen	8
git	Versionsverwaltung, Industriestandard, genutzt von allen hier getesteten Projekten	5
jq	Kommandozeilenprogramm zur **JSON**-Datenverarbeitung	12
redis	Cache/Datenstruktur-Server	3

Tabelle 3.1: Informationen zu den untersuchten Projekten

3.4 Initiale Anamnese

Zunächst wurden alle Test-Suites der Projekte ausgeführt und die Testabdeckungsrate erfasst. Dabei ist hier die Abdeckung aufgrund von Limitationen der zugrundeliegenden Werkzeuge (**Linux Test Project GCOV extension [43] (LCOV)**) als Zeilenabdeckung erfasst. Für die betrachteten Projekte ist die durchschnittliche Zeilenabdeckungsrate 78,40% mit einem Bereich von 67,71% bis 86,59%. Die Qualität der Tests wird in dieser Arbeit nicht betrachtet, vielmehr geht es um die nicht getesteten Teile der Projekte. Denn anhand der ungetesteten Teile lässt sich (ohne manuelle Inspektion) keine Aussage über das Verhalten und Defekte des Programms treffen, und Regressionen lassen sich nicht automatisch feststellen.

Projekt	Version	Zeilenabdeckung			Funktionsabdeckung		
		n	von	Prozent	n	von	Prozent
cURL	8.4.0	25018	31251	80,06%	1376	1470	93,61%
git	2.42.0	119027	137456	86,59%	8539	8998	94,90%
jq	1.7	7408	9347	79,26%	550	664	82,83%
redis	7.2.1	41546	61356	67,71%	2572	3782	68,01%

Tabelle 3.2: Testabdeckung untersuchter Projekte

Zusätzlich zu den Test-Suites wurden einige stichprobenartige Prüfungen der Abdeckung durchgeführt. Diese deuten auf die ursprüngliche Hypothese hin, Fehlerbehandlung sei oft nicht in Testabdeckung eingeschlossen (rot hervorgehobene Zeilen sind in den folgenden Auszügen nicht Abgedeckt). Um diese Hypothese genauer und nicht nur stichprobenartig zu überprüfen, werden im folgenden Abschnitt Anforderungen gesammelt, die nötig für das Messen der Zeilenabdeckungsrate in Bezug zur Umgebungsfehlerbehandlung sind.

```
    139            :
    140   44005760 : void* jv_mem_alloc(size_t sz) {
    141   44005760 :   void* p = malloc(sz);
    142   44005760 :   if (!p) {
    143          0 :     memory_exhausted();
    144            :   }
    145   44005760 :   return p;
    146            : }
    147            :
    148       2954 : void* jv_mem_alloc_unguarded(size_t sz) {
    149       2954 :   return malloc(sz);
    150            : }
    151            :
    152      75957 : void* jv_mem_calloc(size_t nemb, size_t sz) {
    153      75957 :   void* p = calloc(nemb, sz);
    154      75957 :   if (!p) {
    155          0 :     memory_exhausted();
    156            :   }
    157      75957 :   return p;
    158            : }
    159            :
    160          0 : void* jv_mem_calloc_unguarded(size_t nemb, size_t sz) {
    161          0 :   return calloc(nemb, sz);
    162            : }
    163            :
    164       2723 : char* jv_mem_strdup(const char *s) {
    165       2723 :   char *p = strdup(s);
    166       2723 :   if (!p) {
    167          0 :     memory_exhausted();
    168            :   }
    169       2723 :   return p;
    170            : }
    171            :
    172          0 : char* jv_mem_strdup_unguarded(const char *s) {
    173          0 :   return strdup(s);
    174            : }
    175            :
    176   74766870 : void jv_mem_free(void* p) {
    177   74766870 :   free(p);
    178   74766870 : }
    179            :
    180     939409 : void* jv_mem_realloc(void* p, size_t sz) {
    181     939409 :   p = realloc(p, sz);
    182     939409 :   if (!p) {
    183          0 :     memory_exhausted();
    184            :   }
    185     939409 :   return p;
    186            : }
```

Abbildung 3.1: jq/src/jv_alloc.c

```
        :
  343 : jv jv_load_file(const char* filename, int raw) {
  343 :   struct stat sb;
  343 :   int fd = open(filename, O_RDONLY);
  343 :   if (fd == -1) {
    0 :     return jv_invalid_with_msg(jv_string_fmt("Could not open %s: %s",
        :                                              filename,
    0 :                                              strerror(errno)));
        :   }
  343 :   if (fstat(fd, &sb) == -1 || S_ISDIR(sb.st_mode)) {
    7 :     close(fd);
    7 :     return jv_invalid_with_msg(jv_string_fmt("Could not open %s: %s",
        :                                              filename,
        :                                              "It's a directory"));
        :   }
  336 :   FILE* file = fdopen(fd, "r");
  336 :   struct jv_parser* parser = NULL;
  336 :   jv data;
  336 :   if (!file) {
    0 :     close(fd);
    0 :     return jv_invalid_with_msg(jv_string_fmt("Could not open %s: %s",
        :                                              filename,
    0 :                                              strerror(errno)));
        :   }
  336 :   if (raw) {
  287 :     data = jv_string("");
        :   } else {
   49 :     data = jv_array();
   49 :     parser = jv_parser_new(0);
        :   }
        :
```

Abbildung 3.2: jq/src/jv_file.c

```
        :
    4 : static enum bisect_error bisect_replay(struct bisect_terms *terms, const char *filename)
      : {
    4 :         FILE *fp = NULL;
    4 :         enum bisect_error res = BISECT_OK;
    4 :         struct strbuf line = STRBUF_INIT;
      :
    4 :         if (is_empty_or_missing_file(filename))
    0 :                 return error(_("cannot read file '%s' for replaying"), filename);
      :
    4 :         if (bisect_reset(NULL))
    0 :                 return BISECT_FAILED;
      :
    4 :         fp = fopen(filename, "r");
    4 :         if (!fp)
    0 :                 return BISECT_FAILED;
      :
   48 :         while ((strbuf_getline(&line, fp) != EOF) && !res)
   44 :                 res = process_replay_line(terms, &line);
      :
    4 :         strbuf_release(&line);
    4 :         fclose(fp);
      :
    4 :         if (res)
    0 :                 return BISECT_FAILED;
      :
    4 :         return bisect_auto_next(terms, NULL);
      : }
      :
```

Abbildung 3.3: git/builtin/bisect.c

```
868           : /**
869           :  * A split_patches_conv() callback that converts a mercurial patch to a RFC2822
870           :  * message suitable for parsing with git-mailinfo.
871           :  */
872         2 : static int hg_patch_to_mail(FILE *out, FILE *in, int keep_cr UNUSED)
873           : {
874         2 :         struct strbuf sb = STRBUF_INIT;
875         2 :         int rc = 0;
876           :
877        14 :         while (!strbuf_getline_lf(&sb, in)) {
878           :                 const char *str;
879           :
880        14 :                 if (skip_prefix(sb.buf, "# User ", &str))
881         2 :                         fprintf(out, "From: %s\n", str);
882        12 :                 else if (skip_prefix(sb.buf, "# Date ", &str)) {
883           :                         timestamp_t timestamp;
884           :                         long tz, tz2;
885           :                         char *end;
886           :
887         2 :                         errno = 0;
888         2 :                         timestamp = parse_timestamp(str, &end, 10);
889         2 :                         if (errno) {
890         0 :                                 rc = error(_("invalid timestamp"));
891         0 :                                 goto exit;
892           :                         }
893           :
894         2 :                         if (!skip_prefix(end, " ", &str)) {
895         0 :                                 rc = error(_("invalid Date line"));
896         0 :                                 goto exit;
897           :                         }
898           :
899         2 :                         errno = 0;
900         2 :                         tz = strtol(str, &end, 10);
901         2 :                         if (errno) {
902         0 :                                 rc = error(_("invalid timezone offset"));
903         0 :                                 goto exit;
904           :                         }
905           :
906         2 :                         if (*end) {
907         0 :                                 rc = error(_("invalid Date line"));
908         0 :                                 goto exit;
909           :                         }
```

Abbildung 3.4: git/builtin/am.c

```
84            : /*
85            :  * curl_slist_append() appends a string to the linked list. It always returns
86            :  * the address of the first record, so that you can use this function as an
87            :  * initialization function as well as an append function. If you find this
88            :  * bothersome, then simply create a separate _init function and call it
89            :  * appropriately from within the program.
90            :  */
91      1067 : struct curl_slist *curl_slist_append(struct curl_slist *list,
92            :                                      const char *data)
93            : {
94      1067 :   char *dupdata = strdup(data);
95            :
96      1067 :   if(!dupdata)
97         0 :     return NULL;
98            :
99      1067 :   list = Curl_slist_append_nodup(list, dupdata);
100     1067 :   if(!list)
101        0 :     free(dupdata);
102           :
103     1067 :   return list;
104           : }
105           :
```

Abbildung 3.5: curl/lib/slist.c

```
127        238 : static int write_in_file(const char *path, const char *mode, const char *format, va_list args)
128            : {
129        238 :         FILE *fp = NULL;
130        238 :         int res = 0;
131            :
132        238 :         if (strcmp(mode, "w") && strcmp(mode, "a"))
133          0 :                 BUG("write-in-file does not support '%s' mode", mode);
134        238 :         fp = fopen(path, mode);
135        238 :         if (!fp)
136          0 :                 return error_errno(_("cannot open file '%s' in mode '%s'"), path, mode);
137        238 :         res = vfprintf(fp, format, args);
138            :
139        238 :         if (res < 0) {
140          0 :                 int saved_errno = errno;
141          0 :                 fclose(fp);
142          0 :                 errno = saved_errno;
143          0 :                 return error_errno(_("could not write to file '%s'"), path);
144            :         }
145            :
146        238 :         return fclose(fp);
147            : }
```

Abbildung 3.6: git/builtin/bisect.c

```
  49         43 : CURLcode Curl_fopen(struct Curl_easy *data, const char *filename,
  50            :                     FILE **fh, char **tempname)
  51            : {
  52         43 :   CURLcode result = CURLE_WRITE_ERROR;
  53            :   unsigned char randsuffix[9];
  54         43 :   char *tempstore = NULL;
  55            :   struct_stat sb;
  56         43 :   int fd = -1;
  57         43 :   *tempname = NULL;
  58            :
  59         43 :   *fh = fopen(filename, FOPEN_WRITETEXT);
  60         43 :   if(!*fh)
  61          0 :     goto fail;
  62         43 :   if(fstat(fileno(*fh), &sb) == -1 || !S_ISREG(sb.st_mode))
  63          0 :     return CURLE_OK;
  64         43 :   fclose(*fh);
  65         43 :   *fh = NULL;
  66            :
  67         43 :   result = Curl_rand_alnum(data, randsuffix, sizeof(randsuffix));
  68         43 :   if(result)
  69          0 :     goto fail;
  70            :
  71         43 :   tempstore = aprintf("%s.%s.tmp", filename, randsuffix);
  72         43 :   if(!tempstore) {
  73          0 :     result = CURLE_OUT_OF_MEMORY;
  74          0 :     goto fail;
  75            :   }
  76            :
  77         43 :   result = CURLE_WRITE_ERROR;
  78         43 :   fd = open(tempstore, O_WRONLY | O_CREAT | O_EXCL, 0600);
  79         43 :   if(fd == -1)
  80          0 :     goto fail;
  81            :
  82            : #ifdef HAVE_FCHMOD
  83            :   {
  84            :     struct_stat nsb;
  85         43 :     if((fstat(fd, &nsb) != -1) &&
  86         43 :        (nsb.st_uid == sb.st_uid) && (nsb.st_gid == sb.st_gid)) {
  87            :       /* if the user and group are the same, clone the original mode */
  88         43 :       if(fchmod(fd, (mode_t)sb.st_mode) == -1)
  89          0 :         goto fail;
  90            :     }
  91            :   }
  92            : #endif
  93            :
  94         43 :   *fh = fdopen(fd, FOPEN_WRITETEXT);
  95         43 :   if(!*fh)
  96          0 :     goto fail;
  97            :
  98         43 :   *tempname = tempstore;
  99         43 :   return CURLE_OK;
 100            :
 101          0 : fail:
 102          0 :   if(fd != -1) {
 103          0 :     close(fd);
 104          0 :     unlink(tempstore);
 105            :   }
 106            :
 107          0 :   free(tempstore);
 108            :
 109          0 :   return result;
 110            : }
 111            :
```

Abbildung 3.7: curl/lib/fopen.c

3.5 Feststellung der Anforderungen und Konzeption

Es soll untersucht werden, wie die Testabdeckung von Kontrollfluss bei Umgebungsfehlern für die gewählten C-Programme aussieht. Dafür muss zunächst entschieden werden, welche Funktionsaufrufe als **Umgebungsaufrufe**, welche nichtdeterministisches Verhalten und Fehler der Umgebung in den Programmen einführen können, betrachtet werden. Für Programme ist die Schnittstelle des Betriebssystems die Grenze zwischen Programm und Umgebung und definitionsgemäß alles, was über diese Grenze verläuft, ein **Interaktionspunkt**. Diese Schnittstelle kann je nach Plattform von direkten Systemaufrufen (Linux [1]) zu Funktionsaufrufen in dynamischen Bibliotheken (Windows [78], macOS [70] etc.) variieren. In dieser Arbeit wird die **POSIX**-C-Bibliothek als die primäre Schnittstelle zwischen Programm und Umgebung und daraus folgend Funktionsaufrufe von **POSIX**-Bibliotheksfunktionen als **Umgebungsaufrufe** behandelt. Dies hat vor allem drei Gründe:

1. Die untersuchten Programme (und allgemein viele in C geschriebene Programme, eingebettete Software ausgenommen) zielen auf eine **POSIX**-Laufzeitumgebung ab.
2. **POSIX** ist eine Obermenge der C-Standardbibliothek, die von allen C-Programmen (außer free-standing [14]) verwendet wird. Wenn **POSIX** abgedeckt wird, wird also automatisch die C-Standardbibliothek mit abgedeckt. Damit werden also auch Plattformen wie z. B. Windows teilweise mit abgedeckt, die nicht **POSIX**-konform sind.
3. **POSIX** beinhaltet **APIs** für Speichermanagement und Stringhandling (Format-Specifier etc.), eines der fehleranfälligsten und sicherheitsrelevantesten Themen in C (CWE-787, CWE-416, CWE-125, CWE-476, CWE-122 in **CWE** Top 25 [20] und **CWE KEV** Top 10 [19]).

Von Projekten, in dieser Betrachtung beispielsweise „cURL“, werden manchmal Wrapper-Funktionen/Abstraktionen um die zugrundeliegende Systemschnittstelle gebaut. Bei „cURL“ findet sich z. B. `Curl_cmalloc` als Ersatz für die **POSIX**-Funktion `malloc`. Würde nur nach dem **POSIX**-Aufruf gesucht werden, fände sich lediglich ein Aufruf in der Implementierung von `Curl_cmalloc`, nicht jedoch alle Nutzungen von `Curl_cmalloc`, welches hier semantisch `malloc` entspricht. Daher ist bei Bedarf eine Aufnahme von zusätzlichen Funktionsaufrufen in die Liste der **Umgebungsaufrufe** wünschenswert.

Das Ziel zur Validierung der Hypothese ist es also, alle **POSIX-Umgebungsaufrufe** und ggf. projektspezifische **Umgebungsaufrufe** in den untersuchten Projekten zu finden und zu überprüfen, ob deren Fehlerfälle von der Test-Suite abgedeckt sind. Leider wurde in der Recherche dieser Arbeit keine Software gefunden, welche diese Anforderungen vollständig erfüllt. Eine Übersicht zu Software, welche die Behandlung von Rückgabewerten analysiert, findet sich beispielsweise in der zum **SEI CERT C [61]**-Standard gehörenden Wiki [24][26].

Daher musste im Zuge dieser Arbeit eigene Software entwickelt werden, um diesem Ziel gerecht zu werden. Dafür wird zunächst das Ziel in weitere Anforderungen aufgeschlüsselt, welche zur Erfüllung jenes notwendig sind.

Als erste notwendige Anforderung kann abgeleitet werden: *Es müssen alle **POSIX**-Funktionsaufrufe sowie ggf. projektabhängige Funktionsaufrufe in dem Quelltext der untersuchten Projekte gefunden werden.* Damit bei der Aufnahme weiterer Projekte oder bei dem produktiven Einsatz von im Laufe dieser Arbeit entwickelten Software keine Softwareanpassungen erfolgen müssen, lässt sich diese Anforderung als folgendermaßen generalisieren: *Es muss eine konfigurierbare Liste von Funktionsaufrufen im Quelltext von zu untersuchenden C-Programmen gefunden werden.*

Sind all diese Funktionsaufrufe gefunden, soll entschieden werden, ob der Kontrollfluss, welcher im Fehlerfall eingeschlagen wird, von der Test-Suite des Programms abgedeckt ist. Dafür ist die Information nötig, ob es sich bei **Umgebungsrückgabewerten** um Umgebungsfehler handelt. Es ergibt sich also die zweite Anforderung: *Für alle von den Projekten genutzten **Umgebungsaufrufe** muss eine Klassifizierung der **Umgebungsrückgabewerte** in Fehler und Nicht-Fehler bereitgestellt werden können.*

Um den Kontrollfluss im Fehlerfall zu finden, muss zunächst der erste Punkt gefunden werden, an dem der **Umgebungsrückgabewert** überhaupt Einfluss auf den Kontrollfluss haben kann. Dieser Punkt wird „Konsument“ des Wertes genannt. Im Weiteren muss ermittelt werden, wie dann eine Kontrollflussentscheidung im Fehlerfall ausfällt. Erst wenn dies passiert ist, kann überprüft werden, ob dieser Kontrollfluss von Tests abgedeckt ist. Diese beiden Prozesse werden jeweils „Konsumentenlokalisierung“ und „Kontrollflussauswertung“ genannt.

3.5.1 Konsumentenlokalisierung

Der Konsument eines Rückgabewertes lässt sich in eine von mehreren Kategorien einordnen, für die dann separat Strategien zur Lokalisierung entwickelt werden können.

1. **Umgebungsrückgabewert** wird nie benutzt, d. h. man weiß, dass dieser definitiv keinen Einfluss auf den Kontrollfluss hat. Dies deutet auf eine vergessene Fehlerbehandlung hin. Beispiel:

```
1  // Rückgabewert ist "false" bei Auftritt eines Umgebungsfehlers.
2  bool readFileFalseOnError(FILE *file, char *output, size_t size);
3
4  void main(void) {
5   FILE *file;
6   char buffer[20];
7   // Hier fehlt die Fehlerbehandlung.
8   // Der Rückgabewert von readFileFalseOnError wird ignoriert.
9   // Für den Fehlerfall kann es also keinen gesonderten Programmfluss
10  // geben.
11  readFileFalseOnError(file, buffer, sizeof(buffer));
12 }
```

2. **Umgebungsrückgabewert** ist direkt Parameter an eine andere Funktion oder wird direkt zurückgegeben. Beispiel:

```
1  // Rückgabewert ist "false" bei Auftritt eines Umgebungsfehlers.
2  bool readFileFalseOnError(FILE *file, char *output, size_t size);
3  // Rückgabewert ist "false" bei Auftritt eines Umgebungsfehlers.
4  bool writeFileFalseOnError(FILE *file, char *output, size_t size);
5  // Bricht die Ausführung des Programms ab
6  // wenn der Parameter "ok" "false" ist.
7  void abortOnError(bool ok);
8
9  bool readOrWriteFalseOnError(FILE *file, char *buffer,
10                               size_t bufferSize, bool read) {
11  // Fehlerwert wird direkt zurückgegeben,
12  // d. h. der Aufrufer von readOrWriteFalseOnError
13  // ist für die Behandlung verantwortlich.
14  if(read) {
15    return readFileFalseOnError(file, buffer, bufferSize);
16  } else {
17    return writeFileFalseOnError(file, buffer, bufferSize);
18  }
19 }
```

```
20
21 void main(void) {
22  FILE *file;
23  char buffer[20];
24  // Fehler (Rückgabewert von readFileFalseOnError),
25  // wird an Funktion weitergeben welche diesen behandelt
26  // (hier durch Abbruch des Programms).
27  abortOnError(readFileFalseOnError(file, buffer, sizeof(buffer)));
28  // Fehler wird direkt an die Wrapper-Funktion zurückgegeben
29  // und sollte hier behandelt werden.
30  // Fehlerbehandlung fehlt hier aber.
31  readOrWriteFalseOnError(file, buffer, sizeof(buffer), true);
32 }
```

3. **Umgebungsrückgabewert** ist Teil einer Kontrollflussanweisung. Beispiel:

```
1 // Rückgabewert ist "false" bei Auftritt eines Umgebungsfehlers.
2 bool readFileFalseOnError(FILE *file, char *output, size_t size);
3
4 void main(void) {
5   FILE *file;
6   char buffer[20];
7   if(!readFileFalseOnError(file, buffer, sizeof(buffer))) {
8    // Hier würde der Fehler behandelt.
9    printf("Fehler ist behandelt\n");
10   return;
11  }
12  printf("Alles ok!\n");
13 }
```

4. **Umgebungsrückgabewert** wird in einer Variable zwischengespeichert. Die Variable übernimmt die Rolle des ursprünglichen Aufrufs. Es muss also für die Variable der nächste Konsument gefunden werden. Beispiel:

```
1 // Rückgabewert ist "false" bei Auftritt eines Umgebungsfehlers.
2 bool readFileFalseOnError(FILE *file, char *output, size_t size);
3
4 void main(void) {
5   FILE *file;
6   char buffer[20];
7   // Nutzung der Variable "success" ist ab hier equivalent
8   // zum Aufruf von readFileFalseOnError
9   // was Fehlerbehandlung angeht.
10  bool success = readFileFalseOnError(file, buffer, sizeof(buffer));
```

```
11   if(!success){
12    // Hier würde der Fehler behandelt.
13    printf("Fehler ist behandelt\n");
14    return;
15   }
16   printf("Alles ok!\n");
17 }
```

Das Finden des initialen Konsumenten des Umgebungsfehlers gestaltet sich relativ einfach. Dafür muss nur rekursiv jeweils der Elternausdruck des **Umgebungsaufrufs** durchlaufen werden, bis einer von den gelisteten Fällen erreicht ist, also Zuweisungsausdruck (4. Fall), Rückgabeausdruck oder Parameterausdruck (2. Fall) oder Kontrollflussausdruck (3. Fall). Wird keiner dieser Ausdrücke gefunden, bevor ein Funktionsrumpfausdruck erreicht ist, gibt es keinen Konsumenten, d. h. es handelt sich um Fall 1.

Für Fall 2 lässt sich konzeptuell die Suche nach einem Konsumenten mit der Wrapper-Funktion wiederholen. Diese wird aber zunächst ausgeklammert, denn hier können mit den vorher genannten Anforderungen diese projektspezifischen Funktionen vom Nutzer in die Suche mitaufgenommen werden.

Für Fall 4 muss die nächste Nutzung der Variable gefunden werden, welche einen Einfluss auf den Kontrollfluss haben kann. Diese wird der „Konsument der Variable“ genannt. Diese Suche unterscheidet sich von der vorherigen, da hier nicht nur Elternausdrücke geprüft werden müssen sondern auch nachfolgende Geschwisterausdrücke.

Ist einer von diesen Fällen erreicht, ist die Konsumentenlokalisierung abgeschlossen. Für Fall 3 und 4 muss es nun zur Kontroflussauswertung kommen.

3.5.2 Kontrollflussauswertung

Um zu wissen, ob der Fehlerfall von den Tests abgedeckt ist, muss bekannt sein, wie die Kontrollflussentscheidung ausfällt (if/else-Zweig verfolgt, welcher switch-Case etc.). Dazu wird benötigt:

1. Der Wertebereich von Fehlern
2. Der boolesche Ausdruck der Kontrollflussanweisung
3. Abhängigkeiten des booleschen Ausdrucks (benutzte Variablen und deren Werte)

Allgemein lässt sich Punkt 3 erst zur Laufzeit des Programms beantworten, beispielsweise kann der Ausdruck von einem Funktionsparameter abhängen und der Ausdruck dann nicht für alle Aufrufe der Funktion gleich ausfallen. Daher lässt sich nicht immer mit Gewissheit statisch feststellen, zu welchem Wert der boolesche Ausdruck evaluiert. Trotzdem wird vermutet, dass die meisten der Kontrollflüsse von Fehlerbehandlungsstellen statisch ableitbar sind, denn in **POSIX** werden die meisten Fehlerwerte als ein „Wert ungleich 0“ oder als negativer Wert repräsentiert und daher oft mit der statisch bekannten Konstante 0 verglichen.

Es wird daher für boolesche Ausdrücke folgendes Modell gewählt (Operationen und Datentypen sind zur Klarheit als Haskell-Code angegeben, da hier im Gegensatz zu Pseudocode eine konkrete operationelle Semantik besteht):

```
1  -- Min, Max.
2  data Range = Range Int Int
3  -- Hat immer mindestens einen Range.
4  data RangeSet = RangeSet Range (Maybe RangeSet)
5  -- Alle unterstützten binären Operatoren.
6  data BinOp = LessThan | GreaterThan | GreaterEqual
7             | LessEqual | Equal | NotEqual | And | Or
8  -- Alle unterstützten unären Operatoren.
9  data UnOp = BooleanNegate | ArithmeticNegate
10
11 -- Alle unterstüzten booleschen Ausdrücke.
12 data Expr t = NonTermBin (Expr t) BinOp (Expr t)
13              | NonTermUn UnOp (Expr t)
14              | Term t
15              deriving (Show)
16
17 -- Ausdrücke die sich direkt evaluieren lassen.
18 data LeafExpr t = NonTermUn UnOp t
19                  | NonTermBin t BinOp t
20                  | Term t
21
22 -- Interpretation eines RangeSet.
23 -- Beinhaltet das Set nur:
24 -- Werte ungleich 0 -> Yes
25 -- Werte gleich 0 -> No
26 -- Beides -> Unknown
27 -- Diese Interpretation deckt sich mit der Wahrheitsemantik von C.
28 -- Alle Integer ungleich 0 sind wahr.
29 data TriState = Yes | No | Unknown
30 interpretRangeSet :: RangeSet -> TriState
```

```haskell

-- Boolesche Ausdrücke ohne freie Variablen.
type SubstitutedExpression = Expr RangeSet
-- Boolesche Ausdrücke mit freien Variablen.
-- Freie Variablen entsprechen bennanten Löcher in dem Ausdrucksbaum.
-- Diese können durch beliebige Ausdrücke substituiert werden.
type UnsubsitutedExpression = Expr (Either RangeSet String)

-- Ersetzt alle freien Variablen mit RangeSet welche alle Integer enthalten.
substituteFreeVariables :: UnsubsitutedExpression -> SubstitutedExpression
-- Substituiert alle Vorkommen einer freien Variable mit einem RangeSet.
substituteSingleVariable :: UnsubsitutedExpression ->
                            (String, RangeSet) -> UnsubsitutedExpression
-- Substituiert alle benannten Variablen mit dem dazugehörigen RangeSet,
-- unbekannte Variablen werden mit dem Bereich aller Integer subsituiert.
substitute :: UnsubsitutedExpression -> [ (String, RangeSet) ] ->
    SubstitutedExpression

-- Evaluiert ein Blatt des Ausdrucksbaum.
evaluateLeaf :: LeafExpr RangeSet -> RangeSet

-- Evaluiert einen kompletten Ausdrucksbaum.
evaluate :: SubstitutedExpression -> RangeSet
evaluate (Term t) = evaluateLeaf (Term t)
evaluate (NonTermUn op (Term t)) = evaluateLeaf (NonTermUn op t)
evaluate (NonTermUn op nt) = evaluateLeaf (NonTermUn op (evaluate nt))
evaluate (NonTermBin l op r) = evaluateLeaf (NonTermBin (evaluate l) op (
    evaluate r))
```

Der Algorithmus zur Evaluation eines Ausdrucks verhält sich in der Laufzeit linear in Abhängigkeit zu den Knoten im Ausdrucksbaum. Für jeden Knoten und dessen bereits evaluierte Kinder wird `evaluateLeaf` exakt einmal aufgerufen. Zum besseren Verständnis folgen einige Illustrationen der Evaluation:

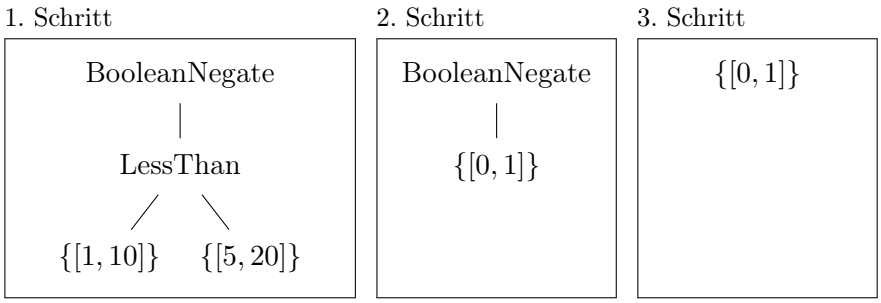

Abbildung 3.8: Visualisierung der Auswertung eines Ausdrucks zu `Unknown`

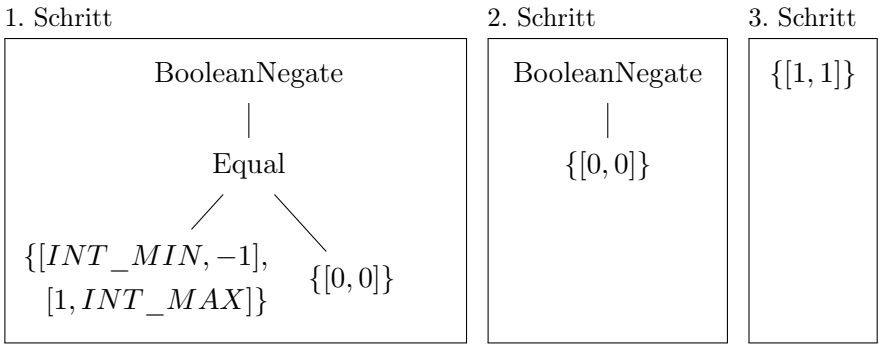

Abbildung 3.9: Visualisierung der Auswertung eines Ausdrucks zu `Yes`

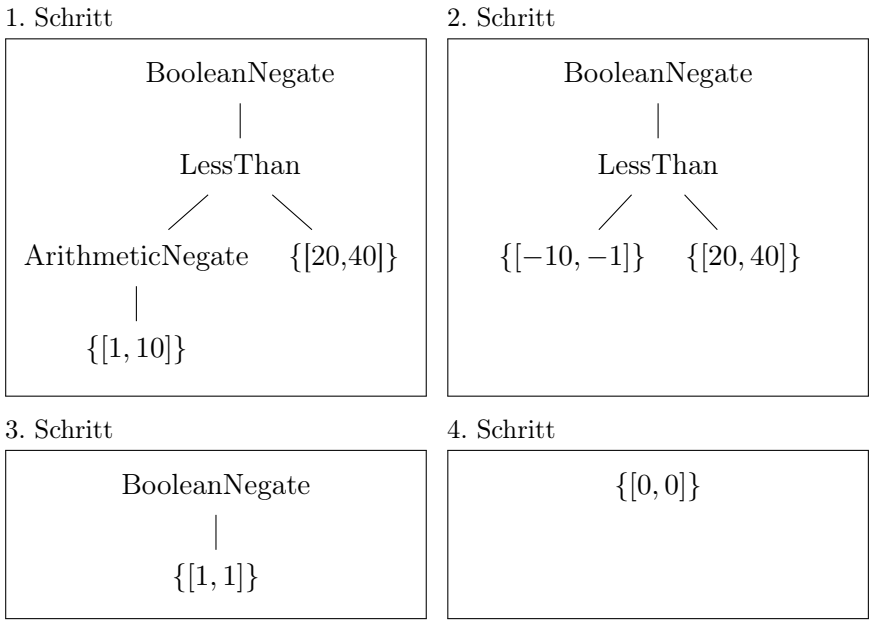

Abbildung 3.10: Visualisierung der Auswertung eines Ausdrucks zu `No`

Für `evaluateLeaf` lässt sich ein Algorithmus finden, dessen Laufzeit mit dem Produkt der Anzahl der Integerbereiche auf jeder Seite eines binären Operators skaliert. Das kann in der Praxis bei einer großen Anzahl von Integerbereichen zu einer nicht tragbaren Laufzeit der Evaluation führen. Die Operatoren wurden so gewählt, dass ein evaluierter Ausdruck nie mehr Integerbereiche umfasst als die Kindausdrücke. Operatoren, bei denen das nicht der Fall ist und die deswegen zu einer Explosion der Laufzeit führen können, sind Multiplikation sowie alle Bitoperatoren bis auf Rechtsshift. Beispielsweise führt eine Multiplikation des kompletten 32-bit-Integerbereichs $[0, 4294967295]$ zu einer Menge

von 2^{31} Bereichen $\{[0,0],[2,2],[4,4]\dots\}$ und somit zu einem nicht tragbaren Anstieg der Laufzeit. Für die Operatoren der Addition, Subtraktion, Division, Modulo und dem Linksshift gilt das nicht. Diese verändern bei der Evaluation nicht die Anzahl der Integerbereiche und können dementsprechend ohne Konsequenzen auf die Laufzeit im Modell nachgerüstet werden. Sie wurden unter der Annahme weggelassen, dass diese nicht häufig bei der Fehlerwertüberprüfung eingesetzt werden, um den Implementationsaufwand gering zu halten. In der Implementation können nicht unterstützte Operatoren gemeldet werden, um zu überprüfen, wie tragbar diese Annahme ist.

Dieses Modell sollte für die meisten Fehlerbehandlungsfälle in C-Programmen hinreichend sein, und für Fälle, in denen Anhand des Modells nicht möglich ist, den Wert eines booleschen Ausdruckes im Fehlerfall herzuleiten, `Unknown` liefern, also auch messbar machen, wie viele Fehlerbehandlungsfälle davon nicht analysiert werden können.

Ein spezifisches Muster, welches in Fehlerbehandlungsfällen von C-Programmen vorkommt, von diesem Modell aber nicht statisch analysiert werden kann, findet sich z. B. bei den **Umgebungsaufrufen** `fread` [28] und `fwrite` [29].

Hier hängt der Fehlerbereich von dem konkreten Wert der Argumente ab. Diese Abhängigkeit muss während der Auswertung erhalten bleiben, um statisch Vergleiche zwischen Fehlerwerten und Argument zu ermöglichen. Es folgt eine kurze Illustration dieses Mangels:

```
1 // Vor Substitution
2 void readOrExit(void *buffer, size_t size, size_t count, FILE *stream) {
3   size_t result = fread(b, size, count, stream);
4   if(result < count) {
5     abort();
6   }
7 }
```

Zum Finden des Kontrollflusses im Fehlerfall muss der booleschen Ausdruck `result < count` evaluiert werden. Da `readOrExit` mit beliebigen Parametern aufgerufen werden kann, muss `count` mit dem kompletten Integerbereich substituiert werden. `result` kann im Fehlerfall den kompletten Integerbreich außer der größten Zahl annehmen. `result < count` wird also zu `[0, SIZE_MAX-1] < [0, SIZE_MAX]` substituiert. Für diesen Ausdruck lassen sich Belegungen finden, welche zu `Yes` sowie zu `No` auswerten (beispielsweise `0 < 2` evaluiert zu `Yes`, `2 < 0` evaluiert zu `No`). Daher evaluiert der Ausdruck nach dem

Modell zu `Unknown`, es kann statisch also nicht festgestellt werden, welcher Kontrollzweig im Fehlerfall gewählt wird.

Diese nicht analysierbaren **Umgebungsaufrufe** werden aufgrund des Implementations- und Modellieraufwands für diese Arbeit hingenommen. Konzeptuell ist es aber möglich, für `fread` und `fwrite` statisch festzustellen, wie der Kontrollfluss im Fehlerfall aussieht.

Es folgen ein paar Beispielanwendungen des in dieser Sektion beschrieben Verfahrens zur besseren Veranschaulichung:

1. Der untersuchte **Umgebungsaufruf** ist `malloc`, das untersuchte Programm ist folgendes:

```
1 void * mallocOrExit(size_t size) {
2    void *result = malloc(size);
3    if(result == 0) {
4       abort();
5    }
6    return result;
7 }
```

   Es wird zunächst jeder Aufruf von `malloc` gefunden. Dies ist im Beispiel nur Zeile 2. Der Konsument des Rückgabewertes wird gesucht. Dieser ist die Variable `result` (Fall 4). Der nächste Konsument der Variable wird gesucht, dabei wird die Kontrollflussanweisung in Zeile 3 identifiziert. Die Konsumentenlokalisation ist abgeschlossen, nun muss der Kontrollfluss im Fehlerfall ausgewertet werden. Der boolesche Ausdruck (`result == 0`) wird anhand des Modells evaluiert. Dabei wird `result` mit dem Fehlerbereich substituiert. Per Dokumentation ist dies bei `malloc` der Wert 0. Nach der Substitution erhält man also `0 == 0`. Dieser Ausdruck evaluiert stets zu `Yes`. Im Fehlerfall geht der Kontrollfluss also in den if-Zweig, Zeile 4 wird als Fehlerbehandlung identifiziert.

2. Der untersuchte **Umgebungsaufruf** ist `getenv`, das untersuchte Programm ist folgendes:

```
1 char const * getUsername(void) {
2    return getenv("USER_NAME"):
3 }
```

   Es wird zunächst jeder Aufruf von `getenv` gefunden. Dies ist im Beispiel nur Zeile 2. Der Konsument des Rückgabewertes wird gesucht. Es wird der Rückgabeausdruck

identifiziert (Fall 2). Es wird gemeldet, dass `getUsername` einen Fehlerfall kommuniziert. Der Nutzer kann jetzt `getUsername` als eigene Funktion mit Fehlerfall definieren, um eine korrekte Behandlung des Fehlers sicherzustellen.

3. Der untersuchte **Umgebungsaufruf** ist `fprintf`, das untersuchte Programm ist folgendes:

```
1  void writeCSV(FILE * file, CSV const * csv) {
2     for(size_t row = 0; row < numRows(csv); row++) {
3        for(size_t column = 0; column < numColumns(csv) - 1; column++) {
4           fprintf("%" PRIu32 ",", getEntry(csv, column, row));
5        }
6        if(numColumns(csv) > 0) {
7         fprintf("%" PRIu32 ",", getEntry(csv, numColumns(csv) - 1, row));
8        }
9        fprintf("\n");
10    }
11 }
```

Es wird zunächst jeder Aufruf von `fprintf` gefunden. Diese liegen im Beispiel in den Zeilen 4, 7 und 9. Der Konsument des Rückgabewertes wird gesucht. Es wird keiner gefunden (Fall 1). Es wird gemeldet, dass in diesen Zeilen der Fehlerfall von `fprintf` nicht behandelt wird.

3.6 Erstellte Werkzeuge

3.6.1 Designeinschränkungen

Im ersten Schritt wird ein Mechanismus benötigt, um die zu untersuchenden **Umgebungsaufrufe** in den ausgewählten Projekten zu finden. Dafür ist ein syntaktisches Verständnis des Quellcodes nötig, der Code muss geparst werden, um anschließend die **Umgebungsaufrufe** zu lokalisieren. Für C-Programme gestaltet sich dies schwierig, da C ein mehrstufiges Kompilationsmodell (siehe 3.11) verwendet.

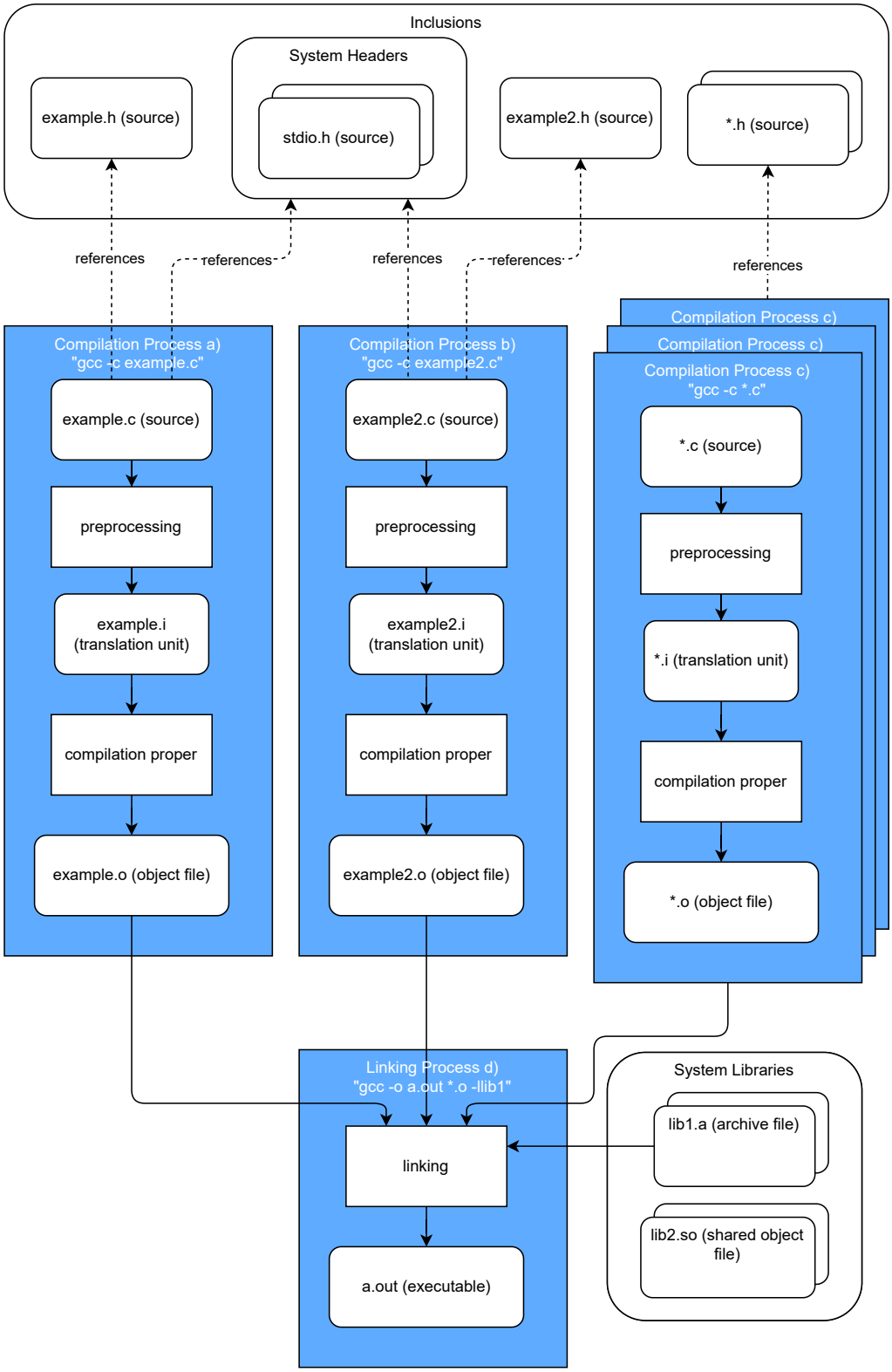

Abbildung 3.11: Typischer Bauprozess von C-Programmen

Im ersten Schritt muss der C-Präprozessor evaluiert werden, um den textuellen Inhalt einer C-Datei zu erhalten, welche dann im zweiten Schritt geparst werden kann. Die Evaluation des Präprozessors hängt aber nicht nur von der Quellcodedatei ab, sondern auch vom Bauprozess, lokalem Dateisystem, Plattform für die gebaut wird und sogar dem aktuellen Datum. Beispielsweise ist es möglich, je nach Bauprozess zwei komplett verschiedene C-Programme zu erhalten:

```
1  #include <stdio.h>
2  #ifdef PROG_A
3     void printSomethingA(void) {
4      printf("ABC");
5     }
6     void main(void) {
7      printSomethingA();
8     }
9  #else
10    void main(void) {
11     exit(0);
12    }
13 #endif
```

Definiert der Bauprozess `PROG_A`, erhält man das Programm:

```
1 void printSomethingA(void) {
2   printf("ABC");
3 }
4 void main(void) {
5   printSomethingA();
6 }
```

Definiert dieser es nicht, erhält man das Programm:

```
1 void main(void) {
2   exit(0);
3 }
```

Ob `PROG_A` gesetzt ist oder nicht, geht nicht aus dem Quellcode hervor. Diese Information kann nur durch Ausführung oder Analyse des Bauprozesses erlangt werden. Für den Bauprozess von C-Programmen existiert leider auch kein De-Facto-Standard, sondern es existieren viele verschiedene Buildsysteme, welche unterschiedliche Nischen ausfüllen [13][60][16]. Alleine in den untersuchten Projekten findet man die Nutzung von make [44], sowie autotools [4].

Zwischen allen Buildsystemen findet sich aber eine Gemeinsamkeit: Letztendlich wird der C-Compiler ausgeführt, um den Quellcode in ausführbare Dateien zu übersetzen. Davon gibt es nur eine Handvoll, die (weitestgehend) kompatibel sind (**GCC**/**clang**-Familie). Projekte versuchen oft auch, portabel zu diesen zu sein. Der Compiler hat definitionsgemäß alle nötigen Informationen, um den Quellcode korrekt zu interpretieren. Hier unterstützt **clang** ein Pluginsystem, welches das Einbinden von dynamischen Erweiterungen im Kompliationsprozess unterstützt. Dies erlaubt das bauprozessunabhängige Finden von **Umgebungsaufrufen**. Zusätzlich stellt **clang** eine **Abstract Syntax Tree (AST)**-Repräsentation des Quellcodes bereit, auf dessen Grundlage die in Sektion 3.5 skizzierte Suche vollzogen werden kann. In dieser Arbeit wird aus den angeführten Gründen die Implementierung über ein **clang**-Plugin gewählt.

3.6.2 Implementation

Um die vorangegangenen Anforderungen umzusetzen, wurde ein Anwendungspaket aus folgenden Bestandteilen erstellt:

1. Ein **clang**-Compiler-Plugin zur Untersuchung von Fehlerbehandlung von **Umgebungsaufrufen** „**error-analyzer**“.
2. Ein Programm zur allgemeinen Auswertung der Pluginausgabe.
3. Ein Programm zur Analyse der Abdeckung von Fehlercodepfaden anhand der Pluginausgabe sowie **LCOV**-Testabdeckungsmetriken.
4. Ein Programm zur Übersetzung von einem für Menschen optimierten Fehlerbeschreibungsformat in das vom Plugin erwartete Format.

Zusätzlich wurde noch ein Hilfsprogramm zur Extraktion aller **POSIX**-Funktionsnamen sowie -beschreibungen aus der HTML-Version des **POSIX**-Standards erstellt. Da die manuelle Fehlerklassifizierung für alle 1183 **POSIX**-Funktionen den Zeitrahmen einer Bachelorarbeit übersteigen würde (**POSIX** ist nicht in einer maschinenlesbaren **Interface Definition Language (IDL)** publiziert, sondern als Prosatext und die Fehlersemantik des Standards ist teilweise komplex), aber dennoch eine vollständige Abdeckung für die untersuchten Softwareprojekte gewollt ist, wurde folgendes Verfahren eingesetzt:

1. Alle **POSIX**-Funktionen werden initial mit einem Tag „unsupported“ markiert.

2. Der **error-analyzer** meldet für alle Funktionen, welche als „unsupported“ markiert sind, deren Vorkommen im kompilierten Code und führt keine weitere Analyse der Aufrufe aus.
3. Alle so gefundenen Aufrufe, also nur **POSIX**-Aufrufe, welche auch tatsächlich in den untersuchten Projekten eingesetzt werden, werden manuell in die Fehlerbeschreibung eingepflegt.

Dadurch kann die Anzahl der einzupflegenden Funktionen auf 555 reduziert werden.

Beim Einpflegen der Funktionen wurde versucht, die komplette relevante Fehlersemantik aus dem **POSIX**-Standard zu übertragen, auch wenn diese noch nicht von dem Compiler-Plugin unterstützt wird. So wird z. B. auch festgehalten, welche **errno**-Werte Fehlerwerte sind, und wann es sich z. B. um Logikfehler handelt.

Am Ende dieses Prozesses erhält man eine Fehlerdatenbank, die ca. die Hälfte aller **POSIX**-Funktionen abdeckt. Sollte noch ein weiteres Projekt untersucht werden, müssen voraussichtlich nur einige wenige Funktionen hinzugefügt werden, die nicht unter die am häufigsten verwendeten fallen. Eine Liste dieser wird direkt bei der initialen Ausführung des Plugins für ein neues Projekt ausgegeben.

Es folgt ein Beispielauszug der Fehlerdatenbank:

```
1  rand_r
2
3  random
4
5  read
6  Error: -1
7  Errno: EAGAIN, EBADF, EBADMSG, EINTR, EINVAL, EIO, EISDIR, EOVERFLOW,
       ECONNREST, ENOTCONN, ETIMEDOUT, EIO, ENOBUFS, ENOMEM, ENXIO
8  Exceptions:
9  Tag: logic
10 Errno: EBADF, EBADMSG, EINVAL, ENOTCONN
11
12 readdir
13 Error: NULL
14 Errno: EOVERFLOW, EBADF, ENOENT
15 Exceptions:
16 Tag: logic
17 Errno: EBADF
18
19 readdir_r
```

```
Tag: unsupported
```

Funktionen, welche keine Fehlerfälle haben, sind nur aufgelistet. Für alle anderen Funktionen sind Klassifizierung der Funktion, Fehlerwerte, **errno**-Werte und Ausnahmen von **errno**-Werten sowie deren Klassifizierung enthalten. Fehlerwerte unterstützten hierbei die Operatoren „<, >, !, <=, >=“ sowie ein logisches „Oder“ mehrerer Ausdrücke.

Das Format wurde aufgrund des hohen manuellen Aufwands bewusst als ein simples, für Menschen les- und editierbares Format gehalten. Der **error-analyzer** nimmt die Fehler aber in einem strukturierteren **JSON**-Format entgegen, um die Komplexität im Plugin gering zu halten und mögliche andere Quellen für Fehlerinformationen, welche z. B. in einer maschinenlesbaren **IDL** verfasst sind, leichter anbinden zu können.

Hier ist eine **JSON**-Übersetzung des vorangegangenen Auszugs. Eine Beschreibung als **JSON**-Schema [79] findet sich im Anhang A.1:

```
[
    {
        "name": "rand_r",
        "error": [],
        "errno": null,
        "comment": null,
        "tag": [],
        "exceptions": null
    },
    {
        "name": "random",
        "error": [],
        "errno": null,
        "comment": null,
        "tag": [],
        "exceptions": null
    },
    {
        "name": "read",
        "error": [
            {
                "operator": "EQUAL",
                "value": -1
            }
        ],
```

```
26          "errno": " EAGAIN, EBADF, EBADMSG, EINTR, EINVAL, EIO, EISDIR,
        ↪ EOVERFLOW, ECONNREST, ENOTCONN, ETIMEDOUT, EIO, ENOBUFS, ENOMEM,
        ↪ ENXIO",
27          "comment": null,
28          "tag": [],
29          "exceptions": "Tag: logic\nErrno: EBADF, EBADMSG, EINVAL, ENOTCONN\n"
30      },
31      {
32          "name": "readdir",
33          "error": [
34              {
35                  "operator": "EQUAL",
36                  "value": 0
37              }
38          ],
39          "errno": " EOVERFLOW, EBADF, ENOENT",
40          "comment": null,
41          "tag": [],
42          "exceptions": "Tag: logic\nErrno: EBADF\n"
43      }
44  ]
```

Der **error-analyzer** lässt sich dann mit der Fehlerbeschreibung für ein beliebiges **clang**-kompatibles C-Projekt als Seiteneffekt der Kompilation aufrufen. Dafür müssen lediglich folgende Flaggen zu der **clang**-Invokation, welche Objektdateien produziert, hinzugefügt werden: `-fplugin=$PATH_TO_ERROR_ANALYZER -fplugin-arg-error_analyzer-$PATH_TO_ERROR_DESCRIPTION`. Das lässt sich je nach Buildsystem unterschiedlich, zumeist aber durch Überschreiben der `CC` Umgebungsvariable, erreichen [67][10][9].

Der **error-analyzer** erstellt daraufhin eine `analysis.log`-Datei im Arbeitsverzeichnis des **clang**-Aufrufs. Diese Datei ist durch „advisory locks“ [27] geschützt, sodass auch parallele Kompilationsprozesse unterstützt werden.

Das Format der `analysis.log`-Datei ist ein „Quasi-JSON“-Format, um einen „append only“-Gebrauch durch den **error-analyzer** zu unterstützen, sowie um eine Konkatenation zwischen Dateien zuzulassen. Im Speziellen handelt es sich semantisch um einen **JSON**-Array. Es wurden aber die Array Klammern weggelassen sowie ein zusätzliches Komma als Suffix eingeführt. Also beispielsweise `1,2,3,` anstelle von `[1,2,3]`. In diesem Format kann einfach ein **JSON**-Objekt gefolgt von einem Komma effizient via „O_APPEND“ [53] an die Datei angehängt werden. Die Überführung in valides **JSON** kann bei der Auswertung durch Hinzufügen von Klammern und Entfernung des Kommas passieren.

Eine Beschreibung des Formats nach dieser Transformation in valides **JSON** findet sich im Anhang A.2.

Schlussendlich lässt sich die `analysis.log` Datei mithilfe der weiteren Programme des Programmpaketes auswerten.

Zur besseren Illustration folgt hier die in dieser Arbeit benutzte Anbindung in Form eines „bash“ Scripts, des „jq“-Projekts:

```
1  #!/usr/bin/env bash
2  # Enable sane error handling.
3  set -euo pipefail
4  # Enter the project directory.
5  pushd jq
6  # We want to instrument the entire compilation process
7  # so we need a clean build.
8  # Clean before building.
9  make clean
10 # jq uses autotools (configure script), set CC before running configure
11 # to use our clang plugin during compilation.
12 CC="clang -fplugin=~/bachelor/failure_injection/tools/clang_plugins/
       error_analyzer/build/liberror_analyzer.so -fplugin-arg-error_analyzer-~/
       bachelor/failure_injection/tools/clang_plugins/error_analyzer/posix.json"
        ./configure
13 # Delete all previous analysis.
14 find -name analysis.log -type f -delete
15 # Compile jq this creates analysis.log during compilation
16 make
17 # Copy complete analysis into our original working directory.
18 cp analysis.log ..
19 # Switch to the original working directory.
20 popd
21 # Analyze log file,
22 # exclude all functions tagged deterministic or logic.
23 # Print to screen and report.txt
24 ../tools/clang_plugins/error_analyzer/analysisInfo.rb -f deterministic,logic
       analysis.log | tee report.txt
25 # Analyze coverage of errors cases.
26 ../tools/clang_plugins/error_analyzer/lineCovered.rb -f deterministic,logic -
       b ~/bachelor/failure_injection/jq/jq/ cov.lcov  analysis.log | tee -a
       report.txt
```

3.6.3 Auswertung der Werkzeuge

Datenerfassung

Um die Effektivität der entwickelten Werkzeuge zu überprüfen, wurden diese wie in der letzten Sektion beschrieben auf die in dieser Arbeit untersuchten Projekte angewandt. Die Ergebnisse davon sind in den Tabellen 3.3 und 3.4 sowie in dem Diagramm 3.12 festgehalten.

Projekt	Weiter-gegeben	Behandelt	Gesamt
cURL	72	371	443
git	2652	702	3354
jq	113	60	173
redis	780	215	995

Tabelle 3.3: Verfolgbare Umgebungsfehler

Projekt	Variable nicht verfolgbar	Unbekannte Operation	Nicht statisch Auswertbar	Gesamt
cURL	50	14	7	71
git	168	146	59	373
jq	15	7	2	24
redis	98	29	15	142

Tabelle 3.4: Nicht verfolgbare Umgebungsfehler

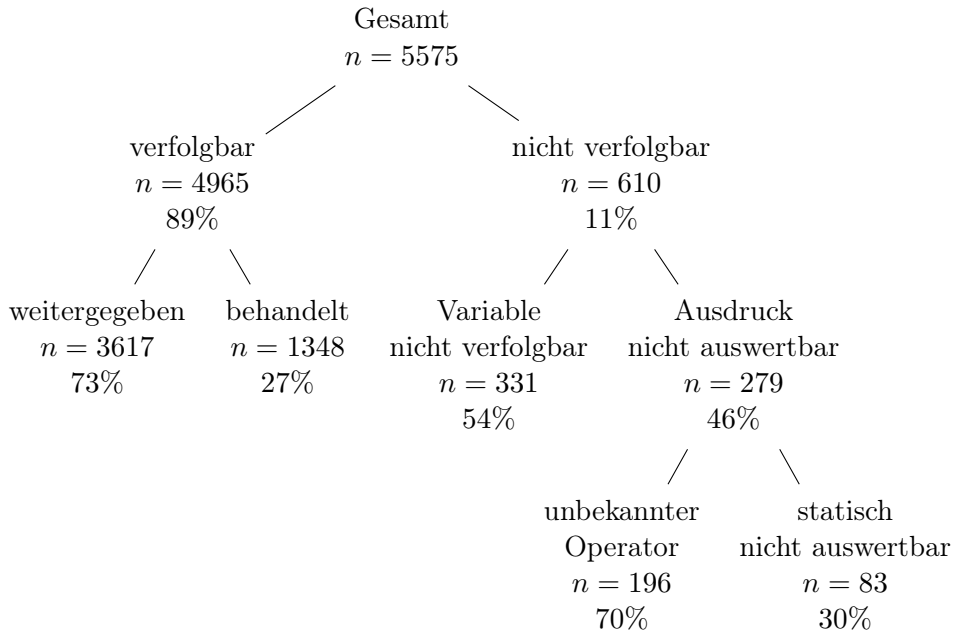

Abbildung 3.12: Visualisierung der Auswertung

Es wird sichtbar, dass die Werkzeuge für die meisten **Umgebungsaufrufe** in der Lage sind, den Kontrollfluss im Fehlerfall zu ermitteln. Von 5575 **Umgebunsaufrufen** sind 610 nicht verfolgbar, also nur ca. 11%. Wichtig bei diesem Ergebnis ist, dass die meisten **Umgebungsaufrufe** den Rückgabewert weitergeben oder ignorieren (Fall 1 und 2 der Fallunterscheidung aus den Anforderungen). Diese Fälle sind leicht zu finden und führen einen Hauptteil des erarbeiten Modells (statische Evaluation und variable Verfolgung) nicht aus. Gleichzeitig machen sie einen beachtlichen Teil der verfolgbaren Fehler aus: 3617 von 4965, also ca. 73%.

Klammert man diese aus, sind von 1958 **Umgebungsaufrufen** 610 nicht verfolgbar, also ca. 31%. Von diesen sind 331, also ca. 54%, auf nicht verfolgbare Variablen zurückzuführen. Für Ausdrücke, welche sich bis zu einer Fallunterscheidung verfolgen lassen, sind 279 von 1627 Ausdrücke nicht auswertbar, also ca. 17%. Von diesen sind 196 auf nicht unterstützte Operatoren zurückzuschließen, also ca. 70%. Für unterstützte Operatoren sind dann 83 von 1431 nicht auswertbar, also ca. 6%.

Analyse nicht auswertbarer Ausdrücke

Bei den nicht unterstützten Operatoren beläuft es sich auf folgende:

clang-Typ	Anzahl
CharacterLiteral	1
ConditionalOperator	1
BinaryOperator ∗	1
BinaryOperator /	2
BinaryOperator =	2
BinaryOperator +	4
BinaryOperator −	4
ArraySubscriptExpr	12
BinaryOperator &	51
MemberExpr	53
UnaryOperator ∗	65

Tabelle 3.5: Nicht unterstützte Operatoren, Beschreibung findet sich in der **clang**-Dokumentation [12]

Die Operatoren „CharacterLiteral“, „BinaryOperator =“, „ArraySubscriptExpr“, „MemberExpr“, sowie „UnaryOperator ∗“ lassen sich trivial in den entwickelten Werkzeugen nachrüsten. Bei „CharacterLiteral“, lässt sich bei der Bildung von booleschen Ausdrücken mit dem dazu gehörigen Wert substituieren. Momentan geschieht dies nur für „IntegerLiterals“ und wurde schlichtweg für „CharacterLiterals“ vergessen. Bei „BinaryOperator =“ handelt es sich um Variablenzuweisungen, bei denen der Wert direkt in einer Kontrollflussanweisung benutzt wird. Diese lassen sich mit dem Wert der Zuweisung substituieren. Die restlichen Operatoren, also „ArraySubscriptExpr“, „MemberExpr“ sowie „UnaryOperator ∗“ können jeweils mit dem für den Typ erlaubten Wertebereich substituiert werden wie es momentan auch für freie Variablen passiert.

Die Operatoren „BinaryOperator +“ und „BinaryOperator −“ müssen im Modell nachgerüstet werden und sind daher nicht komplett trivial zu implementieren. Die Nachrüstung im Modell ist jedoch konzeptuell möglich.

Da das Modell mit Mengen an Integerbereichen arbeitet, lassen sich die Operatoren „BinaryOperator ∗“, „BinaryOperator /“ und „BinaryOperator &“ nicht ohne weiteres einarbeiten. Grund hierfür ist, wie schon in der Sektion 3.5 angeführt, eine Aushebelung der Integerbereiche. Effektiv muss dann mit statt eines Bereichs auf einmal mit der Menge fast aller Integer gearbeitet werden, was zu einem massiven, nicht praktikablen Anstieg der Laufzeit beiträgt.

Zusammengefasst lassen sich von den 196 nicht unterstützten Operatoren 141 ohne großen Aufwand nachrüsten. Geht man von einer gleichbleibenden Rate für Nichtauswertbarkeit von Ausdrücken aus, also 83 von 1431 (ca. 6%), dann lassen sich von diesen neu unterstützten Operatoren 8 Fälle nicht auswerten und 133 Fälle auswerten. Für Ausdrücke, welche sich bis zu einer Fallunterscheidung verfolgen lassen, sind dann 146 anstelle von 279 Ausdrücken aus insgesamt 1627 nicht auswertbar, also ca. 9% anstelle von 17%.

Analyse nicht verfolgbarer Variablen

Um die nicht verfolgbaren Variablen weiter zu untersuchen, wurden pro Projekt stichprobenartig jeweils 10 zufällige Beispiele betrachtet. Die so erhaltenen 40 Fälle wurden manuell betrachtet und nach Ursachentyp gruppiert:

Typ	Anzahl
Bug (**BUG**)	1
Nur erstes Match (**NXT**)	1
Pointer-Indirektion (**PTR**)	1
Direkte Nutzung (**IMM**)	5
Nur in if-Statements (**NIF**)	5
Fehler ignoriert (**IGN**)	11
Komplexer Kontrollfluss (**CPX**)	16

Tabelle 3.6: Nicht verfolgbare Variablen

Es folgt eine Erläuterung der einzelnen Ursachentypen:

1. **BUG**

   Es wurde ein Bug bei dem Algorithmus für die Suche nach der nächsten Nutzung einer Variable gefunden. In der Implementierung wird der **AST** des Quellcodes

vom **Umgebungsaufruf** aus Blatt für Blatt für die erste Nutzung durchsucht. Mit diesem Algorithmus manifestiert sich folgendes Problem:

```
1  void main(void) {
2    int err;
3    if(true) {
4      err = getError(); // <- Umgebungsaufruf
5    } else {
6      err = 0; // <- Wird als erste Nutzung von err gefunden.
7    }
8     if(err) { // <- Sollte eigentlich als erste Nutzung von err gefunden
       werden.
9      // handle error
10   }
11 }
```

Zur Lösung des Problems muss der Suchalgorithmus nicht genommene Zweige ausklammern. Konzeptuell ist dies einfach umzusetzen, die Implementierung wird dadurch aber weitaus komplizierter, da momentan relativ trivial ein Baum durchlaufen wird. Durch diesen Bug gefundene Falsch-Positiv-Treffer (also das Finden von Fehlerbehandlung, die gar keine ist), sollte relativ unwahrscheinlich sein, denn man kann davon ausgehen, dass Geschwisterzweige entweder beide den Fehlerwert behandeln oder beide den Fehlerwert initialisieren.

Ein weiteres Problem, was durch diesen Algorithmus aufkommt, ist die Behandlung von Fehlern in Zweigen:

```
1  void main(void) {
2    int err = getError(); // <- Umgebungsaufruf
3    if(getStrategy()) {
4      if(err) {  // <- Wird momentan als erste Nutzung von err gefunden.
5        // handle error, strategy one
6      }
7    } else {
8      if(err) {
9        // handle error, strategy two
10     }
11   }
12
13 }
```

Fehlt hier die Fehlerbehandlung „strategy two“, wird dies momentan nicht aufgedeckt. Fehlerbehandlung ist zwar in allen Zweigen nötig, aber die Werkzeuge unter-

stützen nur eine Fehlerbehandlung pro **Umgebungsaufruf**. Konzeptuell kann der Suchalgorithmus versuchen, jeweils eine Fehlerbehandlung pro Kontrollflusszweig zu finden sowie Zweige zu melden, in denen keine Fehlerbehandlung existiert. Dies hat aber weitreichende Folgen für die entwickelten Werkzeuge, da es Annahme von einer Fehlerbehandlung pro **Umgebungsaufruf** verletzt und somit keine einfach umzusetzende Änderung ist.

2. **NXT**

   In der Implementation wird immer nur jeweils nach der nächsten Nutzung einer Variable gesucht. In diesem Fall erfolgt die Fehlerbehandlung aber in einer späteren Nutzung.

```
1 void main(void) {
2   int err;
3   err = getError(); // <- Umgebungsaufruf
4   int err2 = err; // <- Algorithmus bricht hier ab, die weitere Nutzung
      in der nächsten Zeile wird nicht gefunden.
5   if(err) {
6     // handle error
7   }
8 }
```

   Grund hierfür ist, dass versucht wird, den ersten Einfluss auf den Kontrollfluss des Programms zu finden. Die vorherige Nutzungen der Variable kann aber potenziell schon Einfluss auf den Kontrollfluss haben, oder der Einfluss kann durch Überschreiben der Variable verloren gehen. Beispiel:

```
1  void main(void) {
2   int err;
3   err = getError(); // <- Umgebungsaufruf
4   clearError(&err); // <- Fehler wird zurückgesetzt, die
      Fehlerbehandlung kann nie erfolgen
5   if(err) {
6     // handle error
7   }
8 }
```

   Diese Fälle benötigen eine aufwändigere statische Analyse und dementsprechend ein Modell, welches über das in dieser Arbeit Entwickelte hinausgeht.

3. **PTR**

Anstatt Variablen direkt zu nutzen, wird Pointer-Indirektion für die Speicherung und Prüfung des Fehlers genutzt. Beispiel:

```
1 void main(void) {
2  int err;
3  int *errPtr = &err;
4  *errPtr = getError(); // <- Umgebungsaufruf
5  if(*errPtr) { // <- Nächste Nutzung der Fehlervariable gefunden, kann
      aber nicht ausgewertet werden.
6    // handle error
7  }
8 }
```

Dies wurde in der Implementation nicht gesondert behandelt, für simple Fälle sollte es aber nachrüstbar sein.

4. **IMM**

   In C wertet der Zuweisungsoperator zu dem Wert der rechten Seite aus, daher wird manchmal das Setzen einer Variable und die Fehlerprüfung kombiniert.

```
1 void main(void) {
2   int readBytes;
3   if((readBytes = read()) < 0) {
4     exit(1);
5   }
6   printf("%d bytes were read\n", readBytes);
7 }
```

   Der **error-analyzer** priorisiert aber die Zuweisung und versucht die nächste Nutzung der Fehlervariable zu finden, anstatt den Ausdruck selbst als Fehlerbehandlung zu werten. Dieses Problem zu beheben, sollte auf eine triviale Implementationsänderung hinauslaufen: Ist eine Zuweisung eines **Umgebungsrückgabewertes** Teil einer Kontrollflussanweisung, soll diese direkt ausgewertet werden.

5. **NIF**

   Die Implementation des **error-analyzer** behandelt die Findung von Konsumenten für **Umgebungsaufrufe** und Variablen unterschiedlich. Für **Umgebungsaufrufe** wird neben einer direkten Nutzung in einer Fallunterscheidung die Detektion einer Weitergabe über Funktionsargumente oder als Rückgabewert unterstützt. Für Variablen fehlt diese Detektion. Hier wird nur die direkte Nutzung einer Variable in einer Fallunterscheidung unterstützt. Als Beispiel gegenübergestellt:

```
1  int makeCallA(void) {
2   return getError(); // <- Umgebungsaufruf, Rückgabewert wird direkt
       weitergegeben und als weitergebener Fehler erkannt.
3  }
4  int makeCallB(void) {
5   int err = getError(); // <- Umgebungsaufruf, Rückgabewert wird in einer
       Variable gespeichert und nach der nächsten Nutzung gesucht.
6   return err; // <- Nächste Nutzung ist gefunden, Rückgabe aber nicht
       unterstützt. Fehlerbehandlung als nicht gefunden gewertet.
7  }
8
9  void makeCallC(void) {
10  int errA = getError(); // <- Umgebungsaufruf, Rückgabewert wird in
       einer Variable gespeichert und nach der nächsten Nutzung gesucht.
11  if(errA) { // <- Nächste Nutzung gefunden, Fehlerbehandlung gefunden.
12    // handle error
13  }
14  if(getError()) { // <- Umgebungsaufruf, direkte Nutzung,
       Fehlerbehandlung gefunden.
15    // handle error
16  }
17 }
```

Grund hierfür ist eine fehlende Behandlung in der Implementierung. Konzeptuell spricht nichts dagegen, diese Detektion nachzurüsten.

6. **IGN**

   Fehlerwerte werden tatsächlich ignoriert. Diese sollten bestenfalls dem Nutzer als ignoriert gemeldet werden. Allgemein ist dies aber ohne weiteres nicht möglich, da dies nur über manuelle, teilweise globale Analyse identifiziert wurde. Triviale Fälle, in denen eine Fehlervariable beispielsweise kein weiteres Mal in einer Funktion benutzt wurde, sollten aber trotzdem detektierbar sein. Diese machen aber von den **IGN**-Fällen nur einen Fall aus.

7. **CPX**

   Es handelt sich um einen komplexeren Kontrollfluss. Beispielsweise werden Fehler in einer Variable aggregiert und dann gemeinsam behandelt:

```
1  void makeCall(void) {
2   int err = getErrorA(); // <- Umgebungsaufruf A
```

```
3  err |= getErrorB(); // <- Umgebungsaufruf B, err ist "true" wenn A oder
       B einen Fehler melden.
4  if(err) { // <- Gemeinsame Fehlerbehandlung
5    // handle error
6  }
7 }
```

Diese Fälle benötigen eine aufwändigere statische Analyse und dementsprechend ein Modell, welches über das in dieser Arbeit Entwickelte hinausgeht.

Von diesen Problemen lassen sich **NIF** und **IMM** trivial lösen. Diese machen ca. 25% der nicht verfolgbaren Variablen aus. Nutzt man diese Rate und bezieht sie auf die in der Auswertung erhaltenen Werte, gingen mit den Verbesserungen von 331 nicht verfolgbaren Variablen 83 zu verfolgbaren über. Mit der in der Auswertung ermittelten Rate gäbe es 61 (73%) weitergegebene Fehler, welche immer korrekt gemeldet werden und 22 (27%) Fehler in Fallunterscheidungen, die ausgewertet werden müssen. Nimmt man die Auswertungsrate der überarbeiteten Auswertung, sind 2 Fehler (9%) nicht auswertbar und 20 (91%) auswertbar.

Auswertung der Analyse

Durch die erste Verbesserung kommt man von 279 nicht auswertbaren Fallunterscheidungen auf 146. Durch die zweite Verbesserung kommt man von 331 nicht verfolgbaren Variablen auf 20 auswertbare Fallunterscheidungen, 2 nicht auswertbare Fallunterscheidungen, 61 weitergegebene Fehler und 248 nicht verfolgbare Variablen. Dadurch ändert sich die Zahl nicht verfolgbarer Fehler von 610 auf 396, also eine Steigerung der Fehlererkennung von 89% auf 93%.

Zusammengefasst sind die entwickelten Werkzeuge hocheffektiv darin, Kontrollfluss bei Umgebungsfehlern auszuwerten sowie fehlende Fehlerbehandlung zu finden. Diese Effektivität lässt sich ohne größere Abstriche oder Neukonzeption noch weiter verbessern. Die Werkzeuge lassen sich einfach in Projekte unabhängig vom Buildsystem einbinden, belegt durch die Integration der in dieser Arbeit untersuchten Projekte. Es lassen sich projektspezifische **Umgebungsaufrufe** untersuchen, wie sie unter anderem bei „cURL“ erforderlich waren, was belegt, dass der projektspezifische Nutzen über die in dieser Arbeit vorgesehene Abdeckung von **POSIX-Umgebungsaufrufen** hinausgehen kann.

Viele der gefundenen Mängel können nur über bessere Kontrollflussanalyse behoben werden. Es stellt sich die Frage, ob ein massiver Anstieg der Komplexität z. B. durch Einführung von symbolischer Ausführung, Bit-Vector-Arithmetik etc. hinzunehmen ist, wenn die Werkzeuge jetzt schon hocheffektiv sind. Da eine perfekte Umgebungsfehlerverfolgung nicht möglich ist, bewegt sich die Effektivität der Werkzeuge mit steigendem Entwicklungsaufwand auf 100% zu. Eine viel aufwändigere Neukonzeption, näher am Stand der Technik von statischer Analyse, um weniger als 7% von Umgebungsfehlern zu verfolgen, wird als nicht gerechtfertigt befunden.

Trotzdem lassen sich noch Weiterentwicklungsmöglichkeiten finden, welche den produktiven Einsatz der Werkzeuge für mehr Anwendungsfälle ermöglichen:

1. Quellcodeannotationen oder Konfigurationsdateien für das Ignorieren von Fehlern.

   Sollen die Werkzeuge nicht nur zur manuellen Analyse, sondern bei **Continuous Integration (CI)** als automatisch ausgeführtes Werkzeug zum Finden von Regressionen eingesetzt werden, ist es wichtig, Falsch-Positiv-Treffer markieren zu können, da diese sonst immer wieder gemeldet werden.

2. Unterstützung von `fread` sowie `fwrite`.

   Diese beiden Funktionen werden momentan aufgrund der Abhängigkeit zwischen Fehlerbereich und Funktionsparameter nicht unterstützt und auch nicht von der vorangegangenen Analyse erfasst. Es ist aber zu vermuten, dass diese relativ häufig, vor allem in Programmen, welche nicht auf **POSIX** abzielen sondern auf die C-Standardbibliothek, benutzt werden.

3.7 Ergebnisse

Die im Laufe dieser Arbeit entwickelten Werkzeuge wurden schließlich eingesetzt, um die Zeilenabdeckungsrate im Umgebungsfehlerfall zu ermitteln. Um sicherzustellen, dass das Ergebnis nicht durch allgemein untertestete **Umgebungsaufrufe** verfälscht ist, wurde die Auswertung nicht nur für Codepfade vollzogen, welche bei Umgebungsfehlern durchlaufen werden, sondern auch für die komplementären Codepfade, welche bei nicht fehlerhaften **Umgebungsaufrufen** durchlaufen werden.

Projekt	Zeilenabdeckung Fehlerpfad				Zeilenabdeckung kein Fehlerpfad			
	n	von	Prozent	Faktor weniger als Durchschnitt	n	von	Prozent	Faktor weniger als Durchschnit
cURL	21	213	9,86%	8,12	293	388	75,52%	1,06
git	364	849	42,87%	2,02	503	703	71,55%	1,21
jq	5	65	7,69%	10,30	35	38	92,11%	0,86
redis	56	312	14,72%	4,59	64	158	40,51%	1,67

Tabelle 3.7: Abdeckung von **Umgebungsrückgabewerten** abhängigen Kontrollfluss

Der Auswertung 3.7 kann man entnehmen, dass sich die Ausgangsfrage mit „Ja“ beantworten lässt. Im Vergleich mit der allgemeinen Zeilenabdeckung der Projekte sind die bei Umgebungsfehlern ausgeführten Codepfade mit einem Faktor von ca. 2-10 untertestet. Die alternative Erklärung, dass Codepfade, in denen **Umgebungsaufrufe** vorkommen, allgemein untertestet sind, lässt sich nicht bestätigen, denn für Codepfade, welche bei erfolgreichen **Umgebungsaufrufen** ausgeführt werden, ist der Faktor $<= 1,7$ und bei jq sogar besser als der Durchschnitt mit ca. 0,86. Fehlerzweige sind also im direkten Vergleich zu Nicht-Fehlerzweigen klar untertestet, genauso im Vergleich mit der durchschnittlichen Zeilenabdeckungsrate der untersuchten Projekte.

Als zusätzliche Ausgabe liefern die Werkzeuge eine Liste von ignorierten **Umgebungsrückgabewerten**.

Projekt	Ignorierte Rückgabewerte	Davon Ausgabeaufrufe	Prozent
cURL	54	24	$44,44\%$
git	2562	1493	$58,27\%$
jq	108	95	$87,96\%$
redis	755	526	$69,67\%$

Tabelle 3.8: Ignorierte **Umgebungsrückgabewerte** in den untersuchten Projekten

Es kann 3.8 entnommen werden, dass das Ergebnis schlimmer ist als erwartet, auch wenn es nicht direkt die Ausgangsfrage betrifft. Es fehlen nicht nur Tests der Fehlerbehandlung,

sondern die Fehlerbehandlung selbst in Vielzahl. Bei der Auswertung wurden Funktionsaufrufe, welche sich deterministisch verhalten (laut Definition dieser Arbeit keine **Umgebungsaufrufe**), sowie Aufrufe, welche nur bei Logikfehlern einen Fehler melden, explizit ausgeschlossen. Grund für diesen Ausschluss ist dass eine Fehlerbehandlung im Fall von Logikfehlern nicht notwendig ist, da diese nur im Falle eines Bugs auftreten und das Programm sich dann in einem nicht definiertem Zustand befindet. Alle hier verbleibenden und von den Projekten ignorierten Fehler müssen also behandelt werden, um eine korrekte Ausführung sicherzustellen.

Um zu untersuchen, warum die Rate der nicht behandelten Umgebungsfehler so hoch ausfällt, wurden die Fehler nach Art des **Umgebungsaufrufs** aufgeschlüsselt. Dabei wird sichtbar, dass in allen Fällen Ausgabefunktionen einen Großteil ($> 40\%$) der nicht behandelten Fehler ausmachen. Es wirkt, als ob dahinter eine Designentscheidung steht, und es nicht einfach vergessen wurde. Die hohe Anzahl an ignorierten Ausgaberückgabewerten deckt sich auch mit der **SEI CERT C** Regel [24], welche das Ignorieren bei Ausgabe nach **stderr [69]** oder **stdout [69]** erlaubt. Die so ausgesprochene Empfehlung ist aber fragwürdig, denn aus Sicht des Programmes ist nicht klar, ob es bei **stdout** um eine interaktive Ausgabe oder eine Pipe (bei der das Schreiben nicht fehlschlagen kann) oder um eine Ausgabe in eine Datei (kann fehlschlagen, wenn die Festplatte voll ist oder es andere Schreibfehler gibt) handelt. Daher lassen sich realistische Fehlerfälle konstruieren, in denen diese Ausgaben fehlschlagen und zu stiller Korruption führen können. Konkret lässt sich bei „redis“ ein solches Fehlverhalten beobachten. Die offizielle Dokumentation gibt das Kommando `redis-cli INCR mycounter > /tmp/output.txt` als Usecase an [57]. Wird hier aber in eine Datei auf eine volle Festplatte umgeleitet, entspricht der Inhalt der Datei nicht mehr der Erwartung. Die Datei ist korrumpiert, obwohl „redis-cli“ noch immer einen erfolgreichen Statuscode zurückgibt. Bei „cURL“ findet sich dieser Fehler nicht, für eine heruntergeladene Datei wird immer per Statuscode gemeldet, ob der Schreibvorgang nach **stdout** erfolgreich war. Auch „git“ führt für diesen Fall Wrapper Funktionen wie `fprintf_or_die` [31] ein, welche bei Schreibfehlern das Programm beenden. Trotzdem finden sich in allen Programmen viele Aufrufe von `fprintf` nach **stderr** mit ignorierten Rückgabewerten. Für **stderr** ist die verbreiteste Semantik, zusätzliche diagnostische Informationen zu liefern und keine Auswirkungen auf die Funktionalität des Programmes zu haben. Daher ist hier eine stille Korruption der Ausgabe nicht so risikobehaftet und möglicherweise weniger wichtig als die vollständige Abarbeitung des Programms. Z. B. will man einen Webserver nicht beenden, nur weil aus Platzgründen kein Logging mehr erfolgen kann. Die Entscheidung Program-

me bei fehlerhafter Ausgabe nach **stdout** zu beenden und fehlerhafte Ausgaben nach **stderr** zu ignorieren, sollte aber dennoch durch Hilfsfunktionen wie `fprintf_or_die` dokumentiert werden. Ansonsten muss für die Vielzahl aller von den Werkzeugen gefunden `fprintf` Aufrufe im Einzelfall entschieden werden, ob das Ignorieren des Rückgabewertes in Ordnung ist oder nicht.

4 Verbesserung des Ist-Zustands

4.1 Einleitung

In der letzten Sektion dieser Arbeit wurde gezeigt, dass Kontrollfluss bei Auftreten von Umgebungsfehlern untertestet ist. In dieser Sektion wird versucht, Ursachen dafür zu finden und Lösungswege zur Bildung einer aussagekräftigen Abdeckung für die untertesteten Codebereiche zu finden.

4.2 Mögliche Ursachen und Lösungsansätze

Über Ursachen des im letzten Abschnitt festgestellten Sachverhalts lässt sich leider nur spekulieren, es lassen sich aber vor allem zwei Gründe vermuten:

1. C unterstützt bis auf dynamische Bibliotheken nicht nativ das Konzept von Schnittstellen, daher lassen sich Konzepte wie „Dependency Injection“ [68], welche die Testbarkeit von Code erhöhen, nur spärlich anwenden. Lösungen dafür sind meistens projektspezifisch, wie beispielsweise „vtables“ bei Linux [52].
2. Bei **Umgebungsaufrufen** handelt es sich oft um „Cross-cutting concerns“ [68]. Dies lässt sich durch klassische Testtechniken wie z. B. „Mocks“ [71][48] nur schwer testen, ohne den Testcode stark an die Implementation zu koppeln und viel Code zu duplizieren.

Um eine bessere Aussage über den Einfluss dieser Vermutungen zu bekommen, wäre es denkbar, die in dieser Arbeit erarbeiteten Methoden auf andere Sprachen ohne Ausnahmebehandlung, dafür aber mit mehr Unterstützung für Testbarkeit, anzuwenden und die Ergebnisse zu vergleichen.

Unabhängig von der Ursache stellt sich die Frage, welche Anforderungen an Software im Umgebungsfehlerfall bestehen, die durch Tests überprüft werden können. Diese Frage

lässt sich nicht allgemein beantworten, da Anforderungen projektspezifisch sind. Einige Beispiele dafür sind:

1. Unterbrochene **Umgebungsaufrufe** über `EINTR` [62] sollen keine Auswirkungen auf das funktionale Verhalten des Programms haben.
2. Pro Netzwerkaufruf kann es zu bis zu drei Fehlern kommen (Retry-Policy), ohne dass es Auswirkungen auf das funktionale Verhalten des Programms gibt.
3. Es wird auch bei Beendigung des Programmes zwischen **Umgebungsaufrufen** eine Invarianz der Umgebung aufrechterhalten, beispielsweise atomares Schreiben einer Datei.

Es gibt aber eine Teilmenge von nichtfunktionalen Anforderungen, die für die meisten Projekte bestehen sollte. So kann man davon ausgehen, dass Software stets in einem definierten Zustand sein soll. Beispielsweise soll es nicht zu Sicherheitslücken durch Speichermanagement kommen. Genauso kann man fordern, dass es nicht zu Ressourcenlecks kommen soll, welche bei langer Laufzeit von Programmen Probleme verursachen können. Für manche der untersuchten Projekte werden diese Anforderung bereits für die Test-Suites der Projekte, aber nur selten im Umgebungsfehlerfall, durch generische Werkzeuge wie Valgrind [76], Compiler-Sanitizer [2] etc. oder eigenentwickelte Lösungen überprüft. Diese nichtfunktionalen Anforderungen, die für jeden **Systemtrace** eines Programms gelten sollen, werden hier „Programm-Invarianten“ genannt.

Aus allgemeiner Sicht gesprochen hat man für auf diese Art instrumentierte Projekte ein Korpus an ausführbaren Code (Test-Suites), für das die „Programm-Invarianten“ gelten soll, sowie ein automatisiertes Mittel dieses Korpus auf die Invarianten zu testen. Das Korpus exerziert jedoch Situationen, in denen es zu Umgebungsfehler kommt, nicht genug, sodass die Invarianten für solche Situationen nicht ausreichend geprüft werden können. Dieser Mangel lässt sich auf Projektebene durch das Schreiben neuer Test-Cases oder Erweiterung bestehender Test-Cases beheben. Da dieser Mangel besteht, wird aber aus den weiter oben angeführten Grunden vermutet, dass Schwierigkeiten in der Umsetzung vorliegen. Eine wichtige sich ergebende Observation ist, dass für die Überprüfung der „Programm-Invarianten“ kein semantisches Wissen über den Ausgang der Test-Cases erforderlich ist. D. h. die „Programm-Invarianten“ sollen immer für den Korpus erhalten bleiben, egal ob die Tests darin fehlschlagen oder nicht. Daher sollte es neben manueller Erzeugung von Test-Cases für Umgebungsfehler durch „Mocks“ u. ä. möglich sein, automatisiert und ohne Rücksicht auf die Testergebnisse Rückgabewerte von **Umgebungsaufrufen** mit Umgebungsfehlern zu ersetzen.

Diese Strategie der automatischen Injektion von Fehlern heißt in der Literatur **SFI**. Dabei muss hier klargestellt werden, dass Umgebungsfehler, die auf vorangegangen beschriebene Weise injiziert werden, nicht den „Faults“, von denen bei **SFI** die Rede ist, entsprechen. Die Injektion von Umgebungsfehlern spiegeln einen neueren Ansatz in der **SFI** wider [15][45][7]. Dabei wird davon ausgegangen, dass „Faults“, die in der Umgebung des Programms auftreten (beispielsweise Festplattenschäden, Abstürze von einem Kommunikationspartner etc.), zu Fehlern (aber noch immer wohl definiertem Verhalten) von **Umgebungsaufrufen** führen. Anstatt „Faults“ in einer Umgebung zu injizieren und darauf zu hoffen, dass diese zu Fehlern führen, welche Auswirkungen auf das Programm haben, wird dieser Schritt übersprungen und direkt diese Fehler injiziert. Leider ist hier die Nomenklatur schwammig, und es werden teilweise „Fault“, „Error“ und „Failure“ als Synonyme verwendet. Um mit diesem Forschungszweig konsistent zu bleiben, wird in dieser Arbeit in allen Fällen „Fault Injection“ als Name für diese Strategie benutzt.

Eines der untersuchten Projekte, „cURL“, setzt **SFI** für eine Teilmenge von **Umgebungsaufrufen** um und wird im nächsten Abschnitt untersucht.

4.3 Lösungsansatz von cURL

Im Laufe dieser Arbeit wurde bei der Untersuchung des „cURL“-Projekts der Einsatz von sogenannten „Torture Tests“ [75] festgestellt. Dabei handelt es sich um einen für das „cURL“-Projekt entwickelten Ansatz, Umgebungsfehler automatisiert bei der Ausführung von Tests zu injizieren und so eine höhere Testabdeckung für Fehlerbehandlung zu erreichen. Dies geschieht komplett Transparent, d. h. es müssen keine neuen Test-Cases geschrieben werden, sondern es werden bereits bestehende Test-Cases genutzt.

Auf das Modell dieser Arbeit übertragen, sieht der Lösungsansatz von „cURL“ folgendermaßen aus: Bei „cURL“ gibt es für jeden Test-Case eine ausführbare Datei. Die Ausführung eines Test-Case entspricht der Ausführung eines Betriebssystemprozesses, das Testergebnis dem Exit-Status des Prozesses. In der normalen Ausführung der Test-Suite werden einfach all diese Programme ausgeführt, und die Ergebnisse aggregiert. Sollen jetzt „Torture Tests“ angewandt werden, wird bei der Ausführung zusätzlich eine Instrumentation dazu geschaltet. Diese Instrumentation nimmt einen **Systemtrace** des Testlaufs auf, dabei wird eine Teilmenge der Interaktionspunkte aufgezeichnet. Konkret werden für folgende Funktionen die Parameter des **Umgebungsaufrufs** sowie der Rückgabewert der Umgebung aufgezeichnet:

`malloc`, `realloc`, `calloc`, `free`, `strdup`, `socket`, `close`, `accept`, `send`, `recv`

Für die Rückgabe der **Umgebungsaufrufe** wird von „cURL“ angenommen, dass diese für einen Testlauf stehts deterministisch sind (semantisch, bei `malloc` können z. B. andere Adressen zurückgeben werden; diese Adressen haben aber keinen Einfluss auf den Kontrollfluss des Programms). D. h. bei mehrfacher Ausführung eines Test-Cases werden stets dieselben **Umgebungsaufrufe** in der gleichen Reihenfolge protokolliert.

Die Aufzeichnung wird jetzt von „cURL“ benutzt, um für alle fehlschlagbaren **Umgebungsaufrufe** einen Fehlerfall zu erzeugen. Die Test-Cases werden jeweils für jeden aufgezeichneten **Umgebungsaufruf** neu ausgeführt. Die Instrumentation schaltet von einer passiven auf eine aktive Instrumentation um und injiziert nur für diesen Aufruf einen Umgebungsfehler. Die eigentlichen Ergebnisse des Test-Cases werden ignoriert. Es wird stattdessen überprüft, dass:

1. Alle Ressourcen wie Speicher und Dateien richtig behandelt wurden. Also es pro Ressource immer genau einen Erstell-Aufruf so wie ein Zerstörungs-Aufruf wie beispielsweise `malloc`/`free` und `fopen`/`fclose` gibt.
2. Es nicht zu einem Absturz gekommen ist.
3. Optional: Tools wie Valgrind überprüfen, dass es auch nicht zu Speicherkorruption kommt, welche keinen Absturz herbeiführt.

Um die Effektivität dieses Verfahrens zu ermitteln, wurden die Werkzeuge auf die Abdeckung bei „Torture Tests“ angewandt. Leider hat sich die Laufzeit der Test-Suite von ca. 1 Minute und 14 Sekunden ohne „Torture Tests“ auf mehrere Stunden mit „Torture Tests“ erhöht und wurde vor Beendigung abgebrochen. Um diese hohe Laufzeit zu erforschen, wurde der Testlauf durch Codeanpassungen instrumentiert, um Statistiken über die Laufzeit zu erhalten.

Es ist zu erwarten, dass die Laufzeit pro Test-Case linear mit Anzahl der **Umgebungsaufrufe** ansteigt. Jeder Test-Case muss pro **Umgebungsaufruf** ein weiteres Mal aufgerufen werden, um diesen fehlschlagen zu lassen. Liest z. B. ein Test-Case über eine Schleife 1 Mebibyte in 1024 Kibibyte Blöcken (mit gesonderten `read` Aufrufen), hat dieser Test-Case 1024 **Umgebungsaufrufe** zu `read` und wird von „Torture Tests“ 1024-mal ausgeführt, um für jedes `read` einen Fehler zu erzeugen. Natürlich sind die fehlschlagenden Ausführungen kürzer, da sie nur bis zu einem bestimmten Punkt des originalen Test-Cases gefolgt von Fehlerbehandlung laufen. Die Annahme, dass diese noch genauso lange brauchen wie der originale Test-Case, sollte aber eine gute Obergrenze für den

Faktor für die verlängerte Laufzeit eines Test-Case durch „Torture Tests“ geben. In dem Beispiel wäre der Faktor also 1024, dazu kommt vermutlich noch ein weiterer Faktor für Overhead.

Zunächst wurde die Anzahl von fehlschlagbaren **Umgebungsaufrufen** pro Test-Case ermittelt. Dabei wird hier aufgrund von externen Ausreißern auf eine grafische Darstellung verzichtet. Es wurde ermittelt, dass von 1481 Test-Cases der Median für die Anzahl an **Umgebungsaufrufe** 152 ist, das 3. Quartil 188 und das 95. Perzentile 324. Dabei gibt es einige extreme Ausreißer, mit 16 Test-Cases (1%) mit mehr als 1000 **Umgebungsaufrufe** und 4 Test-Cases (0,3%) mit mehr als 10000 **Umgebungsaufrufen**.

Die Ausführung der Tests lässt sich sehr gut parallelisieren, da diese unabhängig voneinander sind. Hier wurde die von „cURL“ empfohlene Rate von (28) „7 * number of CPU cores“ [18] genutzt. Dabei lassen sich einzelne Test-Cases nicht parallelisieren, sondern müssen seriell von Anfang bis Ende abgearbeitet werden. Dementsprechend ist die Mindestdauer für die Ausführung aller Tests mindestens so lang wie der längste Test-Case zur Ausführung benötigt. Daher wird vermutet, dass die lange Laufzeit durch die gefundenen Ausreißer verursacht wird. Um dies zu überprüfen, wurde die Ausführungsdauer pro Test-Case protokolliert sowie eine durchschnittliche Ausführungszeit pro Fehlerinjektionsschritt gebildet. Für die Test-Cases die abgebrochen wurden, wurde anhand des Durchschnitts auf die vollendete Laufzeit extrapoliert, da bekannt ist, wie viele Ausführungen der Test-Cases noch nötig sind, um eine komplette Umgebungsfehlerabdeckung zu erhalten.

Auch hier wurde aufgrund der Ausreißer auf eine visuelle Darstellung verzichtet. Dabei wurde der Median von 6 Sekunden, das 3. Quartil von 12 Sekunden und das 95. Perzentile von 75 Sekunden für die Dauer der Ausführung von einem Test-Case ermittelt. Hier gibt es Ausreißer von 11 Test-Cases die länger als 1000 Sekunden benötigen und einen Test-Case der länger als 10000 Sekunden (14584 Sekunden, ca. 4 Stunden) benötigt. Es lässt sich jetzt (bis auf einen Faktor von 2) berechnen, wie lange für die Ausführung einer kompletten Test-Suite im Optimalfall (kein Overhead, alle parallelen Ausführungen haben die gleiche Effizienz und keinen Einfluss aufeinander) mit 28 parallelen Test-Cases benötigt wird [49].

Dabei kann man erkennen, dass die Ausführungszeit durch die Ausreißer dominiert ist, der gesamte Test-Suite braucht stets so lange wie der längste Test-Case. Erst mit dem Weglassen von 11 Ausreißern wird der Test-Suite nicht mehr durch diese dominiert. Lässt

man die 11 Ausreißer weg und berechnet die optimale Laufzeit, erhält man 983 Sekunden, also ca. 16 Minuten.

Um diese vermutete These, dass die Laufzeit nur durch Ausreißer dominiert ist, zu überprüfen, wurde die Test-Suite nochmal ohne diese 11 Ausreißer Test-Cases ausgeführt und diese Zeit mit dem Optimum und der Ausführungszeit ohne „Torture Tests“ verglichen.

Dabei wurde eine Laufzeit von ca. 25 Minuten gemessen. Das Entspricht ca. eine 20,8-Fach längeren Laufzeit als die Ausführung ohne „Torture Test“ und ca. das 1,5-fache des berechneten Optimums.

Es folgt die Auswertung der „Torture Tests“ mit den entwickelten Werkzeugen:

Torture Tests	Zeilen abgedeckt	Zeilen gesamt	Prozent
Nein	6	139	4,32%
Ja	118	139	84,89%

Tabelle 4.1: Auswertung „Torture Tests“ isoliert auf Fehlerbehandlung von `malloc`, `realloc`, `calloc`, `strdup` und `socket`

Torture Tests	Zeilen abgedeckt	Zeilen gesamt	Prozent
Nein	25018	31251	80,06%
Ja	26233	31251	83,94%

Tabelle 4.2: Auswertung „Torture Tests“:
Projektweite Zeilenabdeckung

Aus den Messdaten wird erkennbar, dass die Abdeckung für die **Umgebungsaufrufe**, für die Fehler injiziert werden, massiv ansteigt. Auf die Gesamtabdeckung des Projektes fallen die Auswirkungen weniger dramatisch aus. Dabei ist aber anzumerken, dass das von „cURL“ eingesetzte Verfahren problemlos auf weitere **Umgebungsaufrufe** wie z. B. `clock_gettime`, `recvfrom` etc. ausgeweitet werden kann. Die so gewonnene Abdeckung ist wertvoll, vor allem da „cURL“ hauptsächlich End-to-End-Test des Programms benutzt. Für realistische Nutzungsszenarien ist durch „Torture Tests“ gezeigt, dass es bei fehlschlagenden **Umgebungsaufrufen** wahrscheinlich nicht zu Sicherheitsproblemen oder Ressourcenlecks kommt.

Der durch „Torture Tests“ erhöhte Entwicklungsaufwand ist begrenzt, da dieser nicht in Relation zu der Komplexität des restlichen Projekts steht. Erhöhter Aufwand beim Schreiben von Test-Cases besteht nicht. Neben dem einmaligen Entwicklungsaufwand ist die Laufzeit der Test-Suite ein Negativaspekt. Werden die Ausreißer nicht entfernt, ist die Ausführung der kompletten Test-Suite in **CI** fragwürdig, als zusätzliche Maßnahme bei Releases, vergleichbar mit Fuzz-Tests, aber durchaus denkbar. Werden Ausreißer entfernt oder Ursache für diese behoben, ist durchaus denkbar, die Nutzung von „Torture Tests“ auch bei der lokalen Entwicklung einzusetzen.

4.4 Ausblick und verwandte Arbeiten

Ein wichtiger Aspekt, der durch die „Torture Tests“ von „cURL“ belegt wird, ist, dass es in der Praxis möglich ist, Umgebungsfehlerfälle nach einer gewissen Definition erschöpfend zu testen, ohne dass es zu einer untragbaren Laufzeitexplosion kommt. Konkret wird für jeden Test-Case erschöpfend das Verhalten bei Auftritt genau eines Umgebungsfehlers getestet. Der gesteigerte Aufwand verhält sich linear zu Anzahl der **Umgebungsaufrufe** pro Test-Case. Es stellt sich die Frage, wie realistisch ein erschöpfendes Testen von mehr als einem Umgebungsfehler pro Test-Case ist. Die Antwort auf diese Frage hängt von der Beschaffenheit der Fehlerbehandlung ab. Angenommen jegliche Fehlerbehandlung beendet schlichtweg das Programm, d. h. bei Auftritt eines Umgebungsfehlers wird das Programm beendet, ohne weitere **Umgebungsaufrufe** zu machen. In diesem Fall kann es nie zu mehr als einem Umgebungsfehler pro Test-Case kommen. Das exhaustive Testen ist also bei Injektion von einem Umgebungsfehler für jeden **Umgebungsaufruf** abgeschlossen.

Zur weiteren Untersuchung lässt sich Fehlerbehandlung in zwei Klassen unterteilen:

1. Abbruch: Es wird nicht versucht, die normale Ausführung fortzusetzen.
2. Rettung: Es wird versucht, die normale Ausführung fortzusetzen.

Unter „Abbruch“ fällt beispielsweise das Verhalten von einem Compiler bei Syntaxfehlern. Die ausführbare Datei wird nicht erzeugt, sondern der Kompilationsprozess abgebrochen. Unter „Rettung“ fällt beispielsweise das Wiederversuchen (Retry) von Netzwerkaufrufen, da vermutet wird, dass es sich um transiente Fehler handelt. In vielen Szenarien ist Rettung begrenzt, so ist z. B. oft eine maximale Anzahl von Retries, bevor aufgegeben und von Rettung zu Abbruch übergegangen wird, festgelegt.

Für eine Fehlerbehandlung der 1. Klasse sollte meistens keine bis eine kleine konstante Zahl von noch folgenden **Umgebungsaufrufen** geben (weitaus weniger als bei der Ausführung ohne Umgebungsfehler). D. h. für jeden Test-Case, bei dem ein Umgebungsfehler (Klasse 1) injiziert wurde, gibt es nur noch wenige Möglichkeiten einen weiteren Umgebungsfehler zu injizieren. Gibt es nur Klasse-1-Fehlerbehandlungen, dann sollte die Anzahl der nötigen Ausführungen von Test-Cases bis zur erschöpfenden Abdeckung aller möglichen Umgebungsfehler mit einem Faktor (die noch verbleibenden **Umgebungsaufrufe** im Fehlerfall) linear zu der Anzahl der **Umgebungsaufrufe** im Test-Case sein. Für eine Fehlerbehandlung der 2. Klasse sollten nach der Fehlerinjektion die gleichen **Umgebungsaufrufe** liegen wie vor der Injektion. D. h. wird ein Fehler 2. Klasse injiziert, verdoppelt sich der restliche Testaufwand. Hier kommt es also offensichtlich bei vielen Fehlern 2. Klasse zu einer Laufzeitexplosion.

Es wird angenommen, dass die meisten Fehlerbehandlungen der 1. Klasse angehören. Dies könnte für das „cURL"-Projekt gezeigt werden, indem diese je nach Erfolg des Test-Cases bei Umgebungsfehler in Abbruch oder Rettung eingeordnet werden. Ebenso kann die Anzahl der **Umgebungsaufrufe**, die nach einem Umgebungsfehler ausgeführt werden, mit der Anzahl von **Umgebungsaufrufen** ohne Fehlerfall verglichen werden. Um dies zu überprüfen, müssen aber Codeänderungen an der „cURL"-Test-Suite vorgenommen werden. Daher bleiben diese Vermutungen in dieser Arbeit vorerst offen. Ist dies der Fall, gibt es möglicherweise viele Test-Cases, welche realistisch erschöpfend getestet werden können.

In der Literatur finden sich nur wenige Ansätze, die eine systematische „Fault Injection" anhand von Laufzeitinformation versuchen, konkret [15][80][39]. Stattdessen wird oft mit probabilistischer Injizierung oder systematischer Überprüfung anhand von statischen Informationen gearbeitet [45]. Diese Ansätze sind im Vergleich zu den neueren Ansätzen, die sich auf dynamische Instrumentierung stützen, weitaus weniger fruchtlos und werden hier nicht weiter untersucht. Interessanterweise datiert die Umsetzung von „Torture Tests" auf 2003 [35], also mehr als fünf Jahre älter als ähnliche Ansätze zur systematischen Durchsuchung aller möglichen Laufzeitfehler in der Forschung.

Die „Torture Tests" von „cURL" sind stark vergleichbar zur **Automatic Driver Fault Injection [15] (ADFI)** von Kai Cong et al. Dabei handelt es sich sogar um einen Spezialfall mit den Parametern $MF = 1$ und $MFS = \infty$, d. h. Anwendung von **ADFI** mit diesen Parametern führt zu der gleichen Ausführung wie sie durch „Torture Tests" erzeugt werden. Der Parameter „MF" gibt dabei an, wie viele Umgebungsfehler maximal simultan

in einem Trace injiziert werden sollen. Wird dieser beispielsweise auf 2 gesetzt, werden neben all den Ausführungen, die von „Torture Tests“ untersucht werden, auch Fehler in den Fehlerbehandlungspfaden injiziert usw. Der Parameter „MFS“ erlaubt es, gleiche **Umgebungsaufrufe**, welche auf der gleichen Aufrufebene passieren, zu limitieren. So ist es beispielsweise bei „cURL“ so, dass wiederholt **Umgebungsaufrufe** in Schleifen (Lesen von einem Socket) für eine große Anzahl von **Umgebungsaufrufen** sorgen. Von den „Torture Tests“ werden all diese getestet, was erheblich zu der Laufzeit der „Torture Tests“ beiträgt. **ADFI** arbeitet aber mit der Observation, dass Fehlerbehandlung in solchen Fällen meist gleichförmig ist. Hat man ein fehlschlagendes Lesen in einer Schleife abgedeckt, hat man damit vermutlich auch eine Aussage über weiter Aufrufe gemacht und testet auch keine neuen Fehlerbehandlungen. Eine Weiterentwicklung dieser Funktion, die jeweils die **Umgebungsaufrufe** in den Grenzen von Schleifen fehlschlagen lässt (erster Durchlauf, letzter Durchlauf und ein dazwischenliegender) und nicht einfach nur die ersten „MFS“ Durchläufe, ist vorstellbar.

In der wissenschaftlichen Arbeit zu **ADFI** [15] wird dieser mit den MF Parametern 1 bis 3 ausgeführt und die Anzahl der Permutationen für das exhaustive Testen bei 1, 2 und 3 Fehlern gemeldet:

Category	MF	Wireless Driver		USB Driver		Ethernet Driver							
		ath9k	iwlwifi	ehci_hcd	xhci_hcd	e100	e1000	ixgbe	i40e	tg3	bnx2	8139cp	r8169
PCI	1	1	3	0	0	2	5	5	5	7	3	2	2
	2	1	3	0	0	2	9	8	8	10	3	2	3
	3	1	3	0	0	2	9	9	8	10	3	2	3
Memory	1	5	24	4	1	3	13	11	32	11	9	1	3
	2	5	164	10	1	3	49	53	136	25	34	1	3
	3	5	840	12	1	3	117	156	414	29	51	1	3
DMA	1	3	4	1	6	5	11	9	17	4	13	3	8
	2	3	9	1	6	6	40	51	69	6	77	7	24
	3	3	10	1	6	6	95	171	177	6	221	8	37
ALL	1	9	31	5	7	10	28	25	54	22	25	6	13
	2	9	235	15	7	12	180	209	268	84	234	10	56
	3	9	1375	18	7	12	858	924	1365	175	980	11	130

Abbildung 4.1: Tabelle 4 aus [15], Nötige Permutation bei erschöpfender Suche von Tiefe MF pro Test-Case

In 4.1 lässt sich erkennen, dass die Vermutung, dass die meisten Fehler der Klasse 1 angehören und somit exhaustives Testen möglich ist, für manche Linux Treiber (ath9k, xhci_hcd, e100) gilt. Für die restlichen Werte lässt sich leider schwieriger die Ursache herleiten. Die nötigen Permutationen bis zum erschöpfenden Testen steigen pro Test-Case um ein Vielfaches an. Der Faktor, mit dem dies passiert, scheint aber in den meisten Fällen abzunehmen. Das entspricht nicht dem erwarteten Verhalten für Fehlerbehandlungen

der 2. Klasse. Es lässt sich vermuten, dass es sich hier um Aufräumarbeiten oder ähnliches in einer Fehlerbehandlung handelt, die selbst fehlschlagen kann.

Mit „Torture Tests“ und **ADFI** gibt es also mindestens zwei Verfahren, die eine begrenzte erschöpfende „Fault injection“ umsetzen. Beide Verfahren sind aber an ein Projekt gebunden (Linux und „cURL“), d. h. sie lassen sich nicht einfach auf andere Projekte übertragen. Leider wurden im Laufe dieser Arbeit keine generischen „Fault Injection“ Mechanismen gefunden, die ein solches Verfahren umsetzen. Dennoch gibt es generische Mechanismen für die „Fault Injection“ von Bibliotheksaufrufen wie beispielsweise libfiu [41] und LFI [46]. Diese Mechanismen unterstützen standardmäßig keine Suche, wie sie von „cURL“ und **ADFI** umgesetzt wird, lassen sich aber dahingehend erweitern.

5 Fazit

Zu Beginn dieser Arbeit wurde die Hypothese: „Codepfade, in denen Fehler der Laufzeitumgebung wie z. B. Lese-/Schreibfehler, abgebrochene Systemaufrufe, nicht genügend Arbeitsspeicher etc. behandelt werden, werden weniger durch Softwaretests abgedeckt als Codepfade, in denen die Fehler nicht behandelt werden.“ aufgestellt. Um diese Hypothese zu untersuchen, wurde Nomenklatur und ein allgemeines Modell für nichtdeterministisches Verhalten in Software gefunden. Die Hypothese wurde anhand dessen als eine klar beantwortbare Frage: „Werden Programmpfade, welche aufgrund von Umgebungsfehlern durchlaufen werden, weniger durch Tests abgedeckt als die durchschnittliche Zeilenabdeckung für das gesamte Programm?“ formuliert.

Die Frage wurde mit der Antwort „Ja“ für 4 von 4 untersuchten Softwareprojekten belegt. Daraus wird geschlossen, dass dies auch im generellen Fall zutrifft. Zur Überprüfung dieser Frage wurden eigene Werkzeuge entwickelt. Diese waren in der Lage, nicht nur Antworten auf die aufgeworfene Frage zu liefern, sondern auch Aspekte von allgemein mangelnder Fehlerbehandlung zu beleuchten. Die entwickelten Werkzeuge liefern so einen Mehrwert für eine Vielzahl von C-Projekten. Daher wird die Weiterentwicklung und der produktive Einsatz der Werkzeuge auch außerhalb dieser Arbeit angestrebt.

Schlussendlich wurde nach Lösungsaspekten gesucht, also Ansätze, welche die Antwort auf die Frage mit „Nein“ ausfallen lassen. Dabei wurden von dem aktuellen Stand der Forschung zu „Fault Injection“ sowie durch das „cURL“-Projekt entwickelte Testansätze betrachtet. Dabei handelt es sich um automatisierte Injizierung von Umgebungsfehlern. Diese Ansätze wurden von dieser Arbeit sowie dem Stand der Forschung als praktikabel bewertet und zeigen großes Potenzial. Leider gibt es momentan keine Werkzeuge, die eine einfache Implementation dieser Methoden in Softwareprojekten zulassen. Daher werden hier Weiterentwicklungen angestrebt, welche die Methoden auch in anderen Projekten einsetzbar machen.

Literatur

[1] *ABI Stable Symbols — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/v5.10/admin-guide/abi-stable.html (besucht am 11. 02. 2024).

[2] *AddressSanitizer — Clang 18.0.0git Documentation*. URL: https://clang.llvm.org/docs/AddressSanitizer.html (besucht am 07. 12. 2023).

[3] *Altio/Self_pipe.c (from "The Linux Programming Interface")*. URL: https://man7.org/tlpi/code/online/dist/altio/self_pipe.c.html (besucht am 01. 12. 2023).

[4] *Automake - GNU Project - Free Software Foundation*. URL: https://www.gnu.org/software/automake/automake.html (besucht am 17. 02. 2024).

[5] D. J. Bernstein. *The Self-Pipe Trick*. URL: https://cr.yp.to/docs/selfpipe.html (besucht am 01. 12. 2023).

[6] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments RFC 8259. Internet Engineering Task Force, Dez. 2017. 16 S. DOI: 10.17487/RFC8259. URL: https://datatracker.ietf.org/doc/rfc8259 (besucht am 04. 01. 2024).

[7] Peter M. Broadwell, N. Sastry und Jonathan Traupman. „FIG: A Prototype Tool for Online Verification of Recovery Mechanisms“. In: 2002. URL: https://www.semanticscholar.org/paper/FIG%3A-A-Prototype-Tool-for-Online-Verification-of-Broadwell-Sastry/1347b1ac269c47268ab7d3faeca16d41f944f024 (besucht am 25. 02. 2024).

[8] *C# | Modern, Open-Source Programming Language for .NET*. Microsoft. URL: https://dotnet.microsoft.com/en-us/languages/csharp (besucht am 11. 02. 2024).

[9] *Catalogue of Rules (GNU Make)*. URL: https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html (besucht am 17. 02. 2024).

[10] *CC — CMake 3.29.0-Rc1 Documentation*. URL: https://cmake.org/cmake/help/v3.29/envvar/CC.html (besucht am 17. 02. 2024).

[11] *Chapter 17. Threads and Locks*. URL: https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html (besucht am 08. 02. 2024).

[12] *Clang: Clang::Expr Class Reference*. URL: https://clang.llvm.org/doxygen/classclang_1_1Expr.html (besucht am 17. 02. 2024).

[13] *Comparisons*. URL: https://mesonbuild.com/Comparisons.html (besucht am 17. 02. 2024).

[14] *Conformance - Cppreference.Com*. URL: https://en.cppreference.com/w/c/language/conformance (besucht am 19. 02. 2024).

[15] Kai Cong u. a. „Automatic Fault Injection for Driver Robustness Testing“. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 13. Juli 2015, S. 361–372. ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771811. URL: https://doi.org/10.1145/2771783.2771811 (besucht am 24. 02. 2024).

[16] *CPP / C++ - Building Systems and Build Automation*. URL: https://caiorss.github.io/C-Cpp-Notes/building-systems.html (besucht am 17. 02. 2024).

[17] *Cpython/Python/Ceval_gil.c at A955fd68d6451bd42199110c978e99b3d2959db2 · Python/Cpython*. GitHub. URL: https://github.com/python/cpython/blob/a955fd68d6451bd42199110c978e99b3d2959db2/Python/ceval_gil.c#L979 (besucht am 07. 12. 2023).

[18] *Curl - Run Curl Tests*. URL: https://curl.se/dev/runtests.html (besucht am 23. 02. 2024).

[19] *CWE - 2023 CWE Top 10 KEV Weaknesses*. URL: https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html (besucht am 18. 02. 2024).

[20] *CWE - 2023 CWE Top 25 Most Dangerous Software Weaknesses*. URL: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html (besucht am 18. 02. 2024).

[21] *CWE - About - CWE Overview*. URL: https://cwe.mitre.org/about/index.html (besucht am 23. 02. 2024).

[22] *CWE - CWE-415: Double Free (4.13)*. URL: https://cwe.mitre.org/data/definitions/415.html (besucht am 19. 02. 2024).

[23] Thomas Dohmke. *100 Million Developers and Counting*. The GitHub Blog. 25. Jan. 2023. URL: https://github.blog/2023-01-25-100-million-developers-and-counting/ (besucht am 18. 02. 2024).

[24] *ERR33-C. Detect and Handle Standard Library Errors - SEI CERT C Coding Standard - Confluence*. URL: https://wiki.sei.cmu.edu/confluence/display/c/ERR33-C.+Detect+and+handle+standard+library+errors (besucht am 08. 02. 2024).

[25] *Errno - Cppreference.Com*. URL: https://en.cppreference.com/w/c/error/errno (besucht am 30. 12. 2023).

[26] *EXP12-C. Do Not Ignore Values Returned by Functions - SEI CERT C Coding Standard - Confluence*. URL: https://wiki.sei.cmu.edu/confluence/display/c/EXP12-C.+Do+not+ignore+values+returned+by+functions (besucht am 11. 02. 2024).

[27] *Flock(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/flock.2.html (besucht am 17. 02. 2024).

[28] *Fread - Cppreference.Com*. URL: https://en.cppreference.com/w/c/io/fread (besucht am 17. 02. 2024).

[29] *Fwrite - Cppreference.Com*. URL: https://en.cppreference.com/w/c/io/fwrite (besucht am 17. 02. 2024).

[30] *General Concepts*. URL: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_11 (besucht am 08. 02. 2024).

[31] *Git/Write-or-Die.c at C875e0b8e036c12cfbf6531962108a063c7a821c · Git/Git*. URL: https://github.com/git/git/blob/c875e0b8e036c12cfbf6531962108a063c7a821c/write-or-die.c#L43 (besucht am 11. 02. 2024).

[32] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Bd. 178. Prentice-hall Englewood Cliffs, 1985. URL: http://homepages.cs.ncl.ac.uk/brian.randell/Seminars/224.pdf (besucht am 22. 02. 2024).

[33] *Home | CVE*. URL: https://www.cve.org/ (besucht am 23. 02. 2024).

[34] „IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7“. In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (Jan. 2018), S. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153. URL: https://ieeexplore.ieee.org/document/8277153 (besucht am 07. 12. 2023).

[35] *Introducing -t to "Torture" the Memory Allocations/Failing/Bail-Outin... · Curl/Curl@b53a5e9*. GitHub. URL: https://github.com/curl/curl/commit/b53a5e92c0a91d063afc9d91461b69acd902834a (besucht am 25. 02. 2024).

[36] *ISO/IEC JTC1/SC22/WG14 - C*. URL: https://www.open-std.org/jtc1/sc22/wg14/ (besucht am 11. 02. 2024).

[37] „ISO/IEC/IEEE International Standard - Systems and Software Engineering–Vocabulary“. In: *ISO/IEC/IEEE 24765:2017(E)* (Aug. 2017), S. 1–541. DOI: 10.1109/IEEESTD.2017.8016712. URL: https://ieeexplore.ieee.org/document/8016712 (besucht am 11. 02. 2024).

[38] *Java Programming Language*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html (besucht am 11. 02. 2024).

[39] Zu-Ming Jiang u. a. „Fuzzing Error Handling Code Using {Context-Sensitive} Software Fault Injection“. In: 29th USENIX Security Symposium (USENIX Security 20). 2020, S. 2595–2612. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/jiang (besucht am 13. 10. 2023).

[40] *Known Exploited Vulnerabilities Catalog | CISA*. URL: https://www.cisa.gov/known-exploited-vulnerabilities-catalog (besucht am 23. 02. 2024).

[41] *Libfiu - Fault Injection in Userspace*. URL: https://blitiri.com.ar/p/libfiu/ (besucht am 14. 07. 2023).

[42] *Libuv/Src/Unix/Signal.c at 5e302730cd29cfeb15d32369dd2edfd9d3c82c11 · Libuv/Libuv*. GitHub. URL: https://github.com/libuv/libuv/blob/5e302730cd29cfeb15d32369dd2edfd9d3c82c11/src/unix/signal.c#L228C31-L228C44 (besucht am 07. 12. 2023).

[43] *Linux-Test-Project/Lcov: LCOV*. URL: https://github.com/linux-test-project/lcov (besucht am 04. 01. 2024).

[44] *Make - GNU Project - Free Software Foundation*. URL: https://www.gnu.org/software/make/ (besucht am 17. 02. 2024).

[45] Paul D. Marinescu und George Candea. „LFI: A Practical and General Library-Level Fault Injector“. In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. 2009 IEEE/IFIP International Conference on Dependable Systems & Networks. Juni 2009, S. 379–388. DOI: 10.1109/DSN.2009.5270313.

[46] Paul D. Marinescu und George Candea. „LFI: A Practical and General Library-Level Fault Injector“. In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. 2009 IEEE/IFIP International Conference on Dependable Systems & Networks. Juni 2009, S. 379–388. DOI: 10.1109/DSN.2009.5270313. URL: https://ieeexplore.ieee.org/abstract/document/5270313 (besucht am 13. 10. 2023).

[47] *Memory Model - Cppreference.Com*. URL: https://en.cppreference.com/w/cpp/language/memory_model (besucht am 08. 02. 2024).

[48] *Mocks Aren't Stubs*. martinfowler.com. URL: https://martinfowler.com/articles/mocksArentStubs.html (besucht am 21. 02. 2024).

[49] Lalla Mouatadid. „CSC 373 Summer 2016 - Week 9“. Lecture Notes. University of Toronto. URL: http://www.cs.toronto.edu/~lalla/373s16/notes/makespan.pdf (besucht am 21. 02. 2024).

[50] Roberto Natella u. a. „On Fault Representativeness of Software Fault Injection“. In: *IEEE Transactions on Software Engineering* 39.1 (Jan. 2013), S. 80–96. ISSN: 1939-3520. DOI: 10.1109/TSE.2011.124. URL: https://ieeexplore.ieee.org/abstract/document/6122035 (besucht am 25. 02. 2024).

[51] Robert O'Callahan u. a. „Engineering Record and Replay for Deployability“. In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '17. USA: USENIX Association, 12. Juli 2017, S. 377–389. ISBN: 978-1-931971-38-6.

[52] *Object-Oriented Design Patterns in the Kernel, Part 1 [LWN.Net]*. URL: https://lwn.net/Articles/444910/ (besucht am 24. 02. 2024).

[53] *Open(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/open.2.html (besucht am 17. 02. 2024).

[54] Carl Adam Petri. „Kommunikation Mit Automaten“. In: (1962). URL: https://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/ (besucht am 22. 02. 2024).

[55] *RAII - Cppreference.Com*. URL: https://en.cppreference.com/w/cpp/language/raii (besucht am 19. 02. 2024).

[56] *RAII - Rust By Example*. URL: https://doc.rust-lang.org/rust-by-example/scope/raii.html (besucht am 19. 02. 2024).

[57] *Redis CLI*. Redis. URL: https://redis.io/docs/connect/cli/ (besucht am 11. 02. 2024).

[58] *Rr: Lightweight Recording & Deterministic Debugging*. URL: https://rr-project.org/ (besucht am 01. 12. 2023).

[59] *Rust Programming Language*. URL: https://www.rust-lang.org/ (besucht am 11. 02. 2024).

[60] *SconsVsOtherBuildTools*. GitHub. URL: https://github.com/SCons/scons/wiki/SconsVsOtherBuildTools (besucht am 17. 02. 2024).

[61] *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*. 30. Juni 2016. URL: https://insights.sei.cmu.edu/library/sei-cert-c-coding-standard-rules-for-developing-safe-reliable-and-secure-systems-2016-edition/ (besucht am 11. 02. 2024).

[62] *Signal(7) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man7/signal.7.html (besucht am 24. 02. 2024).

[63] *Signalfd(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/signalfd.2.html (besucht am 01. 12. 2023).

[64] *Skalibs: The Selfpipe Library Interface*. URL: https://skarnet.org/software/skalibs/libstddjb/selfpipe.html (besucht am 01. 12. 2023).

[65] Ian Sommerville. *Software Engineering*. 10., aktualisierte edition. Hallbergmoos/-Germany: Pearson Studium, 1. Okt. 2018. 896 S. ISBN: 978-3-86894-344-3.

[66] *Standard C++*. URL: https://isocpp.org/ (besucht am 11. 02. 2024).

[67] *Standard Configuration Variables (Automake)*. URL: https://www.gnu.org/software/automake/manual/html_node/Standard-Configuration-Variables.html (besucht am 17. 02. 2024).

[68] Gernot Starke u. a. *iSAQB Glossary of Software Architecture Terminology*. Leanpub, 3. Juni 2016. URL: https://leanpub.com/isaqbglossary (besucht am 24. 02. 2024).

[69] *Stdin, Stdout, Stderr - Cppreference.Com*. URL: https://en.cppreference.com/w/c/io/std_streams (besucht am 11. 02. 2024).

[70] *Technical Q&A QA1118: Statically Linked Binaries on Mac OS X*. URL: https://developer.apple.com/library/archive/qa/qa1118/_index.html (besucht am 11. 02. 2024).

[71] *Test Double*. URL: https://martinfowler.com/bliki/TestDouble.html (besucht am 21. 02. 2024).

[72] *The Go Programming Language*. URL: https://go.dev/ (besucht am 11. 02. 2024).

[73] *The Go Programming Language Specification - The Go Programming Language*. URL: https://go.dev/ref/spec (besucht am 19. 02. 2024).

[74] *Throw Expression - Cppreference.Com*. URL: https://en.cppreference.com/w/cpp/language/throw#Stack_unwinding (besucht am 25. 02. 2024).

[75] *Torture*. URL: https://everything.curl.dev/internals/tests/torture (besucht am 18. 02. 2024).

[76] *Valgrind Home*. URL: https://valgrind.org/ (besucht am 07. 12. 2023).

[77] *Wayback Machine*. Top Github repositories by stars for the C language. 7. Dez. 2023. URL: https://web.archive.org/web/20231207221219/https://api.github.com/search/repositories?q=stars:%3E1%20language:c&sort=stars&order=desc&per_page=100 (besucht am 07. 12. 2023).

[78] *Windows Architecture*. 20. Feb. 2014. URL: https://learn.microsoft.com/en-us/previous-versions/cc768129(v=technet.10) (besucht am 11. 02. 2024).

[79] Austin Wright u. a. *JSON Schema: A Media Type for Describing JSON Documents*. Internet Draft draft-bhutton-json-schema-01. Internet Engineering Task Force, 10. Juni 2022. 78 S. URL: https://datatracker.ietf.org/doc/draft-bhutton-json-schema-01 (besucht am 17. 02. 2024).

[80] Pingyu Zhang und Sebastian Elbaum. „Amplifying Tests to Validate Exception Handling Code“. In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012 34th International Conference on Software Engineering (ICSE). Juni 2012, S. 595–605. DOI: 10.1109/ICSE.2012.6227157. URL: https://ieeexplore.ieee.org/abstract/document/6227157 (besucht am 25. 02. 2024).

A Anhang

A.1 Schemata

Listing A.1: Eingabeschema **error-analyzer**

```
{
  "type": "array",
  "items": {
    "type": "object",
    "required": ["name"],
    "properties": {
      "name": {"type": "string"},
      "error": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "operator": {
              "enum": [
                "SMALLER"      , "LARGER"       , "EQUAL"        ,
                "NOT_EQUAL"    , "SMALLER_EQUAL", "LARGER_EQUAL"
              ]
            },
            "value": {"type": "integer"}
          }
        }
      },
      "errno": {"description": "RESERVED, ignored for now"},
      "comment": { "type": ["string", "null"] },
      "tag": {
        "type": "array",
        "items": {
          "anyOf": [ {"type": "string"}, {"const": "unsupported"} ]
        }
      },
```

```
      "exceptions": {"description": "RESERVED, ignored for now"}
    }
  }
}
```

Listing A.2: Ausgabegabeschema **error-analyzer**

```
{
  "type": "array",
  "items": {
    "type": "object",
    "required": ["func", "callSiteLoc", "type"],
    "properties": {
      "func": {"type": "string"},
      "callSiteLoc": {"$ref": "#/$defs/sourceLocation"},
      "type": {
        "enum": [
          "UNSUPPORTED_FUNCTION"         ,
          "UNHANDLED_RETURN_VALUE"       ,
          "ERROR_PASSED_AS_RETURN"       ,
          "ERROR_PASSED_TO_FUNCTION"     ,
          "ERROR_STORED_IN_VARIABLE"     ,
          "ERROR_HANDLED_IN_IF_STATEMENT"
        ]
      },
      "consumerDump": {"type": "string"}
    },
    "allOf": [
      {
        "if": {
          "properties": {
            "type": {"const": "ERROR_STORED_IN_VARIABLE"}
          }
        },
        "then": {
          "required": ["varLoc", "handler"],
          "properties": {
            "varLoc": {"$ref": "#/$defs/sourceLocation"},
            "handler": {
              "oneOf": [
                {"$ref": "#/$defs/handler"                },
                {                             "type": "null"}
              ]
            }
          }
```

```
        }
      },
      {
        "if": {
          "properties": {
            "type": {"const": "ERROR_HANDLED_IN_IF_STATEMENT"}
          }
        },
        "then": {
          "required": ["handler"],
          "properties": { "handler": {"$ref": "#/$defs/handler"} }
        }
      }
    ]
  },
  "$defs": {
    "sourceLocation": {"type": "string", "pattern": ":\\d+:\\d+$"},
    "handler": {
      "type": "object",
      "required": ["type"],
      "properties": {
        "type": { "enum": ["KNOWN", "UNKNOWN"] }
      },
      "allOf": [
        {
          "if": {
            "properties": { "type": {"const": "UNKNOWN"} }
          },
          "then": {
            "properties": { "exprDump": {"type": "string"} },
            "required": ["exprDump"]
          }
        },
        {
          "if": {
            "properties": { "type": {"const": "KNOWN"} }
          },
          "then": {
            "required": [
              "evaluatesOnError",
              "expr"            ,
              "handlerLocStart" ,
              "handlerLocEnd"   ,
              "otherLocStart"   ,
```

```
            "otherLocEnd"
          ],
          "properties": {
            "expr": {"type": "string"},
            "evaluatesOnError": {
              "enum": ["TRUE", "FALSE", "UNKNOWN"]
            },
            "handlerLocStart": {
              "oneOf": [
                {"$ref": "#/$defs/sourceLocation"                },
                {                                    "type": "null"}
              ]
            },
            "handlerLocEnd": {
              "oneOf": [
                {"$ref": "#/$defs/sourceLocation"                },
                {                                    "type": "null"}
              ]
            }
          },
          "otherLocStart": {
            "oneOf": [
              {"$ref": "#/$defs/sourceLocation"                },
              {                                    "type": "null"}
            ]
          },
          "otherLocEnd": {
            "oneOf": [
              {"$ref": "#/$defs/sourceLocation"                },
              {                                    "type": "null"}
            ]
          }
        }
      }
    ]
  }
 }
}
```

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original