Ostfalia
Hochschule für angewandte Wissenschaften

Fakultät Elektro- und Informationstechnik

Tim Klamt

Anwendungstechniken von modernen Fuzz-Tests auf Softwaresysteme

Abschlussarbeit zur Erlangung des Hochschulgrades
Bachelor of Engineering (B.Eng.)

im Studiengang Wirtschaftsingenieurwesen Elektro- und Informationstechnik

an der Ostfalia Hochschule für angewandte Wissenschaften
– Hochschule Braunschweig/Wolfenbüttel

Erster Prüfer: Prof. Dr.-Ing. F. Büsching
Zweiter Prüfer: B. Sc. Lucas Harms

Eingereicht am: 2024-07-14

Diese Arbeit wurde im Rahmen einer Kooperation mit der Siemens Mobility GmbH erstellt.

Autor

Tim Klamt
Matrikelnummer
Studiengang: Wirtschaftsingenieurwesen Elektro- und Informationstechnik
Studienrichtung: -

Erstprüfer

Prof. Dr.-Ing. Felix Büsching
Fakultät Elektro- und Informationstechnik
Ostfalia Hochschule für Angewandte Wissenschaften – Hochschule Braunschweig/Wolfenbüttel
Salzdahlumer Straße 46/48
38302 Wolfenbüttel

Zweitprüfer

B. Sc. Lucas Harms
Siemens Mobility GmbH
Ackerstraße 22
38126 Braunschweig

Bearbeitungszeitraum

Beginn: 2024-05-14, Ende: 2024-08-14

Erklärung

(ggf. Erklärung gem. §22, Nr. 8, BPO 2013)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Ort/Datum eigenhändige Unterschrift

Abstract

Fuzz-Tests haben sich zu einem wichtigen Werkzeug entwickelt, um Schwachstellen in Softwaresystemen frühzeitig entdecken zu können. Durch die Entdeckung von prominenten Fehlern, wie dem Heartbleed-Bug, erfährt diese Testtechnik eine steigende Beliebtheit. Dies hat zu der Entwicklung von immer fortschrittlicheren Fuzzern und Fuzz-Techniken geführt. Diese Arbeit betrachtet Fuzzing vor dem Hintergrund der kommerziellen Softwareentwicklung. Dabei wird diese Technik im Kontext der Theorie von Softwaretests eingeordnet und die Frage untersucht, wie sie in den Testprozess integriert werden kann. Es werden verschiedene Formen von Fuzz-Tests vorgestellt und dabei besonders die Sachverhalte im Detail erklärt, die in vielen Dokumentationen nicht erklärt oder als gegebenes Wissen angenommen werden. Auf diese Weise soll dem Leser der Übergang von einem Einsteiger zu einem erfahrenen Anwender erleichtert werden. Im Anschluss wird die Funktionsweise gängiger Sanitizer erklärt, welche häufig mit Fuzz-Tests kombiniert werden. Abschließend wird Structure-Aware Fuzzing, durch Kombination von libFuzzer mit Googles Protocol Buffern, als vielversprechende Technik vorgestellt und untersucht.

Inhaltsverzeichnis

Abkürzungsverzeichnis

San	Sanitizer
ASan	Address-Sanitizer
MSan	Memory-Sanitizer
UBSan	Undefined-Behavior-Sanitizer
IDE	Integrated Development Environment
protobuf	Protocol Buffer
LPM	libprotobuf-mutator
CFG	Control Flow Graph

Symbolverzeichnis

T_{gesamt}	Anzahl der insgesamt durchgeführten Testfälle

1 Einleitung

Fuzzing ist der Oberbegriff für eine Reihe von Techniken zum Testen von Software, die in ihrer einfachsten Form, als „breaking things with random inputs" beschrieben werden kann. Bei einem Fuzz-Test wird ein Testobjekt wiederholt mit einer Vielzahl zufällig generierter oder mutierter Inputs aufgerufen, um zu testen, ob einer dieser Inputs einen Absturz zur Folge hat. Hierbei erfolgt in der Regel eine Kombination mit dem Einsatz von Codesanitizern. Dies sind Tools zur Fehlererkennung, die ein Testobjekt bei verschiedenen unerwünschten Verhaltensweisen abstürzen lassen. Seit der Einführung von Fuzz-Tests durch Miller et al. [1] im Jahr 1990, hat sich diese Testtechnik stetig weiterentwickelt. Besonders nach der Entdeckung des sog. Heartbleed-Bugs [2] im Jahr 2014, hat die Popularität dieser Technik stark zugenommen. Dies hat dafür gesorgt, dass seitdem immer fortschrittlichere Arten von Fuzz-Tests entwickelt wurden. Mittlerweile sind eine Vielzahl verschiedener Fuzzer frei verfügbar und der Begriff Fuzzing steht repräsentativ für die vielen filigraneren Techniken, die seitdem daraus abgeleitet wurden.
Beginnt man sich mit dem Thema zu beschäftigen, so stellt man fest, dass im Bereich der verfügbaren Informationen eine große Lücke zwischen den Einsteigern und den sehr erfahrenen Anwendern oder Entwicklern von Fuzzern klafft. Einfache Tutorials oder Anleitungen, wie man einen Fuzz-Test ausführen kann, sind weit verbreitet. Ihr Informationsgehalt ist jedoch begrenzt und es wird in keiner Weise erklärt, wie die im Hintergrund ablaufenden Schritte im Detail funktionieren. Demgegenüber stehen die Informationen, welche von den erfahrenen Anwendern oder den Entwicklern von Fuzzern bereitgestellt werden. Diese nehmen viele der Details als gegebenes Wissen an und erklären diese ebenfalls nicht. Der Übergang vom Einsteiger zu einem erfahrenen Anwender/Entwickler ist dadurch nur schwer möglich. Dies hat unter anderem zur Entstehung von „The Fuzzing Book" [3] geführt. Die Autoren versuchen verschiedene Fuzz-Techniken Schritt für Schritt zu erklären und in Python nachzubauen. Das Problem ist allerdings, dass auch dabei der Transfer des Wissens auf die Funktionsweise von gängigen Fuzzern wie libFuzzer oder AFL/AFL++ nicht gegeben ist. Fragen, wie ein Fuzzer das Feedback zur Codeabdeckung erhält, oder wie Sanitizer im Detail funktionieren, bleiben offen.
Ziel dieser Arbeit ist es, einen Beitrag zu leisten, um die Lücke zwischen Einsteigern und erfahrenen Anwendern/Entwicklern zu schließen. Hierfür soll zunächst die allgemeine Theorie zum Testen von Software betrachtet werden und Fuzzing in diesem Bereich eingeordnet werden. Anschließend soll auf verschiedene Arten von Fuzz-Tests eingegangen werden. Es soll ein Überblick über verschiedene Techniken und deren Funktionsweise gegeben werden. Dabei soll auf die oft unbeantworteten Fragen eingegangen werden und erklärt werden, wie bestimmte Funktionen von gängigen Fuzzern im Detail funktionieren. In ähnlicher Weise soll auf die Funktionsweise von Codesanitizern eingegangen werden. Zum Schluss wird der Einsatz von Structure Aware Fuzzing mit Protocol Buffern als vielversprechende Technik vorgeführt und untersucht. Dem Leser soll es dadurch möglich sein, derartige Fuzz-Tests zu verstehen und anwenden zu können. Alle Bewertungen und Untersuchungen finden stets vor dem Hintergrund der kommerziellen Softwareentwicklung statt. Es wird sich insgesamt mit der Frage beschäftigt, ob und wie sich Fuzz-Tests als ein fester Bestandteil des Testprozesses, in der kommerziellen Softwareentwicklung integrieren lassen.

2 Software Testing

Das folgende Kapitel soll eine allgemeine Einführung in die Theorie zum Testen von Software geben und dabei insbesondere die Bereiche aufgreifen, die im Zusammenhang mit Fuzzing stehen. Dies soll helfen, um Fuzzing in diesem Bereich richtig einordnen zu können. Es wird herausgestellt, wieso der Einsatz von Fuzz-Tests als ergänzende Technik in die Softwareverifikation Sinn ergibt. Darüber hinaus werden einige grundlegende Begriffe und Techniken erklärt, die im weiteren Verlauf dieser Arbeit Anwendung finden.

2.1 Allgemeines

Software gehört zu den komplexesten und variabelsten Produkten, die hergestellt werden. Im Gegensatz zu Produkten, die in Serie gefertigt werden, ist es bei der Entwicklung von Software nicht möglich, ein standardisiertes Testverfahren mit vordefinierten Tests und Analysen auf alle hergestellten Artefakte zu übertragen. Die Erstellung von Software erfordert immer eine individuelle Verifikation, deren Schwierigkeit mit Komplexität und Variabilität des Produktes zunimmt. Häufig übersteigen die Kosten der Verifikation die Hälfte der Gesamtkosten von Softwareentwicklung und -wartung, weshalb diesem Bereich besonders im kommerziellen Umfeld ein großer Fokus zukommt. Die Auswahl geeigneter Teststrategien ist hierbei entscheidend, um die richtige Qualität des Softwareproduktes sicherzustellen und dabei den Rahmen bestehender Kostenvorgaben nicht zu überschreiten. Im Laufe der Zeit sind verschiedene Techniken zur Verifikation von Softwareprodukten entwickelt worden, diese setzen an verschiedenen Stellen (z.B. unterschiedliche Fehlerklassen) an, um die erforderlichen Qualitätsstandards sicherzustellen. Dabei kann keine Test- oder Analysetechnik allen Zwecken dienen, sodass eine produktspezifische Kombination verschiedener Techniken erforderlich ist. Verschiedene Techniken kommen dabei mit unterschiedlichen Tradeoffs und haben oft einander ergänzende Stärken und Schwächen. Die Wahl, welche Techniken zum Einsatz kommen, ist letztendlich vom Produkt, den einzuhaltenden Qualitäts- und Sicherheitsstandards, den Kosten und Ressourcen, sowie dem Zeitplan abhängig. [4]

Die Gründe für die Kombination verschiedener Techniken:

- Effektivität für unterschiedliche Fehlerklassen
- Anwendbarkeit an verschiedenen Zeitpunkten des Projektes
- Unterschiedliche Zwecke, z.B. Unit-Test, Integrations-Test, System-Test
- Wirtschaftlichkeit, teure Testverfahren nur wo nötig

Neben der Frage welche Techniken sich am besten eignen, stellt sich auch die Frage über den Aufwand, der zum Testen der Software betrieben werden sollte. Dabei ist das Testen von Software grundsätzlich eine risikobasierte Aktivität. Sie liefert den Stakeholdern eine Aussage über die Qualität des Produktes und bildet damit die Grundlage für die Verbesserung der Qualität. Da es nicht möglich

ist jeden einzelnen Fall zu testen, ist es die Aufgabe des Softwaretesters, die Menge aller möglichen Tests auf eine realistische und sinnvolle Teilmenge zu reduzierten. Basierend auf den Risiken und dem Einsatzzweck muss hierbei eine Entscheidung getroffen werden, was wichtig zu testen ist und was nicht. Grundsätzlich gibt es hierbei eine optimale Menge an Testaufwand, die sich aus der Beziehung zwischen der Anzahl an gefundenen Fehlern und den dafür anfallenden Kosten ergibt. Diese Beziehung ist in Abbildung 2.1 dargestellt. [5]

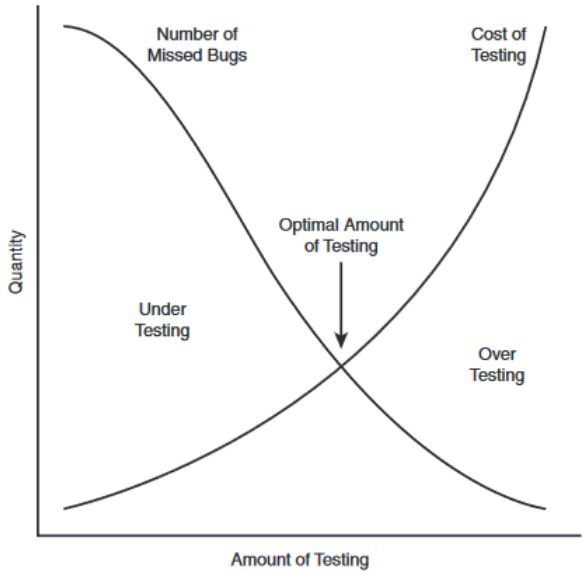

Abbildung 2.1: Ermittlung des optimalen Test-Aufwands [5]

An dieser Stelle zeigt sich, dass es beim Testen von Software darauf ankommt ein optimales Zusammenspiel aller Faktoren zu schaffen. Es gilt eine geeignete Kombination von Techniken einzusetzen. Basierend auf den Risiken muss der Tester eine kluge Entscheidung treffen, was in welchem Ausmaß zu testen ist. Auf diese Weise sollen bei möglichst niedrigen Kosten möglichst viele Fehler entdeckt werden. Dadurch entsteht ein Bedarf an immer leistungsfähigeren Testtechniken. Dies hat Fuzzing als eine Technik in den Fokus gebracht, deren Leistungsfähigkeit durch die Entwicklungen der letzten Jahre zunehmend gestiegen ist. Fuzz-Tests zeichnen sich durch ihren hohen Automatisierungsgrad, die große Anzahl an generierten Testfällen und die leistungsfähige Fehlererkennung in Kombination mit Sanitizern aus. Darüber hinaus gibt es mittlerweile viele fortschrittliche Fuzz-Techniken, die sich für verschiedene Arten von Testobjekten eignen. Im weiteren Verlauf dieser Arbeit soll aufgezeigt werden, wieso Fuzz-Tests ein großes Potenzial haben, den Test-Prozess von Software zu optimieren und damit in der Lage sein können, die Qualität der Software bei minimalem Kostenaufwand zu steigern.

2.2 Definitionen und Begriffe

Für den weiteren Verlauf dieser Arbeit ist es wichtig, eine klare Definition der verwendeten Begriffe zu schaffen. So beschreibt das Testen grundsätzlich einen Ansatz, bei dem überprüft wird, ob das Testobjekt einen Satz von definierten Anforderungen erfüllt. Ein Test wird ausgeführt, indem einzelne Testfälle mit dem Testobjekt durchgeführt und ihre Ergebnisse evaluiert werden. Der einzelne

Testfall besteht aus einem Satz von Inputdaten, Bedingungen, sowie den hierbei zu erwartenden Outputs bzw. Ergebnissen. Ein Testfall dient in der Regel dazu, die korrekte Umsetzung einer Teilmenge der aufgestellten Anforderungen zu belegen. [4] In Bezug auf Fuzzing sprechen wir also von einem Fuzz-Test, der eine Vielzahl von unterschiedlichen Testfällen generiert, um zu evaluieren, dass das Testobjekt mit den generierten Inputs kein unerwünschtes Verhalten zeigt.
Da die Entwicklung von Softwaresystemen ein vielstufiger Prozess ist, können Tests auf unterschiedlichen Ebenen des Entwicklungsprozesses eingesetzt werden. Hierfür haben sich unterschiedliche Entwicklungsmodelle etabliert, von denen das in Abbildung 2.2 dargestellte V-Modell allgemein bekannt ist. In diesem Modell wird jeder der Entwicklungsphase des absteigenden Zweiges eine Testphase im aufsteigenden Zweig gegenübergestellt.

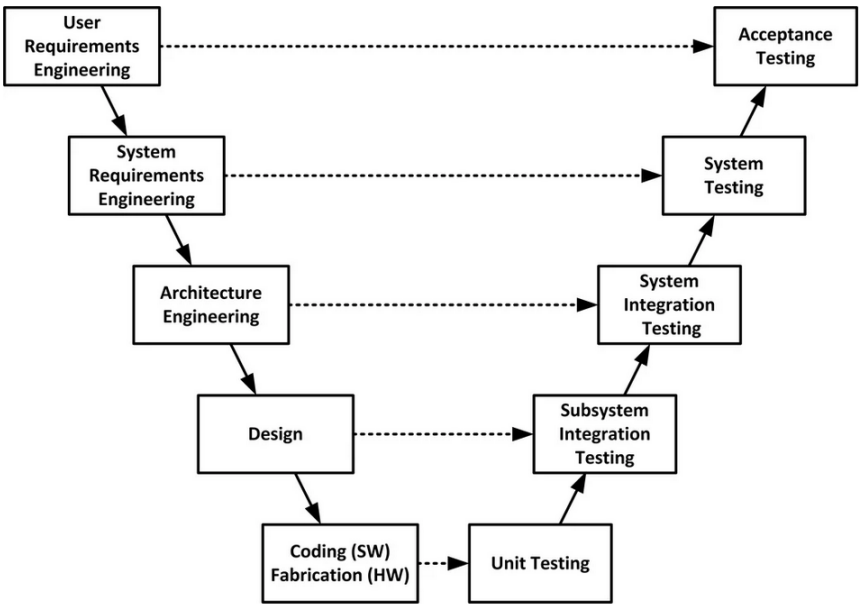

Abbildung 2.2: Das V-Modell bei der Enwicklung von Systemen [6]

Im Zuge der Überprüfungen im Entwicklungsprozess ist es wichtig, zwischen Verifikation und Validierung zu unterscheiden. Die Verifikation umfasst Techniken zur Überprüfung, ob die Entwicklungen der entsprechenden Stufe den dort festgelegten Spezifikationen entspricht. Die Validierung umfasst hingegen Überprüfungen, ob die festgelegten Spezifikationen dem grundsätzlichen Zweck des Produktes bzw. den Nutzeranforderungen entsprechen. Einfach gesagt bedeutet das: Verifikation überprüft, ob das System richtig gebaut wurde, während Validierung überprüft, ob überhaupt das richtige System gebaut wurde. In dieser Arbeit wird vom Testen im Sinne der typischen Verifikationstätigkeit gesprochen. In der Literatur kann der Begriff jedoch auch im Zuge der Validierung auftauchen, z.B. das Testen der Nützlichkeit des Produktes. [7][4]

2.3 Anforderungen in der Eisenbahn-Signaltechnik

Häufig definiert die Domäne, in der die Software Anwendung findet, welche Standards bei der Entwicklung und Verifikation mindestens eingehalten werden müssen. Da diese Arbeit bei der Siemens Mobility GmbH in der Stellwerksentwicklung erstellt wurde, gelten hier die in DIN EN 50129 festgelegten Anforderungen für die Anerkennung sicherheitsbezogener elektronischer Systeme im Bereich

der Eisenbahn-Signaltechnik [8]. Dort können die in Abbildung 2.3 dargestellten Anwendungsbereiche der wichtigsten Normen für Bahnanwendungen entnommen werden.

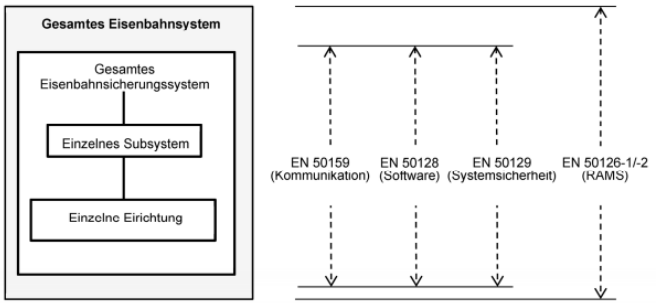

Abbildung 2.3: Anwendungsbereich der Normen für Bahnanwendungen [8]

Dabei wird bezüglich der Software auf die DIN EN 50128 verwiesen, welche die Anforderungen beschreibt, denen die Entwicklung, Bereitstellung und Wartung von sicherheitsrelevanter Software für Eisenbahnsteuerungs- und Überwachungsanwendungen entsprechen muss. Es findet eine Unterteilung in fünf verschiedene Software-Sicherheits-Integritätslevel statt (SIL 0 - SIL 4), wobei SIL 4 das höchste Sicherheits-Integritätslevel darstellt. Zu Verifikation und Testen findet sich im Anhang dieser Norm die Übersicht aus Abbildung 2.4, wobei in der Stellwerksentwicklung SIL 4 eingehalten wird. M steht dabei für Mandatory, R für Recommended und HR für Highly Recommended. [9]

Tabelle A.5 – Verifikation und Testen (6.2 und 7.3)

TECHNIK/MASSNAHME	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Formaler Beweis	D.29	–	R	R	HR	HR
2. Statische Analyse	Tabelle A.19	–	HR	HR	HR	HR
3. Dynamische Analyse und Tests	Tabelle A.13	–	HR	HR	HR	HR
4. Metriken	D.37	–	R	R	R	R
5. Verfolgbarkeit	D.58	R	HR	HR	M	M
6. Softwarefehler-Auswirkungsanalyse	D.27	–	R	R	HR	HR
7. Testabdeckung für Code	Tabelle A.21	R	HR	HR	HR	HR
8. Funktions-/Black-Box-Tests	Tabelle A.14	HR	HR	HR	M	M
9. Leistungstests	Tabelle A.18	–	HR	HR	HR	HR
10. Schnittstellentests	D.34	HR	HR	HR	HR	HR

Forderungen:

1) Für Software-Sicherheits-Integritätslevel 3 oder 4 ist die genehmigte Kombination von Techniken: 3, 5, 7, 8 und eine von 1, 2 oder 6.

2) Für Software-Sicherheits-Integritätslevel 1 oder 2 ist die genehmigte Kombination von Techniken: 5 gemeinsam mit einer von 2, 3 oder 8.

ANMERKUNG 1 Die Techniken/Maßnahmen 1, 2, 4, 5, 6 und 7 sind für Verifikationstätigkeiten.

ANMERKUNG 2 Die Techniken/Maßnahmen 3, 8, 9 und 10 sind für Testtätigkeiten.

Abbildung 2.4: Verifikation und Testen nach EN50128 [9]

Die detaillierte Auseinandersetzung mit den für SIL 4 nötigen Anforderungen würde den Rahmen dieser Arbeit überschreiten und in viele Punkte sind diese von der Begründung der eingesetzten Techniken im Verifikationsbericht abhängig. Grundsätzlich lässt sich Fuzzing hierbei als ergänzende Technik zur allgemeinen Verbesserung der Softwarequalität einordnen. Je nach eingesetzter Methode beim Fuzzing, werden unterschiedliche der genannten Techniken angeschnitten. So kann Fuzzing als

Black-Box Test oder als White-Box Test durchgeführt werden. Beim Einsatz von Coverage-Guided Fuzzing als White-Box Test spielt die Testabdeckung des Codes eine zentrale Rolle.
Für die Ergänzung der Schnittstellentests eignen sich Fuzz-Tests besonders gut. Ziel der Schnittstellentests ist die Demonstration, dass die Schnittstellen von Unterprogrammen keine Fehler enthalten, die zu Ausfällen in einer speziellen Anwendung der Software führen. Ebenso die Aufdeckung aller Fehler, die wichtig sein können. Hierzu werden die folgenden Stufen beschrieben, welche gut mit Fuzz-Tests abbildbar sind [9, S.113]:

- alle Schnittstellenvariablen mit ihren Extremwerten
- alle Schnittstellenvariablen einzeln mit ihren Extremwerten
- den gesamten Wertebereich jeder Schnittstellenvariablen
- alle Werte aller Variablen in Kombination

Der große Nachteil von Fuzz-Tests ist diesbezüglich, dass die Generierung von Testfällen in vielen Fuzz-Techniken auf zufälligen Mutationen beruht. Dadurch ist es schwierig, die Abdeckung der geforderten Verifikationstechniken vollständig und reproduzierbar nachzuweisen. In diesem Punkt liegt jedoch auch der Vorteil von Fuzz-Tests als Ergänzung zur manuellen Erstellung von Testfällen. Der Entwickler implementiert in der Regel nur eine angemessene Anzahl von Testfällen, um dadurch die Einhaltung der Anforderungen zu demonstrieren. Hierbei kann es passieren, dass Extremfälle oder Fehlerquellen beim Implementieren der Testfälle übersehen werden. Im Gegensatz dazu generiert ein Fuzz-Test in kürzester Zeit Millionen verschiedener Testfälle. Aufgrund der Anzahl und Willkür der generierten Testfälle werden so häufig Extremfälle und Fehlerquellen aufdecken, die vom Entwickler nicht bedacht wurden. Eine Ergänzung des Fuzz-Tests um den Einsatz von Sanitizern macht diesen Ansatz umso effektiver.

2.4 Ablauf von Unit-Tests - Eine realistische Betrachtung

Während in der Literatur vor allem ein sehr strukturierter Testprozess mit einer klaren Abgrenzung der einzelnen Techniken beschrieben wird, sieht die Realität des einzelnen Entwicklers häufig anders aus. Für ein möglichst realistisches Bild soll hierfür der Unit-Test betrachtet werden. Wie im V-Modells aus Abbildung 2.2 dargestellt, handelt es sich bei Unit-Tests um das Testen von einzelnen Softwarebausteinen auf der niedrigsten Stufe [5]. In der Regel bedeutet dies, dass ein Entwickler einen kleinen Programmbaustein wie eine Funktion oder Klasse implementiert und anschließend die dazugehörigen Tests entwickelt. Aus Kostengründen handelt es sich auf dieser Stufen nur in den seltensten Fällen bei Entwickler und Tester um zwei verschiedene Personen. Der Grund hierfür ist, dass es dem Entwickler nach der Implementierung besonders leicht fällt, die Tests für den zuvor geschriebenen Code zu entwickeln.
Neben besonderer Vorgaben kommt an dieser Stelle häufig eine Kombination aus White-Box Testing (oft auch Structural Testing genannt), Functional Testing und Requirement-Based Testing zum Einsatz. Basis für die Implementierung ist in der Regel eine funktionale Spezifikation des Programmverhaltens. Bei kleineren Artefakten eines Programmes ergibt sich diese funktionale Spezifikation häufig auch implizit aus dem Programm selbst. Dies ist zum Beispiel der Fall, wenn eine Hilfsfunktion zum Verarbeiten von Hexadezimalzahlen implementiert wird, damit sich die Vorgaben aus der formalen funktionalen Spezifikation des Programmverhaltens, im Anschluss leichter realisieren lassen. Grundsätzlich sind diese formalen oder informalen funktionalen Spezifikationen die wichtigste

Quelle für das Design der Tests. Hierbei beschreibt Functional Testing das Ableiten von Tests aus dieser funktionalen Spezifikation. Die Frage lautet: „Welche Testfälle sind zu implementieren, um zu zeigen, dass der Programmbaustein das gewünschte Verhalten zeigt" . In der Literatur wird Functional Testing ganz klar als Black-Box Test klassifiziert, bei dem die interne Struktur des Codes keine Rolle spielt, sondern ausschließlich die gegebenen Spezifikationen betrachtet werden. In der Realität wird diese Grenze jedoch nicht so klar eingehalten. Der Entwickler, der sowohl die Unit-Tests, als auch den zu testenden Code implementiert, wird zwangsläufig die interne Codestruktur in den Test des Programmverhaltens mit einbeziehen. [4]
Darüber hinaus wird bei den Unit-Tests typischerweise White-Box Testing explizit eingesetzt. Beim White-Box Testing wird neben der Funktionalität zusätzlich die interne Codestruktur des Programm-Bausteins getestet. Hierbei wird die interne Logik des Bausteins im Detail untersucht. Dazu können Tests aller Programmzweige, logischen Abläufe oder Schleifen gehören. Es wird beispielsweise untersucht, wie sich einzelne Programmzweige im Normalfall, bei Extremwerten oder Negativfällen verhalten. Außerdem wird über die Codeabdeckung häufig abgesichert, dass jede Zeile des Quellcodes innerhalb der Testfälle mindestens einmal durchlaufen wird. Für die effektive Ableitung der Testfälle im White-Box Testing kommen die Fähigkeiten des Entwicklers wesentlich mehr zum Tragen, als dies beim Functional Testing der Fall ist. [10] Des Weiteren lassen sich mit diesen Techniken unterschiedliche Fehlerklassen aufdecken. [4]
Neben funktionalen Anforderungen kann es nichtfunktionale Anforderungen geben. Dazu gehören explizite Anforderungen an die durchzuführenden Tests. Um die Abdeckung aller Anforderungen innerhalb der Entwicklung von Softwaresystemen nachweisen zu können, wird in der Regel eine Form von `Requirement Tracing` stattfinden. Typischerweise geschieht dies dadurch, dass die entsprechenden Stellen der Implementierung und des dazugehörigen Tests einer oder mehrerer Anforderungen innerhalb des Codes durch definierte Kommentare markiert werden. Jeder Anforderung wurde im Vorfeld ein eindeutiger Identifikator zugewiesen, welcher sich innerhalb des Kommentars wiederfinden muss. Durch spezielle Tools lässt sich der komplette Quellcode scannen, um so eine Matrix zu erstellen, mit der sich die Erfüllung jeder Anforderung nachverfolgen lässt. Aus diesem Grund beruht die Entwicklung von Unit-Tests zu einem großen Teil auf Requirement-Based Testing. [7]

2.5 Grenzen des Testens

Das Testen von Programmen als Verifikationstechnik zielt darauf ab, über `proof by cases` einen logischen Beweis zu liefern, dass das untersuchte Programm den spezifizierten Behauptungen standhält. Typischerweise handelt es sich bei diesen Behauptungen um einen Satz von definierten Anforderungen. Um hier einen absoluten logischen Beweis liefern zu können, müsste durch exzessives Testen jedes mögliche Verhalten des Programms untersucht werden. Dies ist jedoch praktisch unmöglich, da dieser Vorgang selbst bei simplen Programmen, in keiner absehbaren Zeit durchführbar ist. Betrachtet man in Listing 2.1 eine einfache Funktion zur Addition zweier Integer-Variablen. [4]

```
1 int sum(int a, int b) {return a+b;}
```

Listing 2.1: Simple Funktion, die zwei Integer addiert

Um jedes mögliche Verhalten dieses Programmes abzudecken, wären Testfälle mit $2^{32} \cdot 2^{32} = 2^{64}$ (32 Bit System) verschiedenen Inputs, also ca. $1{,}8 \cdot 10^{19}$ Testfälle nötig, um die Behauptung von Korrektheit beweisen zu können. Angenommen man würde einen Testfall pro Nanosekunde durchführen (10^{-9} s), dann würde der logische Beweis $1{,}8 \cdot 10^{10}$ Sekunden erfordern, was ca. 571 Jahren entspricht. Dies ist der Grund wieso man das Testen von Software als optimistische Technik bezeichnet.

Dabei ist keine endliche Anzahl an Tests in der Lage, absolute Korrektheit zu garantieren. Bei der Verifikation durch Testen wird also in Kauf genommen, dass Fälle existieren, für die das Programm weiterhin fehlerhaft ist. [4] Das bedeutet, dass ein Programm, das alle Tests erfolgreich durchläuft, nicht zwingend korrekt ist. Auf dieser Tatsache beruht das folgende Zitat von Dijkstra [11]:

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

Trotz dieser Limitierung in Bezug auf absolute Korrektheit, bleibt das Testen von Software ein wichtiger Baustein, um Fehler zu entdecken und so die Qualität der Software zu verbessern. Im Zuge dessen gibt es einige Grundsätze, die helfen können, mit diesem Sachverhalt besser umzugehen. Der erste lautet: „Je mehr Fehler man findet, desto mehr Fehler sind vorhanden." Dies klingt zuerst trivial, ist für den Entwickler von Tests jedoch ein wichtiger Grundsatz. Findet dieser beim Testen plötzlich in kurzer Zeit viele Fehler, dann sollte er diese Bereiche umso genauer untersuchen, da es sehr wahrscheinlich ist, dass dort weitere Fehler enthalten sind. Dies kann folgende Gründe haben:

- Programmierer können einen schlechten Tag haben
- Programmierer wiederholen häufig gleiche Fehler
- Einzelne Fehler können erste Spuren auf große strukturelle Fehler sein

Das Gleiche gilt an dieser Stelle auch für den umgekehrten Fall dieses Grundsatzes. Wenn der Tester trotz verstärkter Bemühungen keine Fehler finden kann, dann ist es umso wahrscheinlicher, dass der Code sauber geschrieben wurde. [5]
Ein weiterer Grundsatz wurde 1990 von Boris Beizer unter dem Begriff `Pestizid Paradoxon` eingeführt [12]. Mit dieser Analogie auf Pestizide wird das folgende Phänomen beschrieben. Für jede eingesetzte Methode, die dazu dient Fehler zu finden bzw. zu verhindern, bleibt ein Rest subtilerer Fehler, gegen die die eingesetzten Methoden ineffektiv sind. Das bedeutet, je mehr eine Software mit den gleichen Methoden getestet wird, desto „immuner" wird sie gegen diese Methoden. In der Realität ist das Testen von Software ein sich wiederholender Prozess. Nach einigen Iterationen ist es möglich, dass alle Fehler gefunden sind, zu deren Entdeckung die eingesetzten Techniken in der Lage sind. Damit weitere Fehler entdeckt werden können, müssen Tester neue Tests schreiben bzw. neue Techniken einsetzen, um somit das `Pestizid Paradoxon` zu überwinden.
Der letzte Grundsatz bezüglich der Grenzen von Tests lautet: „Nicht jeder gefundene Fehler wird auch behoben". In der Realität der Softwareentwicklung ist Perfektion an einigen Stellen nicht mit angemessenem Aufwand zu erreichen. Das Vorhandensein eines gefundenen Fehlers muss nicht zwangsläufig zu einem schlechten Produkt führen. Unter den gegebenen Ressourcen muss oft eine indiviuelle, risikobasierte Abschätzung getroffen werden, welche Fehler wann zu beheben sind und welche nicht. Die Entscheidung einen Fehler nicht zu beheben kann dabei die folgenden Gründe haben. [5]

- Nicht genug Zeit bzw. Ressourcen innerhalb des Projektes
- "It’s not a bug, it’s a feature!" - Ein fehlgeschlagener Test muss nicht zwangsläufig ein Fehler sein bzw. der Fehler muss sich nicht auf die Qualität des Produktes auswirken
- Die Fehlerbehebung ist zu riskant und kann neue Fehler erzeugen
- Der Aufwand ist es nicht wert - Der Fehler ist in einem selten genutzten Teil der Software oder es gibt bereits einen Workaround

2.6 Einordnung von Fuzzing

Nachdem nun viele theoretische Betrachtungen zum Testen von Software erläutert wurden, gilt es die Frage zu klären, wie sich Fuzzing Einordnen bzw. in einen Test- / Entwicklungsprozess integrieren lässt. Fuzzing als Testverfahren unter der Beschreibung - „breaking things with random inputs" - fällt in vielen Bereichen aus den konventionellen Vorgehensweisen für Softwaretests heraus. Grundsätzlich ist es eine von vielen Techniken zum Testen von Software, die sich unter dem Oberbegriff Fuzzing in eine Vielzahl filigranerer Techniken unterteilt. Auf einzelne Techniken von Fuzzing wird im weiteren Verlauf dieser Arbeit genauer eingegangen. Je nach eingesetzter Fuzz-Technik werden unterschiedliche Bereiche der Theorie des Testens von Software abgedeckt.
Was alle Fuzz-Techniken gemein haben ist, dass sie sich von konventionellen Ansätzen zum Testen von Software unterscheiden. Da die Generierung von Testfällen beim Fuzzing in vielen Fällen auf mehr oder weniger zufälligen Mutationen basiert, besteht bei Fuzz-Tests immer eine gewisse Willkür. Im Gegensatz zu den besprochenen konventionellen Testtechniken sorgt das dafür, dass sich mit Fuzz-Tests die explizite Erfüllung bestimmter Anforderungen nicht belegen lässt. Auch der konkrete Nachweis des korrekten Verhaltens eines Softwarebausteins entsprechend des Functional Testings gestaltet sich damit schwierig. In Bezug auf die in der Stellwerksentwicklung geltende SIL 4 Norm zeigt sich, dass Fuzzing alleine nicht in der Lage ist einzelne geforderte Vorgaben eindeutig zu erfüllen und dies wiederhol- und nachweisbar darzulegen. Wieso sollte man Fuzzing dann einsetzen?
Betrachtet man Fuzzing, dann ist diese Technik vermutlich am ehesten dem aus der Cybersecurity bekannten Penetration Testing zuzuordnen. Dabei versucht der Tester in Form eines simulierten Angriffs gezielt Schwachstellen in Computersystemen aufzudecken und diese auszunutzen [13]. Im Gegensatz zu konventionellen Tests wird beim Fuzzing getestet, ohne dabei den Eintritt eines im Vorfeld definierten Ergebnisses abzuprüfen. Stattdessen wird ähnlich wie beim Penetration Testing versucht das Testobjekt zu unerwünschtem Verhalten bzw. dieses zum Absturz zu bringen. In dieser komplett unterschiedlichen Herangehensweise im Vergleich zu konventionellen Testverfahren, liegt eine Stärke von Fuzzing. Im Hinblick auf das zuvor erläuterte `Pestizid Paradoxon` ist es besonders ratsam neue Tests durchzuführen, die sich von den bisher verwendeten Techniken maßgeblich unterscheiden. Auf diese Weise birgt der Einsatz von Fuzz-Tests das Potenzial neue Fehlerklassen zu entdecken, deren Entdeckung mit bisherigen Tests unmöglich war. Darüber hinaus eignen sich Fuzz-Tests durch die hohe Anzahl generierter Testfällen besonders gut, um sie mit Sanitizern als Tool zur Entdeckung weiterer Fehlerklassen zu kombinieren.
Da Fuzz-Tests nicht in der Lage sind andere Testverfahren vollständig zu ersetzen, kann Fuzzing ausschließlich als ergänzende Technik zu diesen angewandt werden. Das bedeutet wiederum, dass hierfür zwangsläufig zusätzliche Ressourcen aufgewendet werden müssen. Dies muss mit einer signifikanten Verbesserung der Softwarequalität in Abhängigkeit zu den dafür aufgewendeten Kosten gerechtfertigt werden. An dieser Stelle ist es vorteilhaft, dass viele Fuzzer kostenfrei und leicht zugänglich sind. Neben Clang sind LibFuzzer und Sanitizer bereits fester Bestandteil der LLVM-Compiler-Infrastructur [14]. Im Gegensatz zu dem dargestellten Ablauf von Unit-Tests sollten Fuzz-Tests im besten Fall von wenigen Testern mit tiefem Verständnis von Fuzzing implementiert werden. Der Grund hierfür ist, dass die Effektivität von Fuzz-Tests stark von den Fähigkeiten des Entwicklers abhängt. Auch wenn sich theoretisch jede einzelne Funktion fuzzen lässt, sollten besonders geeignete Stellen im Quellcode für den Einsatz von Fuzz-Tests ausgewählt werden. Dies ist bereits durch die Limitierung der Hardware bedingt, da Fuzz-Tests über längere Zeiträume durchgeführt werden. Wie sich im weiteren Verlauf dieser Arbeit zeigen wird, muss anschließend eine für das Testobjekt geeignete Fuzz-Technik ausgewählt werden und diese im besten Fall zusätzlich auf das Testobjekt angepasst werden. Sind diese Faktoren gegeben, dann lassen sich effektive Fuzz-Tests relativ schnell

und kostengünstig implementieren.
Sobald ein Fuzz-Test an der richtigen Stelle im Code implementiert ist, kommen weitere Stärken dieser Technik zum Tragen. So ist Fuzzing nach der Implementierung ein automatisiertes Testverfahren, das ohne menschlichen Eingriff eine Vielzahl neuer Testfälle generiert. Die aus zufälligen Mutationen resultierende Willkür dieses Verfahrens, welche zunächst wie ein Nachteil wirkt, ist dabei ein Vorteil. Zum einen unterscheidet sich die Ableitung von Testfällen damit grundlegend von dem Vorgehen eines menschlichen Testers. Viel wichtiger ist aber, dass bei jeder Iteration des Testprozesses immer einzigartige neue Testfälle generiert werden. Konventionelle Tests würden hingegen in jeder Iteration die exakt gleichen, fest programmierten Testfälle abprüfen. Im besten Fall hat man Fuzz-Tests daher einmalig für geeignete Stellen des Testobjektes implementiert. Solange sich die dabei genutzten Schnittstellen nicht ändern, muss die Implementierung des Fuzz-Tests nicht verändert werden. Wenn man anschließend die Durchführung der Fuzz-Tests als festen Bestandteil der CI/CD Pipeline integriert, dann werden automatisiert vor jedem Merge neue einzigartige Testfälle erzeugt und durchgeführt. Jede Iteration des Testprozesses besitzt damit das Potenzial neue Fehler zu finden. Wendet man Techniken wie das im nächsten Abschnitt vorgestellte Coverage-Guided Fuzzing an, dann ist der Fuzz-Test bei jeder Iteration des Testprozesses in der Lage im Testobjekt weiter vorzudringen. Voraussetzung dafür ist, dass die gesammelten Testdaten, also generierte und für interessant befundene Inputs, zwischen den Iterationen des Testprozesses aufbewahrt und weiterverwendet werden. Auf diese Weise haben Fuzz-Tests nicht nur das Potenzial mit jeder Iteration neue Fehler zu entdecken, sondern auch andere Stellen innerhalb des Testobjektes zu testen. Aus diesen Gründen ist es sinnvoll Fuzzing als Ergänzung zu konventionellen Testverfahren in der Softwareentwicklung einzusetzen.

3 Fuzzing

Fuzzing kann im Grunde als ein Prozess beschrieben werden, bei dem ein Programm wiederholt mit verschiedenen, generierten Inputs ausgeführt wird, die syntaktisch oder semantisch verformt sein können [15]. Unter den zahlreichen Techniken zum Testen von Software erfährt Fuzzing eine zunehmende Beliebtheit. Dies zeigt sich auch an der steigenden Anzahl an Publikationen zu diesem Thema, welche in Abbildung 3.1 dargestellt ist.

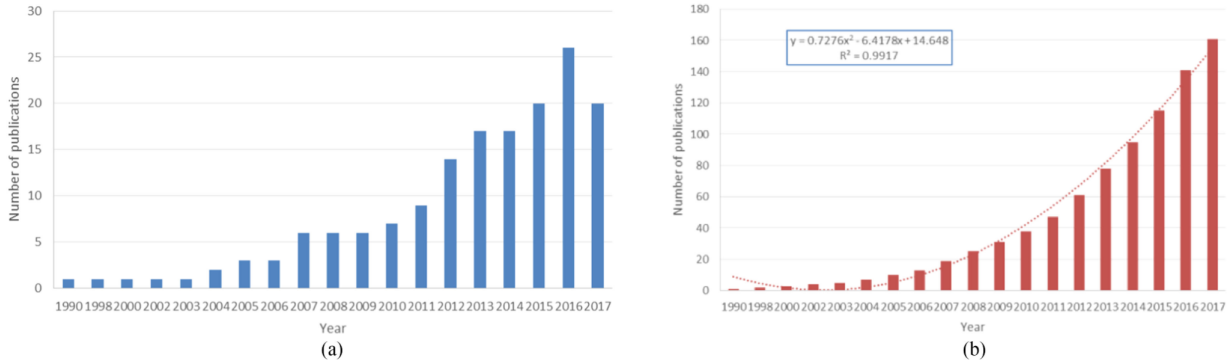

Abbildung 3.1: Zahl der veröffentlichten Paper zu Fuzzing zwischen 1990 und 2017, (a) pro Jahr und (b) kumuliert [16]

Sowohl die Forschung, als auch die Community der Anwender haben in den letzten Jahren viel Aufwand betrieben, um die Techniken von Fuzz-Tests zu verbessern. Dies hat dazu geführt, dass es schwer ist, einen Überblick über den aktuellen Stand der Entwicklung und Forschung zum Thema Fuzzing zu behalten. Dieses Kapitel dient dazu, einen Überblick über theoretisches Wissen zum Thema Fuzzing zu geben. Dieses Wissen bildet die Grundlage, um die Funktionsweise gängiger Fuzzer im Allgemeinen verstehen zu können. Basierend darauf, wird es möglich deren Verhalten auf die eigenen Bedürfnisse und das spezifische Testobjekt anzupassen.

3.1 Motivation

Seit der Einführung von Fuzz-Tests durch Miller et al. [1] im Jahr 1990 hat sich Fuzzing als eine effektive Technik zum Finden von Schwachstellen und Fehlern in Softwaresystemen etabliert. Besonders in der heutigen Zeit, wo viele Bereiche des täglichen Lebens auf der Anwendung von Software basieren, nimmt Softwaresicherheit einen immer höheren Stellenwert ein. Schwachstellen innerhalb einer Software ermöglichen es potentiellen Angreifern diese auszunutzen und dadurch ernorme Schäden anzurichten [17]. Diesbezüglich betreibt das National Institut of Standards and Technology (NIST) eine umfassende Datenbank mit der die Details von gefundenen Schwachstellen entsprechenden Interessengruppen zum Zweck der Forschung und Diskussion bereitgestellt werden [18]. Das Auftreten von Schwachstellen in einer Software kann die folgenden Auswirkungen haben:

- Diebstahl von Daten
- Störungen des Betriebs
- Finanzielle Schäden und rechtliche Konsequenzen
- Schädigung des Firmenimages

Während eine Störung des Betriebs bei kritischer Infrastruktur wie dem Stromnetz oder öffentlichen Transportmitteln enorme Konsequenzen wie den Verlust von Menschenleben nach sich ziehen kann, so trifft die Firmen in der Regel der aus einer Sicherheitslücke resultierende Imageverlust am stärksten. Dies hat dazu geführt, dass Firmen wie Google sogenannte `Vulnerability Reward` Programme eingeführt haben [19]. Diese Programme animieren die Entwickler-Community dazu, gezielt nach Schwachstellen in der Software zu suchen. Entdecker von Schwachstellen erhalten hierbei eine monetäre Belohnung. Dieser Ansatz hat maßgeblich zu der Entwicklung und Beliebtheit von Fuzzing beigetragen. Die Gemeinschaft hat begonnen die Fuzz-Techniken zunehmend weiterzuentwickeln und mit diesen gezielt auf die Suche nach Schwachstellen in öffentlichen Repositories zu gehen. Das bekannteste Beispiel in diesem Kontext ist der sogenannte `HeartBleed` Bug. Hierbei handelte es sich um eine Sicherheitslücke innerhalb der OpenSSL Library, welche kryptografische Protokolle für die Netzwerkkommunikation implementiert. Die bekannteste Anwendung ist die Absicherung der Kommunikation in `HTTPS`. Die Schwachstelle lag in dem sog. Heartbeat Service, der in der Kommunikation überprüfen soll, ob der Kommunikationspartner noch aktiv ist. Der Client übergibt dem Service einen String und die Anzahl an Zeichen. Der Server antwortet anschließend mit dem exakt gleichen String. Die Schwachstelle bestand darin, dass man eine beliebige Zahl an Zeichen übergeben konnte, sodass der Server mit Daten antwortete, die im Speicher hinter dem String stehen. [2] Der Ablauf ist in Abbildung 3.2 veranschaulicht.

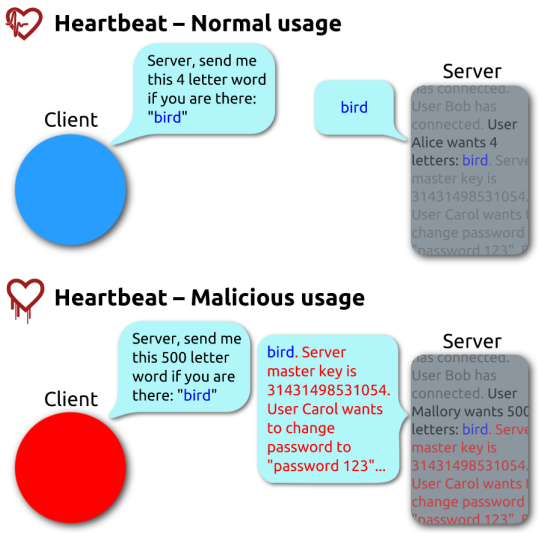

Abbildung 3.2: Simplifizierte Darstellung des Heartbleed Bugs [2]

Diese Sicherheitslücke bestand über Jahre und welche sensiblen Daten abgegriffen wurden lässt sich nicht sagen. Entdeckt wurde sie als ein Fuzz-Test mit Memory Sanitizern durchgeführt wurde. Dieser Test registrierte in kürzester Zeit, dass ein out-of-bounds Speicherzugriff stattgefunden hat. [3]

3.2 Klassifizierung

Die Techniken von Fuzz-Tests können grundlegend in die drei Kategorien `Black-Box`, `Grey-Box` und `White-Box` Fuzzing eingeteilt werden. Diese Klassifizierung wird anhand der Informationen, die dem Fuzzer während der Laufzeit zur Verfügung stehen, vorgenommen. [16] Das Black-Box Fuzzing ist eine Technik, die keinerlei Informationen vom oder über das Testobjekt verwendet. Stattdessen wird das Testobjekt blind mit zufällig generierten oder mutierten Daten ausgeführt. Hierbei wird nicht in Betracht gezogen, was innerhalb des Testobjektes passiert oder wie dieses reagiert. Diese Form des Fuzzings ist leicht zu implementieren aber in den meisten Fällen ineffektiv. Der Großteil der generierten Testfälle besteht aus invaliden Inputdaten und es ist auf diese Weise schwer, alle Codezweige innerhalb des Testobjekts zu erreichen. [20]
Um die Codeabdeckung zu erhöhen und die Generierung der Testfälle effektiver zu gestalten, kann man zum Grey-Box Fuzzing übergehen. Diese Kategorie von Fuzz-Tests zeichnet sich dadurch aus, dass hierbei zusätzliche (externe) Informationen über das Verhalten des Testobjekts in die Generierung der Testfälle mit einbezogen werden. Typischerweise werden hierbei Informationen zur Codeabdeckung innerhalb des Testobjektes durch Instrumentierung gewonnen. Dadurch lässt sich evaluieren, welche der generierten Inputs valide sind und mit welchen Inputs neue Codezweige im Testobjekt ausgeführt werden. Diese Informationen ermöglichen es dem Fuzzer, die Generierung der Testfälle auf die Maximierung der Codeabdeckung hin zu optimieren. Neben der Codeabdeckung können auch andere Informationen wie die CPU- oder Speicherauslastung verwendet werden. Der Fuzzer kann mit diesen Informationen die Generierung der Testfälle auf Inputs optimieren, die bei der Ausführung des Testobjektes zu einer besonders hohen Ressourcenverwendung führen. Das gleiche Prinzip wäre auch mit einer Überwachung der Ausführungszeit denkbar. [20]
Bei der letzten Kategorie des White-Box Fuzzings, wird zusätzlich der Quellcode des Testobjektes genutzt. Über den Quellcode werden detaillierte Informationen zum Verhalten des Testobjektes gewonnen, welche dann in die Generierung der Testfälle mit einbezogen werden können. In der ursprünglich präsentierten Form nach Godefroid et al. [21] wird dynamische symbolische Ausführung des Testobjektes genutzt, um so die interne Logik und damit theoretisch alle Programmpfade entdecken zu können [22]. So lässt sich überprüfen, ob tatsächlich alle Codezweige innerhalb des Testobjekts erreicht wurden. Beim Grey-Box Fuzzing kann hingegen nur festgestellt werden, welche Codezweige bereits erreicht wurden. Informationen über Zweige die nicht erreicht wurden, stehen beim Grey-Box Fuzzing nicht zur Verfügung [20]. Die Unterschiede zwischen den drei Kategorien sind in Abbildung 3.3 veranschaulicht.

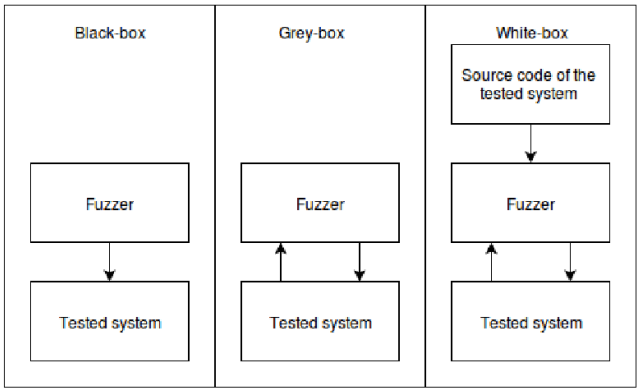

Abbildung 3.3: Übersicht der Klassifizierung von Fuzz-Techniken [20]

Abschließend sei darauf hingewiesen, dass die Klassifizierung in manchen Punkten nicht immer eindeutig ist und je nach Quelle variieren kann. Ein gutes Beispiel hierfür ist BuzzFuzz, dessen Technik als `taint-based directed whitebox Fuzzing` bezeichnet wird [23]. In anderen Arbeiten wird BuzzFuzz jedoch als Grey-Box Fuzzer klassifiziert, da bei dieser Technik nur unvollständiges Wissen über das Testobjekt generiert wird [16]. Die in dieser Arbeit betrachteten Fuzzer libFuzzer und AFL/AFL++ werden als Grey-Box Fuzzer klassifiziert [3].

3.3 Fuzzing mit zufälligen Inputs

Die einfachste Form eines Fuzzers ist eine Funktion, die einen String mit zufälligen Zeichen oder einen Strom von Bytes generiert und damit ein Testobjekt wiederholt aufruft. Dies lässt sich entsprechend Listing 3.1 sehr leicht implementieren. Vom Prinzip her entspricht dieses Beispiel bereits dem grundlegenden Vorgehen von libFuzzer für den Fall, dass kein (valider) initialer Input übergeben wird oder, dass der Fuzzer auf Feedback reagiert.

```
int runs = 1000; // Number of runs for the fuzz-test
std::srand(std::time(nullptr)); // time as seed, #include <ctime> necessary

for(int idx1 = 0; idx1 < runs; idx1++) {

    size_t size = rand() % 10 + 1 ; // Number of generated Bytes
    uint8_t *data = new uint8_t[size];

    for(int idx2 = 0; idx2 < size; idx2++) {
        data[idx2] = rand() % 256; // insert random Bytes
    }

    std::cout << "As a String:" << std::endl;
    std::cout << std::string(reinterpret_cast<const char*>(data), size)
              << std::endl;

    functionUnderTest(data, size); // call function under test

    delete[] data;
}
```

Listing 3.1: Implementierung eines simplen Fuzzers

Wie bereits angedeutet, haben moderne Fuzzer wesentlich mehr Funktionen als die Generierung von zufälligen Inputs. Auch wenn dieses Vorgehen bereits einem Fuzz-Test entspricht, hat die Implementierung viele Schwachstellen. Der Fuzzer reagiert in dieser Form auf keinerlei Feedback durch das Testobjekt. Sollte es zu einem Crash kommen, bricht das gesamte Programm ab. Darüber hinaus kommen hier keine Tools zum Erkennen bestimmter Fehlerklassen zum Einsatz. Auch für den Fall, dass das Testobjekt durch die generierten Inputs in einer Dauerschleife läuft, findet keine Überprüfung der Ausführungszeiten statt, sodass dieser Fall nicht registriert wird. Des Weiteren wird abgesehen von der Konsolenausgabe kein Report generiert, der Informationen über die Inputs enthält, die zum Crash geführt haben. Eine weitere Form von Feedback wären Informationen darüber, welche Inputs überhaupt valide von dem Testobjekt entgegengenommen werden und welche Codezweige damit im Testobjekt erreicht werden können. Auch eine Möglichkeit für die Übergabe valider Inputs, sodass ausgehend von diesen mutiert werden kann, besteht in dieser Form nicht.

Es lässt sich leicht zeigen, wie ineffektiv Fuzzing ohne den Einsatz von intelligenten Techniken sein kann. Betrachtet man beispielsweise die in Listing 3.2 dargestellte Funktion, welche als Input einen String erwartet, der entsprechend einer URL mit `https://` beginnt.

```
void functionToTest(std::string url)
{
    if(url.substr(0,8) != "https://") {
        abort();
    }
    // ...
}
```

Listing 3.2: Funktion die eine URL mit "https://" erwartet

Wird der Fuzzer dabei nicht angepasst, sodass die Bytes weiterhin mit 256 verschiedenen Werten gefüllt werden können, dann ergibt sich die Zahl der Testfälle, die theoretisch zum Finden eines erfolgreichen Inputs gebildet werden müssen nach:

$$(2^8)^8 = 2^{64} \approx 1{,}845 \cdot 10^{19} \tag{3.1}$$

Die Wahrscheinlichkeit für die Generierung eines validen Inputs beträgt also:

$$P = \frac{1}{2^{64}} \approx 5{,}42 \cdot 10^{-20} \tag{3.2}$$

In eigenen Fuzz-Tests kleinerer Programme, hat der Fuzzer häufig um die 100.000 Ausführungen pro Sekunde durchgeführt. Danach ergäbe sich theoretisch zum Finden eines einzigen validen Inputs die Ausführungszeit nach:

$$\frac{1{,}845 \cdot 10^{19} exec}{100000 \frac{exec}{s}} \approx 1{,}845 \cdot 10^{14} s \tag{3.3}$$

$$\frac{1{,}845 \cdot 10^{14} s}{60 \cdot 60 \cdot 24 \cdot 365} \approx 5850456{,}6\ Jahre \tag{3.4}$$

Es zeigt sich, dass diese Form von Fuzz-Test nicht fähig ist, die entsprechende Funktion effektiv zu testen. Wenn man davon ausgeht, dass innerhalb der Funktion folgende Codezweige nach dem `https://` weitere Überprüfungen mit der URL durchführen, dann wird das Vorgehen noch ineffektiver und die Erreichung aller Codezweige ist unmöglich. Darüber hinaus bekommt der einfache Fuzzer kein Feedback über valide Inputs und merkt sich diese nicht. Das bedeutet, selbst wenn mit Glück ein valider Input gefunden wurde, können alle anschließenden Inputs wieder invalide sein. Dies verdeutlicht wie wichtig die Anwendung intelligenter Fuzz-Techniken ist und, dass diese auch auf das zu untersuchende Testobjekt angepasst werden sollten.
Ein einfaches Beispiel für eine testobjektspezifische Anpassung des einfachen Fuzzers, wäre eine Anpassung des Input-Generators. Man könnte eine Einschränkung des Wertebereichs der generierten Bytes von 256 Zeichen auf Werte von 0 bis 122 vornehmen. Diese Anpassung ist in Listing 3.3 dargestellt.

```
for(int idx2 = 0; idx2 < size; idx2++) {
    data[idx2] = rand() % 123; // Insert value 0-122 into bytes
}
```

Listing 3.3: Einschränkung des Wertebereichs des Mutators

Auf diese Weise wäre die Generierung einer URL mit `https://` weiterhin möglich aber die Zeit zum Generieren eines erfolgreichen Inputs würden sich entsprechend der folgenden Gleichung bereits signifikant verkürzen.

$$\frac{\frac{123^8 exec}{100000 exec/s}}{60 \cdot 60 \cdot 24 \cdot 365} \approx 16612{,}5\ Jahre \tag{3.5}$$

Man erkennt jedoch, dass diese Anpassung des Input-Generators noch nicht ausreicht, um den Fuzz-Test effektiv genug zu gestalten. Daher sollte immer eine Kombination von intelligenten Fuzz-Techniken und einer testobjektspezifischen Anpassung vorgenommen werden. Auch die Ausführungsgeschwindigkeit der Testfälle bzw. die Anzahl der Testfälle pro Sekunde ist ein wichtiger Faktor für die Effektivität des Fuzz-Tests. Dies ist der Grund, wieso sich ein großer Teil der Forschung und Entwicklung von Fuzzern mit der Verbesserung der Ausführungsgeschwindigkeit und der effektiven Parallelisierung beschäftigt. Im weiteren Verlauf dieses Kapitels werden diese intelligenten Techniken behandelt, die von modernen Fuzzern in verschiedenen Formen bereitgestellt werden.

3.4 Coverage-Guided Fuzzing

Um die Validität der generierten Inputs eines Fuzz-Tests messen zu können, benötigt es eine Art von Feedback, das dem Fuzzer als Metrik für die Bewertung der generierten Inputs dienen kann. Hierfür bietet sich vor allem die Codeabdeckung an, mit der gemessen werden kann, welche Zweige eines Programms während der Ausführung tatsächlich ausgeführt werden [3]. Auf diese Weise ist es dem Fuzzer möglich, Inputs die zur Ausführung von neuen Codezweigen führen, zu einem Corpus von interessanten Inputs hinzuzufügen. Die Generierung neuer Testfälle findet im Anschluss durch Mutation der im Corpus gesammelten interessanten Inputs statt. Wenn eine dieser Mutationen die Ausführung eines neuen, zuvor noch unentdeckten Codezweigs auslöst, dann wird auch dieser Input zum Corpus hinzugefügt. Basierend auf der Anzahl und Art der durchlaufenen Codezweige wird es darüber hinaus möglich, eine Bewertung der im Corpus gesammelten Inputs vorzunehmen. [24]
Durch dieses Vorgehen wird die Generierung von Testfällen durch den Fuzzer auf das Finden neuer Codezweige optimiert. Würde man Coverage-Guided Fuzzing auf die zuvor betrachtete Testfunktion aus Listing 3.2 anwenden, ohne hierbei initiale valide Inputs bereitzustellen, so würde mit der erstmaligen Generierung eines validen Inputs mit `https://` ein neuer Codezweig gefunden werden, sodass der entsprechende Input zum Corpus hinzugefügt wird. Anschließend würden die Mutationen von diesem validen Input aus fortgesetzt werden, sodass sich die Anzahl der generierten Testfälle mit validen Inputs signifikant erhöht. Hierbei ist es schwer die Verbesserung in der Generierung von validen Inputs zu quantifizieren, da dies maßgeblich vom Testobjekt und dem verwendeten Mutator abhängt. Grundsätzlich macht die Anwendung von Coverage-Guided Fuzzing in den meisten Anwendungsfällen Sinn, da der Fuzzer bei neu entdeckten Codezweigen im Anschluss mit hoher Wahrscheinlichkeit erneut an die gleiche Stelle im Code des Testobjektes vordringt. Da die Mutation von den gespeicherten Inputs aus fortgesetzt wird, wird so sichergestellt, dass der Fuzzer immer tiefer im Code des Testobjektes vordringen kann und weitere Codezweige erreicht.
Für Anwendungsfälle wo Coverage-Guided Fuzzing keinen Sinn ergibt, sind vor allem Fuzz-Tests zu nennen, wo der Fuzzer gezielt und dauerhaft auf eine bestimmte Stelle im Code gelenkt werden soll. In diesem Fall möchte man die Mutation ggf. permanent vom gleichen Input aus durchführen. Wenn man den Quellcode zur Verfügung hat, dann kann man die entsprechende Stelle im Quellcode jedoch heraustrennen und den Fuzz-Test trotzdem mit einem Fuzzer durchführen, der mit Coverage-Guided Fuzzing arbeitet.

3.4.1 Generierung von Coverage-Informationen

Für die technische Umsetzung des Coverage-Feedbacks gibt es bei verschiedenen Fuzzern unterschiedliche Ansätze, welche ihre eigenen Vor- und Nachteile haben. Die in dieser Arbeit genauer betrachteten Fuzzer libFuzzer und AFL/AFL++ verwenden Instrumentierung zum Sammeln der Infomationen über ausgeführte Codezweige im Testobjekt [15]. Die Form der Coverage die von beiden Fuzzern generiert wird bezeichnet man als `Edge-Coverage`, die zusätzlich um einen Zähler ergänzt wird. [24] [25]
Hierzu muss man wissen, dass sich der Ablauf eines Programms in Form eines Graphen beschreiben lässt, dem sogenannten Control Flow Graph (CFG). Dieser beschreibt alle möglichen Pfade, die das Programm während seiner Ausführung durchlaufen kann. Er setzt sich aus `Nodes` und `Edges` zusammen. Eine `Node` innerhalb des CFG repräsentiert einen sog. `Basic Block`, also eine lineare Sequenz von Programm-Instruktionen mit exakt einem Einstiegspunkt und einem Ausstiegspunkt. Innerhalb des `Basic Blocks` finden keine Verzweigungen und Sprünge statt. Am Einstiegs- oder Ausstiegspunkt sind wiederum Übergänge von und zu mehreren anderen `Basic Blocks` möglich. Eine `Edge` beschreibt die einzigartige Beziehung zwischen zwei direkt miteinander verbundenen `Nodes` innerhalb des CFG. Typischerweise versteht man darunter die Sprünge zwischen zwei direkt miteinander verbundenen `Nodes`, wobei es auch sein kann, dass eine Node einen Übergang zu sich selbst hat. Dieser Fall ist ebenfalls eine `Edge`. [26] Zur Veranschaulichung ist in Abbildung 3.4 ein simpler CFG mit vier Nodes dargestellt.

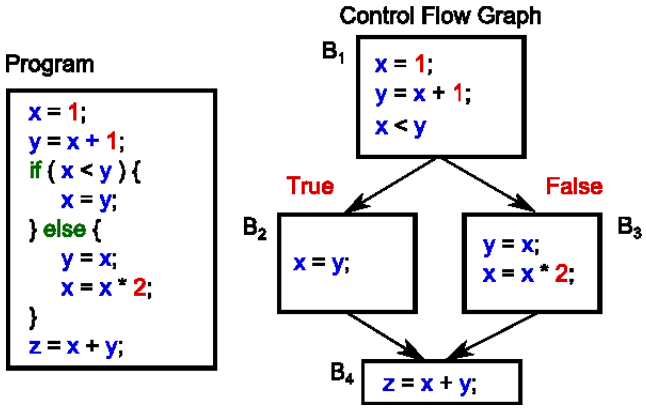

Abbildung 3.4: CFG eines einfachen Programms [27]

Um das Prinzip der Generierung der `Edge-Coverage` zu erklären, wird sich im Folgenden auf AFL/AFL++ bezogen, da es zu diesen Fuzzern eine bessere Dokumentation gibt. AFL/AFL++ führt eine Instrumentierung auf Assembly-Level durch, also nachdem der C/C++ Quellcode vom Compiler in Assembly übersetzt wurde. Hierbei instrumentiert AFL/AFL++ jeden einzelnen `Basic Block`. Ein Beispiel für eine derartige Instrumentierung ist in Listing 3.4 dargestellt. Das Beispiel soll nur das Prinzip veranschaulichen, die exakten Details, die aus dem eingefügten Aufruf folgen, würden den Rahmen dieser Arbeit überschreiten. [28]

```
1 lea     rsp, [rsp-98h] ; allocate some stack space
2 mov     qword ptr [rsp+98h+input+0F80h], rdx
3 mov     qword ptr [rsp+98h+input+0F88h], rcx
4 mov     qword ptr [rsp+98h+input+0F90h], rax ; save registers
5 mov     rcx, 0FC50h ; rcx is the block identifier, generated randomly
6 call    __afl_maybe_log
```

```
7  mov     rax, qword ptr [rsp+98h+input+0F90h]
8  mov     rcx, qword ptr [rsp+98h+input+0F88h]
9  mov     rdx, qword ptr [rsp+98h+input+0F80h]
10 lea     rsp, [rsp+98h] ; recover stack and variables
11 }
```

Listing 3.4: Beispiel für Insertierung durch AFL auf Assembly-Level [28]

Laut dem technischen Whitepaper zu AFL [29] entspricht der in jeden Zweigpunkt instrumentierte Code folgender Pseudocode-Repräsentation:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Das `shared_mem` Array repräsentiert eine 64 kB große Region von geteiltem Speicher, die instrumentiert wird. Jedes Byte, das in diesem Speicherbereich gesetzt wird, repräsentiert einen Zähler für einen spezifischen Übergang zwischen zwei Blöcken, also eine `Edge` im CFG. Diese kann man sich entsprechend der `XOR`-Operation als Tupel (src_branch, dest_branch) vorstellen. Die Größe dieses Speicherbereichs wurde so gewählt, dass Kollisionen für die Anzahl der Zweige innerhalb der meisten Fuzz-Targets selten sind aber gleichzeitig klein genug, dass eine Analyse innerhalb von Mikrosekunden durchgeführt werden kann. Die `Bitshift`-Operation in der letzten Zeile des Pseudocodes dient dazu, die Richtung eines Übergangs zwischen zwei Blöcken zu bewahren. Ohne diese Operation wären die `Edges` (A XOR B) und (B XOR A) nicht zu unterscheiden. Ähnliches gilt für die Unterscheidung der Übergänge (A XOR A) und (B XOR B), also Übergänge, die zurück zum Anfang des entsprechenden Blocks führen. Diese Problematik in der Unterscheidung ohne `Bitshift` besteht verdeutlicht folgendes Beispiel.

```
1  A = 1010 ; B = 1101
2
3  A ^ B = 1010 = B ^ A = 1101
4          1101           1010
5          ----           ----
6          0111           0111
7
8  A ^ A = 1010 = B ^ B = 1101
9          1010           1101
10         ----           ----
11         0000           0000
```

Listing 3.5: Probleme bei der Unterscheidung von Edges ohne Bitshift

Insgesamt bewahrt der Fuzzer so eine globale Liste von Übergängen aus vorherigen Ausführungen. Wenn ein mutierter Input einen Ausführungspfad erzeugt, der neue Übergänge enthält, dann wird dieser Input zum Corpus hinzugefügt. Dabei ist ein Vergleich von Daten aus der Liste mit einem individuellen Pfad einer Ausführung durch die Bitoperationen in wenigen Instruktionen möglich. Das bedeutet, die Analyse und das Updaten der Liste ist mit wenig Rechenaufwand möglich. Dies stellt einen erheblichen Vorteil dar, da der Fuzzer möglichst viele Testfälle pro Zeiteinheit generieren soll. Gleichzeitig hat man den Vorteil der `Edge-Coverage`, die mehr Informationen über den exakten Ausführungspfad im Programm bereitstellt, als reine `Block-Coverage`. [29]

3.4.2 Hybridmetrik aus Edge-Coverage und Hitcounter

Nachdem nun bekannt ist, wie ein Fuzzer ein Feedback durch `Edge-Coverage` generieren kann, gilt es noch zu klären, wieso libFuzzer und AFL/AFL++ eine Hybridmetrik verwenden. Beide Fuzzer kombinieren die `Edge-Coverage` mit einem Zähler, welcher festhält, wie oft eine entsprechende `Edge` des CFG durchlaufen wird [24] [25]. Angenommen es werden zuerst der folgende Programmpfad durchlaufen:

```
#1: A -> B -> C -> D -> E
#2: A -> B -> C -> A -> E
```

Daraus folgt, dass der Fuzzer die folgenden `Edges` neu entdeckt:
(A,B) ; (B,C) ; (C,D) ; (C,A) ; (D,E) ; (A,E)
Angenommen im Anschluss wird nun folgender Programmpfad durchlaufen:

```
#3: A -> B -> C -> A -> B -> C -> A -> B -> C -> D -> E
```

Bei Feedback aus reiner `Edge-Coverage` würde der Fuzzer den Pfad #3 nicht als Neuentdeckung klassifizieren, da alle durchlaufenen `Edges` bereits bekannt sind. Dies ist der Fall, obwohl dieser einen komplett anderen Ausführungspfad verfolgt als die beiden vorherigen Ausführungspfade. Damit der Fuzzer auch den dritten Pfad als einzigartigen interessanten Ausführungspfad klassifizieren kann, ergänzt man die `Edge-Coverage` um einen Zähler. Sobald registriert wird, wie oft eine entsprechende `Edge` innerhalb einer Ausführung durchlaufen wird, ist es auch hier möglich, eine Unterscheidung zwischen den verschiedenen Ausführungspfaden zu treffen. [29]

3.4.3 Corpus Management

Beim Coverage-Guided Fuzzing wird der Corpus zunehmend mit interessanten Inputs gefüllt. Im Laufe der Ausführung des Fuzz-Tests kann der Corpus dabei sehr groß werden und ggf. Duplikate von Inputs enthalten, welche letztendlich bei der Ausführung zur gleichen Coverage führen. Dies ist besonders der Fall bei langlaufenden Fuzz-Tests mit großen Testobjekten oder wenn die gesammelten Corpora mehrerer Testläufe miteinander vereint werden. Gleiches gilt für eine große Menge von bereitgestellten initialen Inputs, von denen vor Beginn des Tests nicht bekannt ist, welche Coverage mit ihnen erreicht wird. Die Folge daraus ist, dass sich die Effektivität des Fuzz-Tests verschlechtert, da Testfälle durch Mutationen mit allen Inputs im Corpus generiert werden. Hat man hierbei viele Duplikate innerhalb des Corpus, dann kommt es statistisch zu mehr Mutationen an Inputs gleicher Pfade. Die Mutationen an Inputs anderer Pfade treten dabei statistisch in den Hintergrund, wodurch es ggf. länger dauert neue Pfade zu entdecken. Es gilt die Devise, je kleiner der Corpus mit dem die gleichen Pfade im Testobjekt durchlaufen werden, desto effektiver das Fuzzing. [30]
Um dies zu erreichen, bieten moderne Fuzzer Funktionen zur Minimierung des Corpus bei gleichzeitigem Erhalt der Coverage an. Im Fall von libFuzzer kann dies durch die `-merge=1` Option beim Ausführen des Fuzz-Tests durchgeführt werden. Auf diese Weise lassen sich auch Corpora miteinander vereinen, indem innerhalb des `NEW_CORPUS_DIR` bereits ein Corpus vorhanden ist. [24]

```
mkdir NEW_CORPUS_DIR  # Store minimized corpus here.
./my_fuzzer CORPUS_DIR # Ausführen des Fuzzers mit gegeben Corpus
./my_fuzzer -merge=1 NEW_CORPUS_DIR FULL_CORPUS_DIR # Merge zweier Corpora
```

Bei AFL/AFL++ kann das Gleiche über den Aufruf von `afl-cmin` erreicht werden [30].

3.4.4 Manuelle Verifikation von Fuzz-Tests über Coverage-Report

Da Coverage-Guided Fuzzing ein Grey-Box-Ansatz ist, stellt sich hierbei die Frage, ob überhaupt alle Codezweige innerhalb des Testobjektes erreicht wurden. Wenn man über den Quellcode des Testobjektes verfügt, dann kann man hier eine manuelle Verifikation des Fuzz-Tests über einen Coverage-Report durchführen. Auf diese Weise erhält man eine Auswertung darüber, welcher Codezweig wie oft durchlaufen wurde. Hierbei ist nicht nur interessant, ob alle Codezweige erreicht wurden, sondern auch, ob die interessanten Stellen im Verhältnis zur Gesamtzahl der Testfälle oft genug durchlaufen wurden. Basierend darauf kann man eine Evaluation der initialen und gesammelten Einträge des Corpus durchführen und diesen ggf. anpassen. Dadurch ist es möglich den Fuzzer in einem weiteren Fuzz-Test gezielt in andere Stellen des Testobjektes zu lenken. Mit Clang kann man einen Coverage-Report generieren, indem man den Code neben entsprechenden Flags des Fuzz-Tests mit folgenden Compiler-Flags kompiliert:

```
-fprofile-instr-generate -fcoverage-mapping
```

Wenn das Kompilat (der Fuzz-Test) anschließend ausgeführt wird, dann wird es nach dem Beenden ein sogenanntes `raw profile` auf den Pfad der Umgebungsvariable `LLVM_PROFILE_FILE` erzeugen. Daher ist es ratsam, den Fuzz-Test wie folgt auszuführen:

```
LLVM_PROFILE_FILE="fuzz_test.profraw" ./fuzz_test
```

Bevor man sich den Coverage-Report anschauen kann, muss dieser indexiert werden und im nächsten Schritt kann man sich diesen anzeigen lassen.

```
(a): Index the raw profile.
llvm-profdata merge -sparse fuzz_test.profraw -o fuzz_test.profdata
(b): Create a line-oriented coverage report.
llvm-cov show ./fuzz_test -instr-profile=fuzz_test.profdata
```

Ein auf diese Weise erzeugter Coverage-Report eines mit libFuzzer durchgeführten Fuzz-Tests ist in Abbildung 3.5 dargestellt. Man kann zwischen Zeile 36 und 39 den Einstiegspunkt des Fuzzers erkennen und, dass dieser 10.000 Testfälle erzeugt hat. Darüber ist eine Funktion zur Testvorbereitung, die das Testobjekt nur ausführt, wenn die von libFuzzer generierten Bytes Vielfache von vier sind. Der Grund hierfür ist, dass die Größe des Integer-Inputarrays abhängig von der Menge generierter Bytes des Fuzzers sein muss. Man erkennt, dass das Testobjekt nur in 3840 Fällen tatsächlich ausgeführt wurde. Auf Basis derartiger Daten wird eine Evaluation des Fuzz-Tests möglich. [31]

```
27|       |void fuzz_memoryLeakFunction(const uint8_t *data, size_t size)
28|  10.0k|{
29|  10.0k|    if ( (size % 4) == 0) {
30|  3.84k|        int aInput[size/4];
31|  3.84k|        std::memcpy(&aInput, data, size);
32|  3.84k|        memoryLeakFunction(aInput, size/4);
33|  3.84k|    }
34|  10.0k|}
35|       |
36|  10.0k|extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
37|  10.0k|    fuzz_memoryLeakFunction(data, size);
38|  10.0k|    return 0;
39|  10.0k|}
```

Abbildung 3.5: Ausschnitt aus dem Coverage-Report eines Fuzz-Tests mit libFuzzer

3.5 Mutation-Based Fuzzing

Abgesehen von der rein zufälligen Generierung von Inputs, basiert Fuzzing in den meisten Fällen auf dem Prinzip der Mutation zur Generierung neuer Inputs bzw. Testfälle. Wie das Beispiel der URL-Funktion aus Listing 3.2 bereits verdeutlichte, ist der Großteil der zufällig generierten Inputs ungültig. Da ein solcher Fuzz-Test ineffektiv ist, wurde beispielhaft der Wertebereich des Inputgenerators eingeschränkt. So wurde gezeigt, dass so die Chance auf gültige Inputs bereits erhöht werden kann. Was wäre aber, wenn im weiteren Verlauf der URL-Funktion ein Zeichen abgeprüft wird, das sich nicht mit den eingeschränkten Werten von 0 bis 122 darstellen lässt? Damit ein Fuzz-Test in der Lage ist die Teile des Testobjekts zu erkunden, die über die initiale Inputverarbeitung hinausgehen, muss die Chance auf valide Inputs stark erhöht werden. Wie beim Coverage-Guided Fuzzing bereits angedeutet, ist die Idee beim Mutation-Based Fuzzing, dass man kleine Änderungen an existierenden validen Inputs durchführt. Das Ziel ist dabei, dass der Input bei kleinen Änderungen gültig bleibt aber dennoch neues Verhalten innerhalb des Testobjektes entdeckt wird. [3]
Grundsätzlich ist die Art der Mutation bzw. der verwendete Mutator ein zentrales Thema bei der Gestaltung von effektiven Fuzz-Tests. Daher bieten moderne Fuzzer wie libFuzzer oder AFL/AFL++ die Möglichkeit eigene Mutatoren (sog. Custom Mutators) einzusetzen. Der praktische Einsatz von Custom Mutatoren wird im weiteren Verlauf dieser Arbeit genauer demonstriert. Um die Effektivität von testobjektspezifischen Mutatoren zu verdeutlichen, soll im Folgenden zunächst weiterhin das Beispiel einer URL betrachtet werden. Angenommen man betrachtet den in Listing 3.6 dargestellten Code, wo eine URL als initialer valider Input besteht, welche durch eine beliebige Mutator-Funktion mutiert wird.

```
std::string valid_input = "http://www.google.com/search?q=fuzzing";
for(int idx = 0; idx < 4; idx++) {
    std::string mutated_input = mutator(valid_input);
    std::cout << mutated_input << std::endl;
}
// Beispielhafte Ausgaben:
//1. Xttp://www.google.com/search?q=fuzzing   // gleiche Zeichenzahl
//2. http://www.google.comR/search?q=fuzzing  // zusaetzliches Zeichen
//3. http://www.google.com/search?q=fuzz      // weniger Zeichen
//4. http://www.google.com/search?q=fuzz ing  // Leerzeichen eingefuegt
```

Listing 3.6: Beispiel: URL mit beliebigem Mutator

An dieser Stelle zeigt sich bereits, wieso ein testobjektspezifischer Custom Mutator Sinn ergibt und auf welche Dinge dabei geachtet werden sollte. Betrachtet man die erste Ausgabe, dann wurde hier die Mutation direkt am Anfang des `http://` vorgenommen. Wenn man weiß, dass dies die Voraussetzung für einen validen Input ist, dann sollte der Mutator ggf. direkt so angepasst werden, dass die Mutation innerhalb dieser Zeichen nicht stattfinden kann. Die zweite und dritte Ausgabe sollen verdeutlichen, dass man sich grundsätzlich die Frage stellen muss, ob der Mutator den Input zeichen- oder auch speichermäßig vergrößern oder verkleinern können soll. Im Fall von Zeichenketten mag dies sinnvoll sein aber bei strukturierten Daten mit definierten Datentypen möchte man diese Art der Mutation ggf. direkt ausschließen. Das vierte Ausgabebeispiel soll verdeutlichen, dass es testobjektspezifische Beschränkungen geben kann, die direkt in den Mutator mit eingearbeitet werden sollten. In diesem Beispiel wird ein Leerzeichen eingefügt, welches jedoch häufig in einer URL mit %20 codiert eingefügt werden muss [32]. Hier wäre es eventuell angebracht, eine Mutation mit einem Leerzeichen zu verbieten oder dieses entsprechend zu codieren. Ähnliches gilt für die Spezifika unterschiedlicher Zeichen-Codierungen. In ISO8859-1 ist jedes Zeichen mit einem Byte codiert, während in UTF-8

einzelne Zeichen mit bis zu vier Byte codiert sein können. Dies zu beachten kann die Effektivität der Mutationen signifikant erhöhen.
Darüber hinaus stellt sich die Frage wie stark mutiert werden soll. In den bisherigen Beispielen hat der Mutator eine einzige Mutation ausgehend vom validen Input durchgeführt. Auf diese Weise kann der Input nur in einem beschränkten Maß entstellt werden. Zum Erreichen weiterer Programmzweige kann es ratsam sein, mehrere gleichartige Mutationen auf einmal auszuführen. Auch Mutationen ausgehend von bereits mutierten Inputs können an dieser Stelle wichtig sein. Des Weiteren spielt die Art der Mutationen eine wichtige Rolle. Hierbei kann man simple Mutationen wie das Flippen, Abschneiden oder Einfügen von Bits durchführen. Man kann in einem Custom Mutator jedoch auch spezifischere Mutationen implementieren. Unter Umständen ist es sinnvoll mehrere Arten von Mutationen miteinander zu kombinieren. Grade im Hinblick auf Fuzz-Tests für Funktionen, die verschiedene Input-Variablen aus Standarddatentypen erwarten, ist dies wichtig. Einige der Inputs dürfen eventuell nur bestimmte Werte annehmen, nur in bestimmter Weise oder gar nicht mutiert werden. Andere sind Zeichenketten und sollen daher eventuell eher stark mutiert werden.
Alles in allem ist die Wahl und testobjektspezifische Anpassung des Mutators ein sehr wichtiger Faktor, der die Effektivität des Fuzz-Tests maßgeblich beeinflusst. Da es schwer ist diesbezüglich eine allgemeingültige Aussage zu treffen, sollen die dargestellten Beispiele und Faktoren vor allem verdeutlichen, dass der Tester alle Möglichkeiten hat, den Mutator nach den eigenen Bedürfnissen anzupassen. Auch die Durchführung eines Fuzz-Tests, der nacheinander unterschiedliche Custom Mutatoren verwendet, ist an dieser Stelle denkbar.

3.6 Grammar-Based Fuzzing

Grammar-Based Fuzzing ist ein Ansatz zum Testen von Programmen, die hochstrukturierte Inputs erwarten. Gute Beispiele für derartige Programme sind Compiler einer Programmiersprache oder Parser für Beschreibungssprachen strukturierter Daten wie XML. Problematisch beim Fuzzen derartiger Programme ist es, dass hierbei in der Regel viel manueller Aufwand erforderlich ist, um die Grammatik der komplexen Inputs für den Fuzzer umsetzbar zu beschreiben. Dies macht diese Technik je nach Anwendungsgebiet schwierig zu implementieren, sowie zeit- und kostenintensiv. Daraus resultiert, dass reines Grammar-Based Fuzzing im Vergleich zu anderen Fuzz-Techniken eher selten eingesetzt wird. Aktuelle Forschungen auf dem Gebiet beschäftigen sich mit dem Einsatz von Machine Learning, um die Nachteile dieser Technik zu überwinden. [33]
Da Grammar-Based Fuzzing ein komplexes Thema ist und es sich aufgrund dessen nicht für den weiteren Einsatz in dieser Arbeit eignet, soll im Folgenden nur ein Einblick gegeben werden, was darunter zu verstehen ist. Grundsätzlich steht dahinter die Idee, dass dem Fuzzer statt valider Inputs zum Mutieren, eine Spezifikation (Grammatik) übergeben wird. Diese Spezifikation beschreibt, wie der Fuzzer sehr komplexe valide Inputs generieren kann. [3] Bleibt man hier bei komplexem C++ Code als Beispiel für valide Inputs und einem Compiler als Testobjekt, dann zeigt sich schnell, wie schwer es ist, eine geeignete Grammatik für den Fuzzer zu definieren. Die Generierung von syntaktisch korrektem Code ist hierbei einfacher möglich, als diesem irgendeine sinnvolle Semantik zu verleihen. Anschließend stellt sich die Frage, wie gut man die Spezifikation semantisch gestalten muss, um dadurch effektivere Testfälle zu erzeugen, als dies der Fall wäre, wenn man den Compiler mit zufälligem Code aus dem Internet testen würde. Um das grundlegende Prinzip zu veranschaulichen, soll ein simplifiziertes Beispiel einer C++ Grammatik für die Zuweisung von Char-Variablen in Listing 3.7 dargestellt werden.

```
struct {
    std::vector<std::string> keyword = {"char", "const char"};
    std::vector<std::string> value = {"0x00", "0x01", ..., "0xFE", "0xFF"};
    std::string statementEnd = ";\n" ;
}grammar;
```

Listing 3.7: Simplifizierte C++ Grammatik für die Zuweisung von Char Variablen

Zum Erzeugen von C++ Code müsste der Fuzzer nun einen riesigen String nach der gegebenen Grammatik erzeugen und diesen in eine cpp-Datei schreiben und damit den Compiler ausführen. Wenn die Semantik an dieser Stelle vernachlässigt werden würde, dann könnte dies für das simplifizierte Beispiel entsprechend Listing 3.8 nach dem Zufallsprinzip erfolgen.

```
std::string assignChar() {
    std::srand(std::time(nullptr)); // #include <ctime> necessary
    std::string statementForCharAssignment;
    std::string variablenName = "var" + std::to_string(rand() % 10000);
    // <keyword> <variablenname> =  <value>;
    statementForCharAssignment =   grammar.keyword[rand() % 2 ] + " "
                                 + variablenName + " = "
                                 + grammar.value[rand() % 256]
                                 + grammar.statementEnd ;
    return statementForCharAssignment;
}
```

Listing 3.8: Zufallsbasierte Erzeugung von Code auf Basis von Grammatik

Dieser simple Grammatik-Baustein zur Erzeugung einer Char-Variablen kann anschließend in größeren Grammatik-Bausteinen verwendet werden, um auf diese Weise komplexere Strukturen zu erzeugen. Es wird deutlich, wie viel Aufwand hierfür erforderlich ist. Der zulässige Wertebereich eines Chars lässt sich beispielsweise noch relativ leicht abbilden. Bei komplexeren Codestrukturen wird man auf immer kompliziertere Probleme stoßen. Eines dieser Probleme ist die Namensgebung von Variablen und Funktionen. Hier wurden Zufallszahlen verwendet, tatsächlich müsste man aber eine eindeutige Namensgebung bewahren und dabei auch einen Überblick erhalten, sodass man benannte Bausteine an anderer Stelle wiederverwenden kann. Dies führt dazu, dass reines Grammar-Based Fuzzing eine sehr komplizierte Technik ist, die hinsichtlich des Aufwands nur an sehr speziellen Stellen einen Nutzen bringen kann. Gemäß der aktuellen Entwicklungen im Bereich der künstlichen Intelligenz ist es jedoch gut möglich, dass sich der Arbeitsaufwand bezüglich der Grammatik-Spezifikation drastisch verringern lässt. Dies könnte dazu führen, dass diese Technik in der Zukunft leichter anzuwenden ist.
Reines Grammar-Based Fuzzing unterscheidet sich von anderen Fuzz-Techniken vor allem dadurch, dass es auf einem generativen Ansatz zur Erstellung von Testfällen beruht, während andere Techniken hierfür Mutationen nutzen. [34] Hinsichtlich der Implementierung von Custom Mutatoren lässt sich das Prinzip von Grammar-Based Fuzzing dennoch mit anderen Fuzz-Techniken kombinieren. So kann man innerhalb von testobjektspezifischen Mutatoren kleinere Grammatik-Bausteine verwenden, um auf deren Basis einzelne Teile der generierten Inputs in festgelegtem Maße zu mutieren. Betrachtet man hierfür erneut die URL als Beispiel, so könnte man nach gleichem Prinzip eine Grammatik für die URL festlegen. Dadurch könnte man sicherstellen, dass nach dem zweiten Punkt in jedem Fall eine der gängigen Domain-Endungen folgt. Auf diese Weise ist es möglich Mutation-Based Fuzzing unter Anwendung generativer Elemente durchzuführen, was zur nächsten Fuzz-Technik in diesem Kapitel führt, dem Structure-Aware Fuzzing.

3.7 Structure-Aware Fuzzing

Der Begriff des Structure-Aware Fuzzings hat sich innerhalb der Literatur noch nicht eindeutig etabliert. An einigen Stellen wird Grammar-Based Fuzzing als generative Technik mit diesem Begriff bezeichnet, während an anderen Stellen mutationsbasierte Techniken mit einem tieferen Verständnis über die Struktur der Inputs damit gemeint sind. Die im späteren Verlauf dieser Arbeit vorgestellte Fuzz-Technik wurde von ihren Entwicklern unter dem Begriff Structure-Aware Fuzzing vorgestellt. Da es sich dabei um eine mutationsbasierte Technik handelt, soll der Begriff in dieser Arbeit für mutationsbasierte Techniken verwendet werden, die ihre Mutationen auf Basis eines tieferen Verständnisses der Inputstruktur durchführen.
Grundsätzlich ist das fehlende Wissen über die Struktur der Inputs eine der größten Limitierungen von Coverage-Guided Fuzzing bzw. Grey-Box Fuzzing im Allgemeinen. Viele Testobjekte verarbeiten hochstrukturierte Daten. Das können auf höherer Ebene zum Beispiel Bilder, Audio, Datenbanken oder Dokumente sein. In der Regel setzt sich dies jedoch auf niedrigeren Ebenen innerhalb der Software fort, selbst wenn dabei nur Teile der komplexeren Inputs höherer Ebenen weiterverarbeitet werden. Einfache Mutationen auf Bit-Ebene sind hier nicht in der Lage effektiv neue Pfade innerhalb des Testobjektes zu erkunden. Viel wahrscheinlicher ist es, dass sie zu invaliden Inputs führen, sodass der Fuzzer gar nicht an die interessanten Stellen der Datenverarbeitung innerhalb des Testobjektes gelangt. [35] Löst man sich in dieser Betrachtung von der Verarbeitung einzelner Dateien mit festgelegten Dateiformaten, dann hat man in der Software viele Stellen, wo einzelne Funktionen eine große Zahl von Inputs mit festgelegten Datentypen entgegennehmen. Einzelne dieser Inputs können variable Größen haben oder dürfen sich nur in bestimmten Wertebereichen bewegen. Es können Objekte von definierten Strukturen oder Klassen sein, die selbst verschachtelt und komplex sein können. Verstärkt wird dies durch die Tatsache, dass im Zuge der Objektorientierung in der Regel Zusammenhänge aus der echten Welt abgebildet werden. Dies führt dazu, dass einzelne Inputs oder Member der Objekte von Inputs häufig nur festgelegt Werte annehmen dürfen und damit nicht beliebig oder gar nicht mutiert werden sollten. Um dieses Problem zu veranschaulichen ist in Listing 3.9 beispielhaft der Header einer beliebigen Funktion zum Verarbeiten eines Tiefenbildes dargestellt.

```
void processDepthImage(
              const DetectionInfo & info,
              const Header & header,
              const char *depthImageData,
              const size_t sizeOfData,
              float depthConstant,
              bool & success) {
    //...check inputs and do something
}
```

Listing 3.9: Funktion mit hochstrukturierten Inputs

Eine deartige Funktion mit Coverage-Guided Fuzzing ohne Informationen zur Struktur der Inputs zu fuzzen ergibt wenig Sinn. Info und Header können Strukturen oder Klassen sein, die selber eigene Member aus Standarddatentypen enthalten. Je nachdem wie diese Objekte verarbeitet werden, kann es wichtig sein wie ihr Inhalt im Speicher angeordnet ist. Außerdem wäre es denkbar, dass Werte innerhalb des Headers oder von `DetectionInfo` nur feste Werte annehmen dürfen, um damit Informationen mit realem Bezug zu codieren. Das Tiefenbild wird in diesem Beispiel als Rohdaten übergeben, dennoch können diese Daten innerhalb der `processDepthImage` Funktion als hochstrukturiert verarbeitet werden. Selbst wenn an dieser Stelle ein initialer valider Input übergeben werden würde, dann können bitweise Mutationen sehr schnell an den falschen Stellen für invalide Inputs sor-

gen und damit den Fuzz-Test ineffektiv machen.
Die Idee beim Structure-Aware Fuzzing ist es, dass der Mutator des Fuzzers mit allen nötigen Informationen über die Struktur der Inputs versorgt wird. Dadurch soll sichergestellt werden, dass alle generierten Inputs zumindest strukturell valide sind. Auf diese Weise wird die Effektivität des Fuzz-Tests erhöht, da sich die Chance auf die Generierung valider Inputs stark vergrößert. Der Ansatz der im weiteren Verlauf dieser Arbeit verfolgt wird, stattet den Fuzzer mit allen Datentypen der Inputs aus. Dadurch kann über den Mutator explizit festgelegt werden, dass ein String wie ein String zu mutieren ist und dieser eine variable Größe haben kann. Ein Char hingegen muss weiterhin 1 Byte groß bleiben und darf nur zwischen 0x00 und 0xFF mutiert werden. Einzelne Datentypen können hierbei leicht von der Mutation ausgeschlossen werden und komplexe verschachtelte Datentypen können bis auf die Standarddatentypen ihrer Member heruntergebrochen und so nachgebildet werden. [36]

3.8 Bewertung von Fuzz-Techniken für Unit-Tests

Für die Auswahl einer geeigneten Fuzz-Technik innerhalb von Unit-Tests gilt zunächst, dass ein kompletter Black-Box-Ansatz nicht zielführend ist, da dieser unter dem Fehlen von Informationen leidet und damit grundsätzlich ineffektiv ist. Im Gegensatz dazu ist ein Grey-Box-Ansatz wie Coverage-Guided Fuzzing immer zu bevorzugen, da der Aufwand bei der Implementierung in beiden Ansätzen ähnlich gering ist. Grundsätzlich ist Coverage-Guided Fuzzing mehr oder weniger der Standard, da die Auswertung der Codeabdeckung ausschließlich Vorteile bringt und nur in den seltensten Fällen unangebracht ist. White-Box-Ansätze kommen hingegen mit einem großen Aufwand bezüglich der Codeanalyse und sind schwieriger zu implementieren [35]. Darüber hinaus ist nach dem Anfangs beschriebenen Ablauf der Unit-Tests der Quellcode und entsprechendes Wissen vorhanden, sodass sich ein Grey-Box-Fuzzer so steuern lässt, dass mit diesem ebenfalls alle Codezweige zu erreichen sind.
Da der Einsatz von Fuzz-Tests zwingend zusätzliche Ressourcen innerhalb des Testprozesses erfordert, ist es ein guter Grundsatz die Fuzz-Tests zunehmend von unspezifisch zu spezifisch bzw. von wenig Implementierungsaufwand zu viel Implementierungsaufwand durchzuführen. Damit ist gemeint, dass zunächst mit standardmäßigem Coverage-Guided Fuzzing begonnen werden kann. Werden hierbei bereits Fehler entdeckt, dann kann man diese zunächst beheben, ohne weiteren Aufwand zu betreiben. Egal ob Fehler gefunden wurden oder nicht, es muss in jedem Fall die Effektivität des Fuzz-Tests untersucht werden. Hier sollten sich die folgenden Fragen gestellt werden:

- Wurden alle Codezweige bzw. interessante Stellen im Testobjekt erreicht?
- Wie gut war mein initialer Corpus?
- Wurde der Fuzz-Test lang genug durchgeführt?
- Benötigt es einen besseren und testobjekt-spezifischeren Mutator?
- Macht eine spezifischere Suche an dieser Stelle Sinn?

Wenn nicht alle Codezweige erreicht wurden, dann wäre eine Verbesserung des Corpus zunächst die einfachste Lösung. Kommt man hingegen zu dem Ergebnis, dass der Fuzz-Test nicht in der Lage war das Testobjekt in gewünschtem Maße zu testen oder man hat Fehler gefunden und vermutet weitere, dann kann das Coverage-Guided Fuzzing erweitert werden. Ab hier entsteht in der Regel größerer Arbeitsaufwand, da nun testobjektspezifisch ein Custom Mutator erstellt werden muss. Dieser muss

so gestaltet werden, dass der Fuzzer dadurch zu dem gewünschten Verhalten bzw. den richtigen Stellen gesteuert werden kann. Wie man diesen gestaltet ist letztendlich testobjektabhängig, wobei die Anpassung auf die Struktur der Inputs des Testobjektes generell ein guter Ansatzpunkt ist.
Alles in allem schließen sich die einzelnen Fuzz-Techniken gegenseitig nicht aus, sondern sollten miteinander kombiniert bzw. umeinander erweitert werden. Die einzige Ausnahme ist hier das rein generative Grammar-Based Fuzzing, welches aufgrund des großen Aufwands wirklich nur in speziellen Fällen angebracht ist. Das Grundprinzip kleine Grammatik-Bausteine innerhalb eines Custom Mutators zu verwenden, sollte in jedem Fall für die Kombination mit anderen Fuzz-Techniken in Betracht gezogen werden. Wichtig ist, dass es immer eine Kosten-Nutzen-Frage ist, ob und wie der Fuzz-Test erweitert werden soll. Hierbei sind zum einen die Fähigkeiten und die Einschätzung des Testers entscheidend. Andererseits stellt sich die Frage der Leistungsfähigkeit einzelner Techniken im Verhältnis zu ihrem Aufwand. Wenn es möglich wäre, eine testobjektspezifische und damit effektivere Fuzz-Technik soweit zu automatisieren, dass sich der Implementierungsaufwand auf ein Minimum reduziert, dann würde es Sinn ergeben, direkt mit dieser Technik zu fuzzen, ohne mit reinem Coverage-Guided Fuzzing zu beginnen. Hierfür gibt es mit den von Google entwickelten `Protocol Buffern` [37] einen vielversprechenden Ansatz, um dadurch Structure-Aware Fuzzing weitestgehend zu automatisieren bzw. mit minimalem Aufwand zu implementieren. Der Frage wie und ob dies möglich ist, wird im weiteren Verlauf dieser Arbeit nachgegangen.

4 Sanitizer

In der Regel werden Fuzz-Tests mit Tools kombiniert, die Fehler durch undefiniertes oder auffälliges Verhalten aufdecken. Hierzu werden sogenannte Code Sanitizer eingesetzt, bei denen der Compiler während des Kompiliervorgangs eine Instrumentierung vornimmt. Der dabei eingefügte Code leistet die Fehlererkennung. Es gibt verschiedene Arten von Sanitizern, die dazu geeignet sind, unterschiedliche Fehlerklassen zu erkennen. Sie unterscheiden sich in ihrer Funktionsweise und den Stellen, an denen der Compiler die Instrumentierung im Code vornimmt. [38] Im Folgenden werden die in Kombination mit Fuzz-Tests häufig verwendeten Arten von Sanitizern vorgestellt.

4.1 Address-Sanitizer

Bei Address-Sanitizern (ASan) handelt es sich um ein Tool zur Erkennung von Speicherfehlern, dessen Einsatz besonders bei Programmiersprachen mit freier Speicherverwaltung wie C,C++ ratsam ist. Im Gegensatz zu anderen Tools zur Erkennung von Speicherfehlern zeichnen sich Address-Sanitizer durch ihr einfaches Funktionsprinzip aus. Dadurch wird die Ausführungsgeschwindigkeit des Codes durchschnittlich nur um ca. 73 % verlangsamt, während die in Tabelle 4.1 dargestellten Fehlertypen bei ihrem Auftreten erkannt werden können. Der große Vorteil dabei ist, dass die Fehlermeldung explizit zwischen den dargestellten Fehlertypen unterscheidet. Auch wenn `use after scope` und `use after return` einen ähnlichen Fehler beschreiben, lässt sich dieser durch die Unterscheidung wesentlich leichter lokalisieren. [39]

Tabelle 4.1: Durch ASan erkennbare Fehlertypen [40]

Fehlertyp	**Erklärung**
Use after free	Zugriff auf Speicher, der aktiv dealloziert wurde
Heap-, Stack-, Global-Bufferoverflow	Zugriff außerhalb des Speicherbereichs
Use after scope	Zugriff auf Speicher, der nur lokal in anderem Scope existierte
Use after return	Zugriff auf Speicher, der nur lokal in einer Funktion existierte
Memory Leaks	Speicher, der nicht genutzt und nicht freigegeben wird

Die Funktionsweise von Address-Sanitizern besteht darin, dass die standardmäßigen `malloc` und `free` Funktionen ersetzt werden. Der Speicherbereich um die durch `malloc` belegten Regionen wird als `poisoned` markiert. Der durch `free` freigesetzte Speicherbereich wird in Quarantäne verschoben und gilt ebenfalls als `poisoned`. Jeder Speicherzugriff wird entsprechend Listing 4.1 um eine Abfrage erweitert, ob der entsprechende Speicherzugriff als `poisoned` gilt. [41]

```
1 if (IsPoisoned(address)) {
2   ReportError(address, kAccessSize, kIsWrite);
3 }
4 *address = ...;  // or: ... = *address;
```

Listing 4.1: Speicherzugriff nach Instrumentierung mit ASan [41]

Um feststellen zu können, ob ein Byte im Speicher als poisoned gilt, nutzen Address-Sanitizer, ähnlich wie andere Tools zur Erkennung von Speicherfehlern, das Konzept von Shadow Memory. Hierfür wird der Adressbereich des Speichers in Memory und Shadow Memory unterteilt. Der Memory wird normal vom Anwendungscode genutzt, während der Bereich des Shadow Memory dazu genutzt wird, um dort Metadaten über den Speicherbereich der Anwendung abzulegen. Wenn ein Byte aus dem Speicherbereich der Anwendung als poisoned markiert wird, dann wird dies im Shadow Memory hinterlegt und ist somit entsprechend Listing 4.2 abprüfbar.

```
1 shadow_address = MemToShadow(address);
2 if (ShadowIsPoisoned(shadow_address)) {
3   ReportError(address, kAccessSize, kIsWrite);
4 }
```

Listing 4.2: Instrumentierung zum Anlegen und Prüfen von poisoned Bytes [41]

Im Gegensatz zu anderen Tools zeichnen sich Address-Sanitizer durch ein sehr effizientes Mapping zwischen Anwendungsspeicher und Shadow Memory aus. Hierbei werden die Metadaten über acht Byte des Anwendungsspeichers in ein Byte des Shadow Memory codiert, wodurch der dadurch entstehende Overhead nur einem Achtel des Adressbereichs entspricht. Dies wird möglich, da sich der Zustand (poisoned/unpoisoned) von acht ausgerichteten Bytes des Anwendungsspeichers (ein 64-Bit Wort) durch neun verschiedene Werte beschreiben lässt. Sind alle acht Bytes des Speicherworts unpoisoned, dann ist der Wert des korrespondierenden Bytes im Shadow Memory null. Sind alle Bytes des Speicherworts poisoned, dann ist der Wert des korrespondierenden Bytes negativ. Hierbei werden verschiedene negative Werte genutzt, um zwischen heap, stack, global und freed Memory differenzieren zu können. Wenn die ersten k Byte des Speicherworts unpoisoned sind und die restlichen 8 - k Byte poisoned, dann ist der Wert des korrespondierenden Bytes k. Die entsprechende Codierung der Werte im Shadow Memory ist in Abbildung 4.1 veranschaulicht. [39]

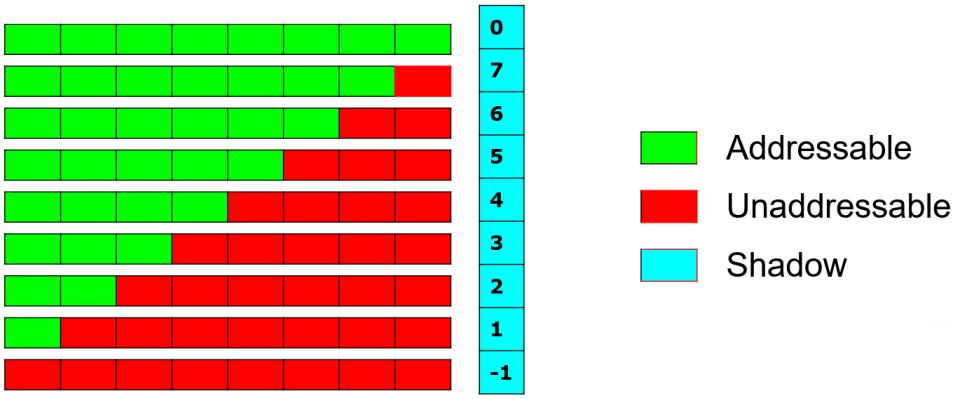

Abbildung 4.1: Codierung der Werte im Shadow Memory [39, Aus Folien d. Präsentation]

Die vom Compiler vorgenommene, erweiterte Instrumentierung ist in Listing 4.3 dargestellt. Wenn die an MemToShadow übergebene Adresse auf eine Adresse im Shadow Memory zeigt, dann kommt es hier zum Crash. Ist dies nicht der Fall und das gesamte Speicherwort gilt nicht als unpoisoned, dann erfolgt die Prüfung der einzelnen Bytes des Speicherwortes. Sollte hier eines der verwendeten Bytes als poisoned markiert sein, dann wird ein entsprechender Fehler erzeugt.

```
byte *shadow_address = MemToShadow(address);
byte shadow_value = *shadow_address;
if (shadow_value) {
  if (SlowPathCheck(shadow_value, address, kAccessSize)) {
    ReportError(address, kAccessSize, kIsWrite);
  }
}
```

Listing 4.3: Erweiterte Instrumentierung zum Anlegen und Prüfen von poisoned Bytes [41]

Das gesamte Prinzip wird anhand der Darstellung des Adressraums in Abbildung 4.2 veranschaulicht. Die Adressen aus dem Speicherbereich der Anwendung werden mit `MemToShadow` in den korrespondierenden Bereich des `Shadow Memory` gemappt. Im Adressbereich des `Shadow Memory` codiert ein Byte für acht korrespondierende Bytes des Anwendungsspeichers, ob diese `poisoned/unpoisoned` sind. Wird versucht `MemToShadow` mit einer Adresse aus dem `Shadow Memory` aufgerufen, wobei versucht wird, das Mapping mit einer Adresse aus dem `Shadow Memory` durchzuführen, dann kommt es zu einem Crash. Dies wird herbeigeführt, da das Mapping in diesem Fall auf einen Speicherbereich erfolgt, der als unerreichbar markiert wurde. Abschließend lässt sich entsprechend Listing 4.3 der Wert eines Bytes im `Shadow Memory` überprüfen. Auf diese Weise lässt sich feststellen, ob die korrespondierenden acht Bytes des Anwendungsspeichers `poisoned/unpoisonend` sind. [39]

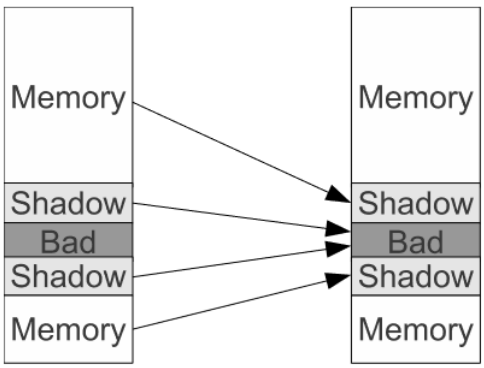

Abbildung 4.2: AddressSanitizer Memory Mapping [39]

Address-Sanitizer sind fester Bestandteil von Clang und GCC, sie können durch das Setzen der `-fsanitize=address` Compiler-Flag aktiviert werden. Die Anwendung ist auch ohne die Verwendung eines Fuzzers möglich.

4.2 Memory Sanitizer

Bei Memory-Sanitizern (MSan) handelt es sich um ein Tool zur Erkennung von Fehlern durch die Nutzung von uninitialisiertem Speicher. Derartige Fehler sind beim Testen schwer zu finden, da sie nicht zwingend zu einem Fehler während der Ausführung eines Programmes führen. Ähnlich wie bei Address-Sanitizern, findet eine Instrumentierung des Codes während der Kompilierung statt. Dabei wird ebenfalls das Konzept von `Shadow Memory` verwendet, um die Fehlererkennung zu ermöglichen. Hierbei verlangsamt sich die Ausführungsgeschwindigkeit des Codes um das 2,5-fache und der Speicherbedarf steigt auf das Doppelte an. Im Vergleich zu ähnlichen Tools zeichnen sich Memory-Sanitizer damit durch ihre schnelle Ausführungsgeschwindigkeit aus. Außerdem ermöglichen sie es

durch einen Origin-Tracking-Mode entsprechende Fehler besser lokalisieren zu können. [42]
Im Gegensatz zu ASan erfordern Memory-Sanitizer ein eins zu eins Mapping zwischen dem Speicherbereich der Anwendung und dem `Shadow Memory`. Das bedeutet, für jedes Bit im Speicherbreich der Anwendung gibt es ein korrespondierendes Bit im `Shadow Memory`, woraus der doppelte Speicherbedarf resultiert. Hierbei wird in jedem Bit des `Shadow Memory` codiert, ob das korrespondierende Bit im Speicherbereich der Anwendung initialisiert (Wert: 0) oder uninitialisiert (Wert: 1) ist. Jeder neu allozierte Speicher erhält einen korrespondierenden `Shadow Memory`, der bis zu der ensprechenden Initialisierung mit Einsen gefüllt ist. Dabei erlauben MSan einige als sicher geltende Operationen mit uninitialisiertem Speicher, wie beispielsweise das Kopieren von uninitialisiertem Speicher. Dies kann im Zuge von Optimierungen des Compilers auftreten, wenn innerhalb eines Kopierkonstruktors, die uninitialisierten Bytes im Padding zwischen zwei Klassenvariablen mit geladen werden. Für das folgende Beispiel könnte der Compiler dabei direkt einen einzelnen 8-Byte Speicherzugriff im Kopierkonstruktor durchführen.

```
class A { char x; int y;}
```

Damit eine derartige Operation keine Fehler erzeugt, werden dabei die `Shadow Values` des geladenen Speicherbereichs in temporären Variablen des Compilers zwischengespeichert. Am Ende des Kopiervorgangs werden diese in den korrespondierenden `Shadow Memory` des neu gefüllten Anwendungsspeicherbereichs übertragen. Andere Operationen, wie z.B. Zweig-Konditionen oder das Dereferenzieren von Pointern, erfordern zwingend initialisierten Speicher bei ihren Operanden. Wird diesen Anweisungen uninitialisierter Speicher als Argument übergeben, dann erzeugt MSan einen Fehler. Alle restlichen Operationen erzeugen einen neuen `Shadow Value` für das Ergebnis der Operation. Der neue `Shadow Value` wird abhängig von den `Shadow Values` der Operanden berechnet, wobei die genaue Berechnung hier von der entsprechenden Operation abhängig ist. [42]
Memory-Sanitizer sind ein fester Bestandteil der LLVM-Clang-Toolchain, wobei sie bei GCC nicht verfügbar sind. Sie können bei Clang durch das Setzen der Compiler-Flag `-fsanitize=memory` aktiviert werden. Die Anwendung ist ebenfalls ohne die Verwendung eines Fuzzers möglich . Der Origin-Tracking-Mode kann über die Compiler-Flag `-fsanitize-memory-track-origins` aktiviert werden. Er verlangsamt die Ausführungsgeschwindigkeit zusätzlich um das 1,5-2,5-fache. Durch die Aktivierung kann die Nutzung jedes uninitialisierten Wertes zu der Stelle zurückverfolgt werden, wo der entsprechende Speicher alloziert wurde. [43]

4.3 Undefined-Behavior Sanitizer

Im Gegensatz zu Programmiersprachen wie Java, die weitgehend als sicher gelten, erlauben C und C++ undefiniertes Verhalten. Dies führt dazu, dass Code mit eigentlich undefiniertem Verhalten nicht zwingend zu einem Kompilierfehler führt. Stattdessen lässt sich das Programm kompilieren und ausführen. Es ist dabei die Aufgabe des Programmierers, die Auswirkungen von Operationen mit undefiniertem Verhalten zu beachten, sodass das Programm keine unerwünschten Resultate liefert. Ein gutes Beispiel für undefiniertes Verhalten in C/C++, sind Überschreitungen von Zahlenbereichen der Datentypen innerhalb arithmetischer Operationen. So führt die Addition bei einem Integer mit `INT_MAX+1` nicht garantiert zu `INT_MIN`, stattdessen ist das Verhalten nicht fest definiert. In diesem Fall spricht man von einem sog. `Signed Integer Overflow` und er würde höchstens eine Warnung, statt eines Kompilierfehlers erzeugen aber ggf. Resultate von Operationen verfälschen. Ähnliches gilt für den Zugriff auf uninitialisierten Speicher, wodurch stattdessen die Werte aus dem Speicher verwendet werden, die vor der Initialisierung an dieser Stelle stehen, sodass das

Verhalten damit undefiniert ist. Darüber hinaus gibt es eine Vielzahl weiterer Operationen, die undefiniertes Verhalten erzeugen. Die Zulässigkeit von undefiniertem Verhalten ist in C/C++ explizit erwünscht, da dadurch viele maschinennahe Operationen möglich werden und äußerst performanter Code generiert werden kann. Beispielsweise führt das undefinierte Verhalten beim `Signed Integer Overflow` dazu, dass ein Compiler $X + 1 > X$ automatisch zu true evaluieren kann. Dies ist nur möglich, da `INT_MAX+1` nicht garantiert zu `INT_MIN` wird. Auf der anderen Seite ist die Zulässigkeit von undefiniertem Verhalten eine häufige Fehlerquelle, wobei Programmierer undefiniertes Verhalten nicht zuverlässig vermeiden können. Häufig bleiben derartige Fehler unbemerkt, sodass Programme im Verborgenen unerwünschtes Verhalten aufweisen. [44] [45]
Aus diesem Grund wurden Undefined-Behavior Sanitizer (UBSan) entwickelt. Unter diesem Begriff sind eine Vielzahl von Tools zur Erkennung unterschiedlicher Arten von undefiniertem Verhalten zusammengefasst. Ähnlich wie bei anderen Sanitizern findet auch hier eine Instrumentierung des Codes während des Kompiliervorgangs statt und es kommt zu einer Verlangsamung der Ausführungsgeschwindigkeit. In der dazugehörigen Dokumentation ist eine Liste aller verfügbaren UBSans, des erkennbaren undefinierten Verhaltens und der für die Aktivierung nötigen Compiler-Flags dargestellt [46]. UBSans sind sowohl unter Clang als auch GCC verfügbar.

4.4 Partielle Deaktivierung von Sanitizer Instrumentierung

Bei der Verwendung von Sanitizern kann es aus verschiedenen Gründen erforderlich sein, dass die Instrumentierung für bestimmte Codeabschnitte partiell deaktiviert werden soll bzw. muss. Zum einen kann es sein, dass in bestimmten Teilen des Codes bewusst Operationen durchgeführt werden, die von einem der Sanitizer als Fehler interpretiert werden. Dies könnten beispielsweise Arten von Speicheroperationen mit Pointerarithmetik sein, die jedoch innerhalb des Quellcodes eine gewünschte Funktion erfüllen. Ein weiterer Grund könnte darin bestehen, dass man die durch die Sanitizer hervorgerufene Verlangsamung der Ausführungsgeschwindigkeit begrenzen möchte. So kann man die Instrumentierung überall dort im Code deaktivieren, wo man diese zur Fehlererkennung nicht benötigt. Dadurch kann der Fuzz-Test mehr Testfälle pro Zeiteinheit generieren und ist insgesamt effektiver. Zuletzt wäre es denkbar, dass man während des Fuzz-Tests mit Hilfe eines Sanitizers einen Fehler gefunden hat. Da der Sanitizer bei jedem weiteren Erreichen dieser Stelle erneut einen Crash erzeugt, wird der Fuzz-Test vermutlich an dieser Stelle stecken bleiben und mit neuen Testfällen nicht tiefer in diesem Zweig des Testobjekts vordringen. Da der Fehler in der Regel nicht unverzüglich behoben wird und der Tester den Fuzz-Test fortsetzen bzw. im Code weiter vordringen möchte, können hier die Sanitizer für den Codeabschnitt des gefundenen Fehlers deaktiviert werden. Auf diese Weise können die während des Fuzz-Tests gefundenen Fehler sukzessive dokumentiert und anschließend übergangen werden.
Um die Instrumentierung von Sanitizern im größeren Ausmaß partiell zu deaktivieren, kann man unterschiedliche Dateien je nach Bedarf mit und ohne Sanitizer kompilieren und diese anschließend linken. Auf diese Weise lassen sich die Sanitizer auf Dateiebene partiell deaktivieren. Darüber hinaus ist mit Clang eine filigranere Deaktivierung der Instrumentierung für einzelne Funktionen möglich. Dies kann dem Compiler signalisiert werden, indem eines der folgenden Funktionsattribute vor die entsprechende Funktion geschrieben wird [46] [47] [48].

```
__attribute__((no_sanitize("address")))          // Disable ASan
__attribute__((no_sanitize("memory")))           // Disable MSan
```

```
__attribute__((no_sanitize("undefined")))           // Disable UBSan
// Statt "undefined" kann Compiler-Flag von spezifischem UBSan verwendet werden
__attribute__((disable_sanitizer_instrumentation))  // Disable all Sanitizers
```

Innerhalb des Codes kann die partielle Deaktivierung für eine einzelne Funktion entsprechend Listing 4.4 aussehen. In diesem Beispiel erzeugt erst der zweite Funktionsaufruf einen Fehler bei der Anwendung von Address Sanitizern.

```
1  __attribute__((no_sanitize("address")))
2  int firstFnc(int* array, int index) {
3    return array[index];
4  }
5
6  int secondFnc(int* array, int index) {
7    return array[index];
8  }
9
10 int main() {
11   int *array = new int[10]();
12   firstFnc(array, 12);  // Keinen Fehler durch ASan
13   secondFnc(array, 12); // Fehler durch ASan
14 }
```

Listing 4.4: Partielle Deaktivierung von Sanitizern für einzelne Funktion

4.5 Sanitizer und Fuzz-Tests

Nachdem die wichtigsten Sanitizer vorgestellt wurden, folgen noch einige allgemeine Informationen für die Kombination mit Fuzz-Tests. Grundsätzlich sollten Fuzz-Tests immer mit einem der Sanitizer ausgeführt werden, da diese die eigentliche Fehlererkennung für den Fuzzer leisten. Nur wenn das Testobjekt crasht, kann dies durch den Fuzzer registriert werden und dieser erzeugt anschließend einen Report mit den entsprechenden Inputs, die zum Crash geführt haben. Ohne den Einsatz von Sanitizern wäre die Fehlererkennung von Fuzz-Tests sehr begrenzt und würde sich auf die Fälle beschränken, die das Testobjekt grundsätzlich crashen lassen. Dies sollte in der Regel bereits durch konventionelle Tests erkannt werden.
Durch den Einsatz der Sanitizer kann eine Vielzahl an Fehlerklassen erkannt werden und diese liefern zusätzlich detaillierte Informationen zu den erkannten Fehlern. Grundsätzlich sollten diese mit der Compiler-Flag `-g` bzw. `-g3` kombiniert werden. Diese sorgt für die Generierung von zusätzlichen Debug-Informationen, wobei `-g3` die höchste Stufe mit den meisten Informationen darstellt. Nur wenn diese Flag aktiviert ist, erhält man beispielsweise bei einem Crash durch ASan einen Stacktrace mit den Informationen zu entsprechenden Codezeilen. Diese sind äußerst hilfreich, um die gefundenen Fehler im Anschluss leichter bewerten und korrigieren zu können. Des Weiteren ist man nicht auf die Verwendung einer Sanitizerklasse beschränkt. Theoretisch könnte man eine Instrumentierung aller Sanitizer gleichzeitig durchführen und anschließend damit einen Fuzz-Test ausführen. Von diesem Vorgehen ist jedoch abzuraten, da jede zusätzliche Instrumentierung die Ausführungsgeschwindigkeit weiter verlangsamt und der Fuzz-Test damit weniger Testfälle pro Zeiteinheit durchführt. In der Regel ist es ratsam den Fuzz-Test zunächst mit ASan zu beginnen, da damit eine Vielzahl der in C/C++ häufig vorkommenden Fehlertypen abgedeckt werden. Abschließend sei noch darauf hingewiesen, dass Sanitizer auch ohne einen Fuzzer eingesetzt werden können. In Kombination mit Fuzz-Tests sind diese durch die große Anzahl generierter Testfälle jedoch besonders effektiv.

5 Structure-Aware Fuzzing mit Protocol Buffern

In diesem Kapitel soll die Durchführung von Structure-Aware Fuzzing durch Kombination von coverage-guided Fuzzern mit Googles Protocol Buffern untersucht werden. Da für den Einsatz von Custom Mutatoren unter AFL/AFL++ sehr viele Schritte notwendig sind, wird hierfür ausschließlich libFuzzer betrachtet. Die Kombination mit AFL/AFL++ ist aber möglich. Es steht die Frage im Fokus, ob und inwieweit sich die Implementierung von derartigen Fuzz-Tests damit vereinfachen bzw. automatisieren lässt. Es soll zunächst erklärt werden, was Protocol Buffer sind und wie diese eingesetzt werden. Anschließend wird gezeigt, wie diese über den `libprotobuf-mutator` mit Fuzzern kombiniert werden können. Diesbezüglich werden praktische Anwendungstechniken aufgezeigt und die Stärken dieser Art von Fuzz-Tests untersucht.

5.1 Protocol Buffer

Protocol Buffer (Protobuf) sind ein von Google entwickelter, sprach- und plattformneutraler Mechanismus zur Serialisierung von typisierten, strukturierten Daten. Im Prinzip handelt es sich dabei um eine Beschreibungssprache, in der festgelegt wird, wie eine Datenstruktur aufgebaut sein sollen. Diese kann von einem Protobuf-Compiler interpretiert werden, sodass dieser auf Basis der Beschreibung, den Quellcode der definierten Datenstrukturen samt dazugehöriger Interfaces erzeugt. Die so erzeugten Datenstrukturen eignen sich durch die Möglichkeit zur sprach- und plattformneutralen Serialisierung für den Einsatz in der Netzwerkkommunikation, da eine derart definierte und serialisierte Nachricht von Protobuf-Quellcode unterschiedlicher Programmiersprachen interpretiert werden kann. Dabei unterstützt Protobuf eine Vielzahl an Programmiersprachen wie C++, C#, Java oder Python. Abgesehen davon kann Protobuf auch nur zum Speichern von Daten verwendet werden. [49] Protobuf ist als Open-Source-Software frei verfügbar, wurde in Teilen aber unter der Apache-Lizenz 2.0 [50] und in anderen Teilen unter der 3-Klausel-BSD-Lizenz [51] veröffentlicht. Beide Lizenzen erlauben das Verwenden, Modifizieren und Verteilen. Lediglich bei der BSD-Lizenz ist darauf zu achten, dass der ursprüngliche Copyright-Vermerk nicht entfernt werden darf.
Das grundlegende Prinzip von Protobuf sieht so aus, dass die Datenstrukturen zunächst in einer oder mehrerer `.proto` Dateien definiert werden. Hierfür gibt es eine festgelegte Syntax, auf die im weiteren Verlauf genauer eingegangen wird. Anschließend können diese `.proto` Dateien mit dem `Protoc` Compiler kompiliert werden, um daraus den entsprechenden Quellcode zu generieren. Für C/C++ erzeugt jede `.proto` Datei eine `.pb.cc` Datei samt dazugehörigem Header. Diese können innerhalb des eigenen Projektes im Quellcode inkludiert und verwendet werden. Für die Serialisierung/Deserialisierung bzw. das Übertragen und Empfangen stehen die sprachspezifischen Laufzeitbibliotheken von Protobuf zur Verfügung. Der gesamte Ablauf ist in Abbildung 5.1 veranschaulicht. [49] Für die Nutzung von Protobuf in Fuzz-Tests ist ausschließlich die Definition von typisierten Datenstrukturen mit anschließender Generierung von Quellcode relevant, sodass der Teil der Serialisierung nicht weiter betrachtet wird. Darüber hinaus wird sich ausschließlich auf C/C++ fokussiert.

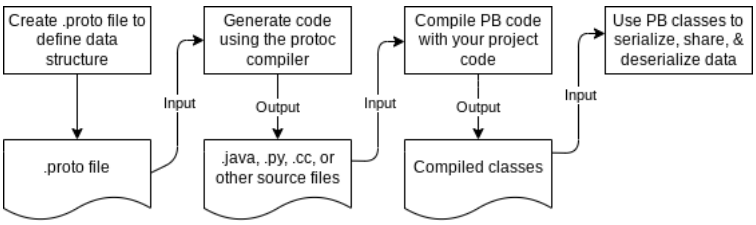

Abbildung 5.1: Workflow von Protobuf [49]

5.1.1 Definition von Datenstrukturen

Vorab sei an dieser Stelle auf die ausführliche Dokumentation [37] und das dazugehörige C++ Tutorial [52] von Protobuf verwiesen. Dort lassen sich alle folgenden und viele weiterführende Informationen nachlesen. In diesem Abschnitt werden ausschließlich die Dinge aufgegriffen, die für den Einsatz von Protobuf in Kombination mit Fuzz-Tests relevant sind. Um Datenstrukturen definieren zu können, erstellt man zunächst eine `.proto` Datei mit beliebigem Namen und öffnet diese mit einer IDE oder einem Texteditor. Zu Beginn muss die Syntax festgelegt werden. Hier stehen die Versionen `proto2` [53] und `proto3` [54] zur Verfügung. Die Syntax `proto3` kann als eine simplifizierte Form von `proto2` verstanden werden, wobei semantisch gleiche Konstrukte in beiden Versionen nach der Serialisierung zur gleichen Binärrepräsentation führen [55]. Da die Serialisierung innerhalb der Anwendung von Fuzz-Tests keine Rolle spielt und nur ein Bruchteil der Funktionalitäten während der Definition benötigt werden, sind viele der Feinheiten beider Syntax-Versionen irrelevant. Für den weiteren Verlauf dieser Arbeit wird `proto2` verwendet und dies kann wie folgt am Anfang der `.proto` Datei festgelegt werden.

```
syntax = "proto2";
```

Über das `package` Keyword hat man die Möglichkeit zu Anfang der Datei ein Namespace festzulegen, sodass alle Datenstrukturen dieser Datei darin gekapselt werden. Auf diese Weise können Namenskonflikte verhindert werden. Das folgende Beispiel würde das Namespace Ostfalia definieren, sodass alle darunter definierten Strukturen mit `Ostfalia::` Präfix zu erreichen sind.

```
package Ostfalia;
```

Die Definition der Datenstrukturen ist im Grunde einer C++ oder Java Syntax sehr ähnlich. Hierfür werden die wichtigsten Details anhand des in Listing 5.1 dargestellten Beispiels erläutert.

```
1  message Person {
2    required string name = 1;
3    required int32 id = 2;
4    optional string email = 3 [default = "m.mustermann@ostfalia.de"];
5
6    message PhoneNumber {
7      optional string number = 1;
8    }
9    repeated PhoneNumber phones = 4;
10 }
11 message AddressBook {
12   repeated Person people = 1;
13 }
```

Listing 5.1: Definition von Datenstrukturen in .proto Datei

Eine Datenstruktur wird wie ein Struct definiert, wobei hier das `message` Keyword zu verwenden ist. Jede mit diesem Keyword definierte Datenstruktur wird nach dem Kompilieren im generierten Quellcode als eigene Klasse erzeugt. Die generierten Klassen erben alles nötige von der Oberklasse `google::protobuf::Message`. Innerhalb einer Datenstruktur können entsprechend Zeile 2-4 Member-Variablen (Felder) definiert werden, wobei die zur Verfügung stehenden Datentypen der Dokumentation entnommen werden können [53]. Hierbei sind viele der Standarddatentypen von C++ vorhanden und im Zweifel kann ein fehlender Datentyp jederzeit über den `bytes` Typ als Bytesequenz repräsentiert werden. Diese kann innerhalb des Quellcodes mit einem Typecast in den gewünschten Datentyp umgewandelt werden. Vor dem Datentypen einer Variablen wird ein sogenanntes Field Label angegeben. Die in der proto2-Syntax vorhandenen Field Label sind in Tabelle 5.1 dargestellt. Wenn ein optionales Feld nicht gesetzt wird, dann wird ein Default-Wert verwendet. Daher sollte man bei optionalen Feldern einen Default-Wert wie in Zeile 4 definieren. Tut man dies nicht, dann wird das Feld automatisch mit einem datentypspezifischen Default-Wert gesetzt. Bei numerischen Datentypen ist dies eine Null, bei Strings ein leerer String und bei boolschen Datentypen ist es false. Für die Verwendung der definierten Datenstrukturen in Fuzz-Tests wurde ausschließlich `required` statt `optional` verwendet, da den Variablen in jedem Fall Werte zugewiesen werden sollen. Die grundlegende Idee hinter `required` ist jedoch, dass eine Datenstruktur als uninitialisiert gewertet wird, wenn einem `required` Feld kein Wert zugewiesen ist. Die Serialisierung einer uninitialisierten Datenstruktur erzeugt einen Fehler aber abgesehen davon verhält sich `required` exakt wie `optional`.

Tabelle 5.1: Field Labels der proto2 Syntax in Protobuf [53]

Field Label	**Erklärung**
optional	Dieses Feld kann aber muss nicht mit Wert gesetzt werden
repeated	Dieses Feld kann gar nicht oder mehrfach vorkommen (wie Array)
map<datatype1, datatype2>	Wie repeated aber mit key/value-Paar
required	Dieses Feld muss mit einem Wert gesetzt werden

Hinter jeder Variablen/Feld muss eine einzigartige Feldnummer in der Definition angegeben werden. Diese darf sich innerhalb einer einzelnen Datenstrutur nicht wiederholen. Bei dieser Feldnummer handelt es sich nicht um einen Variablenwert, sondern um einen Identifikator. Dieser wird bei der Serialisierung verwendet, um einzelne Felder identifizieren zu können. Darüber hinaus können Datenstrukturen auch ineinander verschachtelt werden, dies ist in Zeile 6-8 dargestellt. Außerdem kann eine definierte Datenstruktur entsprechend Zeile 9 bzw. Zeile 12 wie einen zusammengesetzten Datentyp verwendet werden. Innerhalb einer einzelnen `.proto` Datei können auf diese Weise mehrere Datenstrukturen aufeinmal definiert werden. Möchte man Datenstrukturen aus einer anderen `.proto` Datei einbinden, dann kann man diese wie folgt importierten.

```
import "myproject/other_protos.proto";
```

5.1.2 Nutzung von Protocol Buffern im Code

Wenn man Protocol Buffer alleine nutzen möchte, dann kann man der Installationsanleitung des GitHub Repositories folgen [56]. Im weiteren Verlauf dieser Arbeit wird jedoch eine Anleitung gegeben, wie man mit einem Dockercontainer eine komplette Umgebung mit allen nötigen Tools für Structure-Aware Fuzzing mit Protocol Buffern aufsetzen kann. An dieser Stelle soll davon ausgegangen werden, dass der `protoc` Compiler samt Bibliotheken in nutzbarer Form vorliegt. Um die

Nutzung von Protobuf im Quellcode demonstieren zu können, wird die in Listing 5.2 dargestellte Datenstruktur in einer `.proto` Datei definiert.

```
syntax = "proto2";

message TEST {
  required uint32 x = 1;
  required string y = 2;
}
```

Listing 5.2: Einfache Datenstruktur in beispiel.proto Datei

Sofern der `protoc` Compiler in den Umgebungsvariablen hinterlegt ist, kann die `beispiel.proto` Datei wie folgt über die Konsole im gleichen Ordner kompiliert werden. Es sei darauf hingewiesen, dass dem Compiler explizit mitgeteilt werden muss, in welcher Programmiersprache die Datenstrukturen generiert werden sollen. Hierfür wird die Option `cpp_out` für C++ genutzt, wobei andere unterstützte Programmiersprachen analog festgelegt werden.

```
protoc --cpp_out=<Zielpfad> beispiel.proto
```

Nach dem Kompiliervorgang wird eine `beispiel.pb.cc` Datei samt dazugehörigem `beispiel.pb.h` Header erzeugt. Diese können anschließend wie gewohnt im eigenen Quellcode inkludiert werden. Selbst eine so einfache Datenstruktur erzeugt bereits relativ viel Code, da in den erzeugten Klassen neben der Variablen alle möglichen Funktionalitäten mit erzeugt werden. Die generierte Datenstruktur kann entsprechend Listing 5.3 wie eine gewöhnliche Klasse genutzt werden.

```
#include "test.pb.h"
#include <iostream>

int main()
{
    TEST obj;
    obj.set_x(123);
    obj.set_y("Hello World!");
    std::cout << obj.x() << std::endl;
    std::cout << obj.y() << std::endl;
    return 0;
}
```

Listing 5.3: Interaktion mit generiertem Protobuf

Die wichtigsten Funktionen für den Zugriff auf einfache Variablen mit dem `optional` und `required` Label sind in Tabelle 5.2 aufgelistet. Statt des in der Tabelle verwendeten `VARNAME`, muss der Name der entsprechenden Variable verwendet werden.

Tabelle 5.2: Die wichtigsten Zugriffsfunktionen für einfache Variablen [57]

Funktion	**Erklärung**
has_VARNAME( )	Gibt true, wenn Wert der Variable gesetzt wurde. Bei default false
VARNAME( )	Gibt den Wert der Variablen
set_VARNAME(<value>)	Setzt den Wert der Variablen
clear_VARNAME( )	Entfernt den Wert der Variablen, VARNAME() gibt wieder default
mutable_VARNAME( )	(Nur bei Objekten!) Gibt Pointer auf das Objekt

Wenn man mit dem `repeated` Label Arrays von Datentypen in den Strukturen aufbaut, dann verhalten diese sich wie dynamische Arrays. Hierfür können die in Tabelle 5.3 aufgelisteten Funktionen für die Interaktion genutzt werden.

Tabelle 5.3: Die wichtigsten Zugriffsfunktionen für Arrays [57]

Funktion	**Erklärung**
VARNAME_size( )	Gibt die Anzahl der Elemente zurück
VARNAME(<index>)	Gibt den Wert des Elements mit geg. Index
set_VARNAME(<index>, <value>)	Setzt den Wert des Elements mit geg. Index
add_VARNAME(<value>)	Fügt neues Element mit geg. Wert ans Ende
clear_VARNAME( )	Entfernt alle Elemente
mutable_VARNAME( )	Gibt Pointer auf das Objekt

Mit den dargestellten Funktionen sind alle wichtigen Manipulationen der Datenstrukturen möglich. Es sei jedoch darauf hingewiesen, dass es noch weitere Funktionen gibt, die für diese Arbeit nicht aufgegriffen wurden.

5.1.3 Bewertung des allgemeinen Einsatzes von Protocol Buffern

Unabhängig vom Einsatz der Protocol Buffer in Kombination mit Fuzz-Tests, ist der grundsätzliche Einsatz zur Definition großer komplexer Datenstrukturen eine Überlegung wert. Zum einen hat man durch die Definitionen innerhalb der `.proto` Dateien eine übersichtliche Abstraktion der definierten Datenstrukturen, sodass eine vom spezifischen Einsatz losgelöste Betrachtung möglich wird. Zusätzlich lassen sich die definierten Datenstrukturen je nach Bedarf ohne großen Aufwand in andere Projekte und andere Programmiersprachen übernehmen. Hierbei ist der Implementierungsaufwand signifikant verkürzt, da neben der eigentlichen Datenstruktur zahlreiche Funktionen zur Manipulation und Verarbeitung direkt mit implementiert werden. Darüber hinaus hat man den großen Vorteil, dass sich die definierten Datenstrukturen sehr leicht erweitern bzw. verändern lassen. Nach der Anpassung der Definitionen innerhalb der `.proto` Datei sorgt der Compiler für alle wichtigen Veränderungen innerhalb des Quellcodes. Auf diese Weise wird das Auftreten von Fehlern durch übersehene Anpassungen, während der manuellen Erweiterung von komplexen Datenstrukturen verringert. Bei der nachträglichen Veränderung der Definitionen sollten die folgenden Regeln eingehalten werden, um eine Vorwärts- und Rückwärtskompatibilität besonders im Einsatz als Kommunikationsprotokoll gewährleisten zu können. [49]

- Die Feldnummern existierender Felder sollten nicht verändert werden
- Es sollten keine `required` Felder hinzugefügt oder entfernt werden
- `optional` und `repeated` Felder können entfernt werden
- `optional` und `repeated` Felder können mit neuer Feldnummer hinzugefügt werden

Die größte Stärke und der eigentliche Hauptzweck der Protocol Buffer besteht darin, dass dadurch neben der eigentlichen Datenstrukturen automatisch ein plattform- und sprachunabhängiger Kommunikationsmechanismus implementiert wird. Durch die Möglichkeit zur Serialisierung/Deserialisierung der definierten Datenstrukturen wird die Kommunikation zwischen unterschiedlichen Systemen in

unterschiedlichen Programmiersprachen möglich. Derartige Konzepte sind beispielsweise in der Robotik weit verbreitet und gehören zu den wichtigsten Funktionen, die Frameworks wie das Robot Operating System bereitstellen [58]. Dadurch wird der Aufbau verteilter Systeme möglich und es können die Vorteile unterschiedlicher Programmiersprachen miteinander kombiniert werden. Die rechenintensiven Teile eines Systems können auf diese Weise in performanten Programmiersprachen wie C/C++ implementiert werden, während an anderen Stellen Programmiersprachen wie Python, mit der großen Anzahl an frei verfügbaren Bibliotheken, genutzt werden können.

5.2 Von Protocol Buffern zum Fuzz-Test

Es wurde dargestellt was Protocol Buffer sind, wie mit diesen Datenstrukturen definiert werden können und wozu sie eingesetzt werden. Nun bleibt die Frage zu klären, wie sich diese zur Realisierung von Fuzz-Tests verwenden lassen. Hierfür gibt es mit dem `libprotobuf-mutator` (LPM) [59] eine Library für die Mutation von Protocol Buffern. Diese lässt sich als Custom Mutator mit coverage-guided Fuzzern wie libFuzzer und AFL/AFL++ kombinieren und kann nach Bedarf manuell auf das Testobjekt angepasst werden. In der grundlegendsten Form bietet die darin bereitgestellte `Mutator` Klasse die Basis für zufällige Mutation der als Protocol Buffer definierten Datenstrukturen. Dadurch wird es entsprechend Abbildung 5.2 sehr einfach möglich, gezielt einzelne Mutationen an definierten Feldern des Protocol Buffers zu applizieren. [59] Die in der Abbildung dargestellte `orig.txt` kann hier als Beispiel für einen initialen Input im Corpus des Fuzzers dienen.

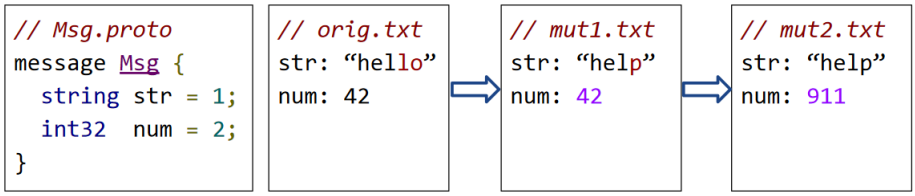

Abbildung 5.2: Beispiel für die Mutation einer protobuf Datenstruktur [60]

In der dargestellten Form ist der Vorteil noch nicht direkt ersichtlich. Es gilt zu bedenken, dass Fuzzer wie libFuzzer entsprechend Listing 5.4 die Inputs als Byte-Array bereitstellen. Das Problem ist allerdings, dass die wenigsten Testobjekte ein Byte-Array als Input akzeptieren. Stellt man libFuzzer den gleichen initialen Input wie in `orig.txt` zur Verfügung, dann wird dieser für ein derart einfaches Beispiel, die Mutationen leicht nachstellen können. Dennoch muss der Tester anschließend die Daten aus dem generierten Byte-Array auf die entsprechenden Inputs der Funktion verteilen. Dies erfordert je nach Komplexität und Anzahl der Inputs einen großen Arbeitsaufwand und stellt den Tester vor Probleme. Betrachtet man beispielsweise den als Input benötigten String. Wird dieser in seiner Länge mutiert, dann wird es schwer zu entscheiden, wie viele der generierten Bytes zum String gehören.

```
1 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
2   // selten akzeptiert ein Testobjekt ein Byte-Array als Input
3   DoSomethingInterestingWithMyAPI(Data, Size);
4   // realisitischeres Testobjekt, Bytes muessten auf Inputs verteilt werden
5   DoSomethingInterestingWithMyAPI(str, num);
6   return 0;
7 }
```

Listing 5.4: Aufruf eines Fuzz-Targets mit libFuzzer [24]

Mit Protocol Buffern und LPM lässt sich dieses Problem leicht beseitigen, wodurch sich der Arbeitsaufwand für den Tester signifikant reduziert. Angenommen man möchte die fiktive Funktion `add_person()` aus Listing 5.5 testen, welche einige Standarddatentypen als Inputs erwartet.

```
void add_person(string vorname, string nachname, uint32_t id, float groesse
    )
{
    // do something
}
```

Listing 5.5: Fiktives Testobjekt add_person()

Für die Inputs kann ein Protocol Buffer entsprechend Listing 5.6 angelegt werden. Idealerweise sollte hierbei im gleichen Zug ein Corpus mit initalen, validen Inputs angelegt werden. Dies wird im Detail zu einem späteren Zeitpunkt beschrieben.

```
syntax = "proto2";

message person_data {
    required string vorname = 1;
    required string nachname = 2;
    required uint32 id = 3;
    required float groesse = 4;
}
```

Listing 5.6: Protocol Buffer für add_person()

Nachdem der dazugehörige Quellcode mit dem `protoc` Compiler generiert wurde, kann dieser inkludiert werden und der Fuzz-Test mit libFuzzer kann für die `add_person()` Funktion entsprechend Listing 5.7 aufgebaut werden. Es ist durch den Einsatz der Protocol Buffer sehr leicht, die mutierten Daten auf die entsprechenden Inputs des Testobjekts zu verteilen, da die Struktur der Daten bekannt ist. Das `DEFINE_PROTO_FUZZER` Makro liefert libFuzzer den Einstiegspunkt und sorgt dafür, dass libFuzzer als Custom Mutator LPM verwendet. Zum Starten des Fuzz-Tests muss der Quellcode nur mit Clang und der `-fsanitize=fuzzer` Flag kompiliert werden, sodass libFuzzer beim Ausführen mit dem Fuzz-Test beginnt. Dieses Beispiel soll vor allem veranschaulichen, wie leicht mit dieser Technik ein Fuzz-Test für Testobjekte durchgeführt werden kann, die eine Vielzahl an Inputs verschiedener Datentypen erwarten. Dabei kann die Anzahl an nötigen Inputs beliebig erhöht werden und es können durch verschachtelte Protocol Buffer auch zusammengesetzte Inputs nachgebildet werden.

```
#include "/libprotobuf-mutator/src/libfuzzer/libfuzzer_macro.h"
#include "person_data.pb.h"

DEFINE_PROTO_FUZZER(const person_data &inputFromFuzzer) {
    string vorname = inputFromFuzzer.vorname();
    string nachname = inputFromFuzzer.nachname();
    uint32_t id = inputFromFuzzer.id();
    float groesse = inputFromFuzzer.groesse();

    // Aufruf des Testobjekts mit mutierten Werten
    add_person(vorname, nachname, id, groesse);
}
```

Listing 5.7: Aufruf des Testobjekts mit Daten aus mutiertem Protocol Buffer

Die Kombination von Protocol Buffern mit Fuzzern markiert den Übergang zu Structure Aware Fuzzing und diese Technik ist in dieser einfachen Form bereits in der Lage, den Arbeitsaufwand bei der Implementierung zu reduzieren. Allerdings sind die von LPM, in der `protobuf_mutator::Mutator` Klasse bereitgestellten, virtuellen Feld-Mutator-Funktionen sehr simpel. Diese sind nicht testobjektspezifisch und sie machen sich die Struktur der Daten nicht weiter zu Nutze. Für bessere Ergebnisse sollten die Mutatoren dieser Klasse durch testobjektspezifische, oder die von den Fuzzern selbst bereitgestellten Mutatoren überschrieben werden. Das Prinzip soll in Listing 5.8 verdeutlicht werden. So kann man eine eigene Mutator-Klasse erstellen, die alles Wichtige von `protobuf_mutator::Mutator` erbt. Innerhalb der eigenen Mutator-Klasse können dann die entsprechenden Funktionen überschreiben werden. Wie mit der eigenen Mutator-Klasse dann eine Mutation eines Protocol Buffers durchgeführt werden kann, ist in der `doOwnMutation()` Funktion veranschaulicht. Der ganze Ablauf wird besonders deutlich, wenn man `libfuzzer_mutator.h` und `libfuzzer_mutator.cc` im LPM Repository betrachtet. Dort werden die virtuellen Mutator-Funktionen durch die Mutatoren der libFuzzer Library überschrieben. [59] Durch gezielte Mutationen der einzelnen Felder kann die Wahrscheinlichkeit für die Generierung valider Inputs stark erhöht werden.

```
class MyProtobufMutator : public protobuf_mutator::Mutator {
  public:
  // redefine the Mutate* methods to perform more sophisticated mutations.
  protected:
  // Example for overriding the mutator for int32 fields
  int32_t MutateInt32(int32_t value) override;
  // testobject-specific implementation in related cpp file
}
  // Manual use of a selfdefined ProtobufMutator
void doOwnMutation(MyMessage* message) {
  MyProtobufMutator mutator;
  mutator.Seed(my_random_seed);
  mutator.Mutate(message, 200);
}
```

Listing 5.8: Aufbau und Nutzung eines eigenen Protobuf-Mutators

5.2.1 Aufbau eines Containers für Fuzzing mit Protocol Buffern

An dieser Stelle soll zunächst eine Schritt-für-Schritt-Anleitung gegeben werden, wie man einen Dockercontainer aufbauen kann, der alle wichtigen Tools und Abhängigkeiten enthält, die für Structure-Aware Fuzzing mit Protocol Buffern nötig sind. Es bietet sich an, den Dockercontainer auf dem AFL++ Dockerimage [61] aufzubauen, da dieses einen funktionsfähigen AFL++ Fuzzer inklusive aller Abhängigkeiten mitbringt. Da libFuzzer Teil der LLVM-Clang-Toolchain ist, steht auch dieser Fuzzer innerhalb des Images zur Verfügung. Beim Testen für diese Arbeit wurde jedoch explizit Clang-14 verwendet und dieser kann wie folgt installiert werden.

```
apt-get install clang-14 libfuzzer-14-dev
```

Nachdem das Image mit dem `docker pull` Kommando heruntergeladen wurde und ein Container mit `docker run` erstellt wurde, kann sich mit diesem verbunden werden. Alle Binärdateien von AFL++ sind an der Wurzel des Dateisystems im `AFLplusplus` Ordner platziert und können über diesen Pfad verwendet werden. Nun gilt es alle Abhängigkeiten der Protocol Buffer und des libprotobuf-mutators zu installieren.

```
apt-get update
apt-get install protobuf-compiler libprotobuf-dev binutils cmake \
    ninja-build liblzma-dev libz-dev pkg-config autoconf libtool
```

Höchstwahrscheinlich wird die vom Package-Manager bereitgestellte CMake Version unterhalb von Version 3.24 liegen. An dieser Stelle sollte man CMake mit `apt remove` deinstallieren. Die neuste CMake Version kann man am leichtesten über den Packageinstaller für Python (pip) wie folgt herunterladen.

```
pip install cmake
```

Die einzige Besonderheit ist hierbei, dass CMake anschließend unter dem Pfad `/usr/local/lib/python3.10/dist-packages` zu finden ist. Diesen muss man daher noch zu `$PATH` hinzufügen. Abschließend kann mit `cmake - -version` überprüft werden, ob die neuste Version von CMake nutzbar ist. Im nächsten Schritt muss das Repository des libprotobuf-mutators [59] geklont werden. Zur Zeit der Bearbeitung dieser Arbeit gab es innerhalb des libprotobuf-mutator Repositories Probleme mit Abhängigkeiten zur Abseil-C++-Library [62], die den Build-Prozess ggf. fehlschlagen ließen. Zusätzlich hat sich der manuelle Build-Prozess sämtlicher Libraries im Einzelnen als sehr umfangreich und kompliziert herausgestellt. Da der Fokus in dieser Arbeit auf dem Konzept von Structure-Aware Fuzzing liegt, wurde hierfür der Branch `af3bb18` [63] von libprotobuf-mutator gewählt, welcher sich in einem Zug gemeinsam mit allen Abhängigkeiten stabil bauen ließ. Innerhalb des geklonten libprotobuf-mutator Ordners muss dafür ein `build` Ordner erstellt werden. Innerhalb des `build` Ordners können die benötigten Libraries durch die folgenden drei Kommandos gebaut und installiert werden.

```
cmake .. -GNinja -DCMAKE_C_COMPILER=clang-14 \
-DCMAKE_CXX_COMPILER=clang++-14 \
-DCMAKE_BUILD_TYPE=Debug \
-DLIB_PROTO_MUTATOR_DOWNLOAD_PROTOBUF=ON \
-DCMAKE_C_FLAGS="-fPIC" -DCMAKE_CXX_FLAGS="-fPIC"

ninja

ninja install
```

Hierbei ist die Option `-DLIB_PROTO_MUTATOR_DOWNLOAD_PROTOBUF=ON` entscheidend, da hierdurch automatisch eine funktionierende Version von Protobuf mit allen Abhängigkeiten heruntergeladen und gebaut wird. Durch diese Option erspart man sich sämtliche Komplexität und Schwierigkeiten, die der Build-Prozess jeder Library im Einzelnen erzeugt. [59] Es sei darauf hingewiesen, dass der `Protoc` Compiler aus dem `libprotobuf-mutator/build/external.protobuf/bin/protoc` verwendet werden sollte, da auf diese Weise Inkompatibilitäten durch verschiedene Versionen vorgebeugt wird. Nach dem Build-Prozess ist alles für das Structure-Aware Fuzzing mit Protocol Buffern vorbereitet. Der libprotobuf-mutator ist in Form der Libraries `libprotobuf-mutator-libfuzzer.a` bzw. `libprotobuf-mutator.a` im `src` Verzeichnis des `build` Ordners verwendbar. Die Abhängigkeiten des Protocol Buffer Quellcodes sind unter `external.protobuf/lib/libprotobufd.a` bzw. `external.protobuf/include` im `build` Ordner verfügbar.

5.2.2 Fuzz-Test der `add_person()` Funktion mit und ohne LPM

Für eine genauere Untersuchung von Structure-Aware Fuzzing mit Protocol Buffern, wird ein Fuzz-Test der add_person() Funktion zuerst mit libFuzzer und LPM durchgeführt. Anschließend wird der gleiche Test ohne LPM durchgeführt und beide Techniken miteinander verglichen. An dem Aufbau des Fuzz-Tests mit LPM, über das `DEFINE_PROTO_FUZZER` Makro aus Listing 5.7, wird hierbei zunächst nichts verändert. Zur Demonstration wird ein Fehler in die `add_person()` Funktion nach Listing 5.9 eingebaut. Hierbei sei darauf hingewiesen, dass die if-Anweisungen explizit geschachtelt wurden, da der Fuzzer so mehr Schritte registrieren kann, als wenn alle Bedingungen in einer if-Anweisung stehen. Die Einführung zusätzlicher Schritte unter Beibehaltung der Programmlogik ist eine nützliche Technik, um dem Fuzzer mehr Informationen zur Coverage innerhalb eines Testobjektes zur Verfügung stellen zu können.

```
1 void add_person(string vorname,string nachname, uint32_t id, float groesse)
2 {
3     if (vorname == "Max") {
4         if (nachname == "Mustermann") {
5             if(id == 1) {
6                 if((groesse < 1.8) && (groesse > 1.6)) {
7                     abort();    //BUG!
8 }   }   }   }   }
```

Listing 5.9: Testobjekt add_person() mit eingebautem Fehler

Der Fuzz-Test wird ohne initialen Corpus für 10000 Ausführungen gestartet. Zusätzlich werden die generierten Inputs während der Ausführung in der Konsole ausgegeben. Hier kommt erneut der Vorteil der Protocol Buffer zum Vorschein, dass die Größe und Inhalt der Felder bekannt sind. Somit müssen die entsprechenden Informationen nicht aus einem Byte-Array extrahiert werden. In Abbildung 5.3 ist ein Teil der Konsolenausgabe dargestellt. Man erkennt am Ende der Ausgabe, dass der Fuzzer in 10000 Ausführungen bereits erkannt hat, dass „Max" und „Mustermann" zur Entdeckung neuer Zweige führen. Diese gehören mit 441 bzw. 54 zu den am häufigsten verwendeten Inputs. Dass der Fuzzer Informationen über besonders häufig verwendete Inputs ausgibt, ist normales Verhalten von libFuzzer und würde auch ohne den Einsatz von LPM erfolgen.

```
Vorname: ax; Nachname: ; Id: 0; Groesse: 0
Vorname: ax; Nachname: ; Id: 0; Groesse: 4.23304e-41
Vorname: ; Nachname: ; Id: 0; Groesse: 4.23304e-41
Vorname: ; Nachname: ; Id: 0; Groesse: 0
Vorname: ; Nachname: Mustermann; Id: 0; Groesse: 0
Vorname: ; Nachname: Mustermann; Id: 0; Groesse: 0
Vorname: ; Nachname: Mustermann; Id: 0; Groesse: 5.87747e-39
Vorname: ; Nachname: Mustevmann; Id: 0; Groesse: 5.87747e-39
Vorname: ; Nachname: Mustevmuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuann; Id: 0; Groesse: 5.87747e-39
Vorname: Max; Nachname: ; Id: 3014656; Groesse: 0
Vorname: Max; Nachname: Max; Id: 3014656; Groesse: 0
Vorname: Max; Nachname: Max; Id: 3014656; Groesse: 5.78564e-39
Vorname: Max; Nachname: Max; Id: 1; Groesse: 5.78564e-39
Vorname: Max; Nachname: Max; Id: 1; Groesse: 5.78564e-39
Vorname: Mustermann; Nachname: ; Id: 0; Groesse: 0
Vorname: ; Nachname: ; Id: 0; Groesse: 0
Vorname: ; Nachname: Max; Id: 0; Groesse: 0
Vorname: Max; Nachname: Max; Id: 0; Groesse: 0
Vorname: Mustermann; Nachname: Max; Id: 0; Groesse: 0
Vorname: Max; Nachname: ; Id: 1; Groesse: 0
Vorname: Ma; Nachname: ; Id: 1; Groesse: 0
#10000  DONE   cov: 94 ft: 117 corp: 6/266b lim: 4096 exec/s: 3333 rss: 31Mb
###### Recommended dictionary. ######
"Max" # Uses: 441
"Mustermann" # Uses: 54
"\001\000\000\000" # Uses: 74
###### End of recommended dictionary. ######
Done 10000 runs in 3 second(s)
```

Abbildung 5.3: Ausgabe des Fuzz-Tests von `add_person()` für 10000 Ausführungen mit LPM

Anhand der Ausgabe der generierten Inputs lässt sich das Mutationsverhalten von LPM genauer untersuchen. Wenn man die im oberen Bereich der Abbildung ausgegebenen Felder betrachtet, dann erkennt man, dass die Felder durch LPM bei der Mutation getrennt behandelt werden. Man kann sehen, dass der Fuzzer bei seiner Arbeit gezielt Mutationen einzelner Felder durchführt. So wird „Max" zu „ax" oder „Mustermann" komplett entstellt und in die Länge gezogen. Darüber hinaus nimmt der Fuzzer interessante Inputs einzelner Felder und setzt diese auch in andere Felder ein. Daher taucht „Max" beispielsweise auch manchmal als Nachname auf. Interessante Felder werden insgesamt häufiger verwendet, gelegentlich lässt der Fuzzer mit LPM einzelne Felder frei. Im späteren Vergleich wird sich zeigen, dass libFuzzer ein derartiges Verhalten beim Mutieren ohne LPM und Protocol Buffer nicht zeigen würde. Stattdessen würde der Fuzzer die Inputs als eine riesige Bytesequenz betrachten und dementsprechend mutieren. Durch die getrennte Mutation einzelner Felder eignet sich Structure-Aware Fuzzing mit Protocol Buffern im Gegensatz zum reinen Coverage-Guided Fuzzing, besonders gut zum Testen von Funktionen mit vielen und vor allem zusammengesetzten Inputs.
Bisher war der Fuzz-Test mit 10000 Ausführungen noch nicht in der Lage den Fehler zu erreichen. Daher wurde der Fuzz-Test so lange durchgeführt, bis alle passenden Inputs und damit der Fehler gefunden wurde. Hierfür wurden die Konsolenausgaben der Inpus deaktiviert, da dadurch die Anzahl der Testfälle pro Sekunde von knapp über 3000 auf durchschnittlich etwa 22000 anstieg. Hierbei wurde der Fehler nach 6 Minuten und 52 Sekunden gefunden. Erinnert man sich an das theoretische `https://` Beispiel mit zufällig generierten Bytes (Kapitel 3.3), dann stellt diese Zeit einen sehr guten Wert dar. Zumal hierbei ebenfalls kein initialer valider Input bereitgestellt wurde und mehrere Inputs den richtigen Wert annehmen müssen.
Im Folgenden soll nun ein Vergleich mit libFuzzer ohne LPM durchgeführt werden. Vorab sei darauf hingewiesen, dass der Vergleich von Fuzzern nur schwer durchzuführen ist und oft wenig Aussagekraft hat. Dies liegt daran, dass es ein zufallsbasiertes Verfahren ist und auch andere Faktoren wie das betrachtete Testobjekt und die unterschiedliche Zahl an Ausführungen pro Sekunde den Vergleich erschweren. Darüber hinaus ist zu beachten, dass die betrachtete `add_person()` Funktion ein sehr einfaches Testobjekt ist, das für reine coverage guided Fuzzer ebenfalls kein Problem darstellt. Für den Vergleich wird zunächst die Zahl der benötigten Testfälle des Durchlaufs mit LPM abgeschätzt. Der Fuzzer gibt keine exakte Aussage über die Zahl der insgesamt durchgeführten Testfälle. Während der Ausführung erhält man jedoch regelmäßig Rückmeldung über die Zahl der bisher ausgeführten Testfälle. Im Durchlauf mit LPM war die letzte Rückmeldung bei ca. 8,39 Mio. ausgeführter Testfälle, wobei kurz danach der Fehler entdeckt wurde. Aus der gemessenen Zeit bis zur Entdeckung des Fehlers und den durchschnittlich pro Sekunde durchgeführten Testfällen ergibt sich die ungefähre Anzahl ausgeführter Testfälle wie folgt. Diese ist wichtig für eine spätere Vergleichbarkeit.

$$T_{gesamt} \approx (6\,min \cdot 60\,\frac{s}{min} + 52\,s) \cdot 22000\,\frac{exec}{s} \approx 9{,}06 \cdot 10^6\,exec \tag{5.1}$$

Nun wird der gleiche Fuzz-Test mit libFuzzer ohne LPM durchgeführt. Dabei wird der Aufruf des Fuzz-Tests nicht mehr mit dem `DEFINE_PROTO_FUZZER` Makro ausgeführt und die Inputs der Funktion müssen aus dem generierten Byte-Array extrahiert werden. Der entsprechende Code ist in Listing 5.10 dargestellt. Bereits hier wird deutlich, dass ohne den Einsatz der Protocol Buffer ein wesentlich größerer Implementierungsaufwand betrieben werden muss, um die Inputs aus dem Byte-Array zu extrahieren. Darüber hinaus muss strikt überprüft werden, ob der Fuzzer überhaupt genug Bytes generiert, um die entsprechenden Inputs zu füllen. Tut man dies nicht, könnten Sanitizer hier bereits einen Out-of-Bounds Speicherzugriff feststellen. Stellt man sich vor, dass man ein Testobjekt mit einer größeren Anzahl an Inputs bzw. mit zusammengesetzten Inputs hat, dann wird der Arbeitsaufwand schnell sehr groß. Darüber hinaus ist die dargestellte Implementierung nicht wirklich

realisitisch, da die Größe der Strings exakt auf den gesuchten Vor- und Nachnamen ausgelegt ist. In der Realität würde man wollen, dass verschiedengroße Strings verwendet werden. Datentypen mit dynamischer Größe sind ohne die Protocol Buffer schwer zu handhaben, doch dazu später mehr.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    std::string vorname;     // 3 Byte  "Max"
    std::string nachname;    // 10 Byte "Mustermann"
    uint32_t id;             // 4 Byte
    float groesse;           // 4 Byte

    if (Size >= 3) {
        vorname = std::string(reinterpret_cast<const char*>(Data), 3);
    }
    if (Size >= 10 + 3) {
        nachname = std::string(reinterpret_cast<const char*>(Data + 3),10);
    }
    if (Size >= 4 + 10 + 3) {
        std::memcpy(&id, Data + 10 + 3, 4);
    }
    if (Size >= 4 + 4 + 10 + 3) {
        std::memcpy(&groesse, Data + 4 + 10 + 3, 4);
    }
    add_person(vorname, nachname, id, groesse);
    return 0;
}
```

Listing 5.10: Fuzz-Test mit libFuzzer ohne LPM

Zunächst wird der Fuzz-Test ohne LPM ebenfalls mit 10000 Ausführungen durchgeführt und dabei die generierten Inputs über die Konsole ausgegeben. Die entsprechende Konsolenausgabe ist in Abbildung 5.4 dargestellt.

```
Vorname: Aa; Nachname: Ma; Id: 22092; Groesse: 5.52801e+19
Vorname: Aa; Nachname: Maxa; Id: 22092; Groesse: 5.52801e+19
Vorname: Ma; Nachname: xM
; Id: 4048386289; Groesse: 1.34222e+08
Vorname: Ma; Nachname: xM; Id: 22092; Groesse: 5.52801e+19
Vorname: MaM; Nachname: axxM; Id: 22092; Groesse: 5.52801e+19
Vorname: aM; Nachname: axxM; Id: 22092; Groesse: 5.52801e+19
Vorname: aM; Nachname: axeMaxM; Id: 1632436224; Groesse: 5.52801e+19
Vorname: Max; Nachname: M
; Id: 4059123064; Groesse: 2.36367e+20
Vorname: Maw; Nachname: M
; Id: 4059123064; Groesse: 2.36367e+20
Vorname: Maw; Nachname: MX; Id: 403616391; Groesse: 2.36367e+20
Vorname: Maw; Nachname: MX; Id: 403616391; Groesse: 4.55226e-14
Vorname: M; Nachname: X ↑; Id: 692913152; Groesse: 5.52801e+19
#10000  DONE   cov: 24 ft: 26 corp: 6/58b lim: 98 exec/s: 0 rss: 28Mb
###### Recommended dictionary. ######
"Max" # Uses: 449
"eMa" # Uses: 407
###### End of recommended dictionary. ######
Done 10000 runs in 0 second(s)
```

Abbildung 5.4: Ausgabe des Fuzz-Tests von `add_person()` für 10000 Ausführungen ohne LPM

Im Vergleich zur Durchführung mit LPM ist zu erkennen, dass der Fuzzer hier zwar „Max" als interessanten Input ausgemacht hat, „Mustermann" hat dierser jedoch noch nicht vermehrt verwendet.

Der häufig genutzte „eMa" Input ist ein reines Zufallsprodukt und lässt sich so auch bei Wiederholung nicht rekonstruieren. Besonders interessant sind die erkennbaren Unterschiede im Mutationsverhalten. Es zeigt sich, dass der Fuzzer ohne LPM die Felder nicht getrennt voneinander behandelt. Stattdessen basieren die Mutationen auf der Betrachtung der generierten Bytesequenz als eine große Einheit. So taucht die interessante Sequenz für `Max` im Nachnamen eher willkürlich innerhalb der Bytes des Nachnamens auf. Dies resultiert daraus, dass der Fuzzer ohne LPM die interessante Sequenz zwar wiederverwendet aber Byte-Swaps auf dem gesamten generierten Byte-Array durchführt. Darüber hinaus zeigen die zahlreichen nicht anzeigbaren Zeichen, dass die Bytes wesentlich willkürlicher mutiert werden. An dieser Stelle sei erneut auf die Problematik mit Datentypen dynamischer Länge hingewiesen. Es ist unter der gegebenen Implementierung gar nicht möglich, einen String für den Nachnamen zu erzeugen, der kleiner als 10 Byte ist. Der Fuzz-Test mit LPM hat diese Einschränkung nicht und kann sowohl längere als auch kürzere Strings erzeugen.
Um den Vergleich fortführen zu können, wurde die Konsolenausgabe deaktiviert und der Fuzz-Test ohne LPM bis zum Erreichen des Fehlers fortgesetzt. Hierbei zeigte sich, dass libFuzzer mit seinem standardmäßigen Mutator eine wesentlich schnellere Ausführungsgeschwindigkeit hat und durchschnittlich ca. 800.000 Ausführungen pro Sekunde erreicht. Der Fehler wurde auf diese Weise nach 3 Minuten und 29 Sekunden entdeckt. Die letzte Rückmeldung über die Zahl der durchgeführten Testfälle erfolgte bei 134 Mio. Ausführungen. Nun wird die ungefähre Anzahl der insgesamt durchgeführten Testfälle nach dem gleichen Schema abgeschätzt.

$$T_{gesamt} \approx (3\,min \cdot 60\,\frac{s}{min} + 29\,s) \cdot 800000\,\frac{exec}{s} \approx 167{,}2 \cdot 10^6\,exec \tag{5.2}$$

Es zeigt sich, dass libFuzzer ohne LPM zeitmäßig mit 3 Minuten und 29 Sekunden, den Fehler in ca. der Hälfte der Zeit des Tests mit LPM gefunden hat. Dies resultiert ausschließlich in der wesentlich schnelleren Ausführungsgeschwindigkeit. Betrachtet man die berechnete Anzahl der insgesamt ausgeführten Testfälle, dann hat der Fuzz-Test mit LPM nur ca. 9,06 Mio. Testfälle bis zur Entdeckung des Fehlers benötigt, während der Test ohne LPM ca. 167,2 Mio. Testfälle benötigt hat.
An dieser Stelle kann keine Aussage getroffen werden, welche die bessere Technik ist. Man kann lediglich feststellen, dass libFuzzer ohne LPM den schnelleren Mutator hat und wesentlich mehr Testfälle pro Zeiteinheit generiert. Demgegenüber sorgt LPM über eine gezieltere Mutation einer gegebenen Struktur dafür, dass der Fehler zwar zeitlich langsamer aber mit wesentlich weniger Testfällen erreicht wird. Grundsätzlich ist aber der Implementierungsaufwand des Fuzz-Tests unter Einsatz von LPM und Protocol Buffern geringer. Ohne Protocol Buffer ist die Extraktion der generierten Bytes sehr aufwendig und es entstehen Probleme mit Datentypen dynamischer Länge. Hierbei gilt es zu bedenken, dass es sich bei der `add_person()` Funktion um ein sehr einfaches Testobjekt handelt. Bei Testobjekten mit einer größeren Anzahl bzw. zusammengesetzten Inputs, treten die Vorteile des Einsatzes von LPM und Protocol Buffern stärker in den Vordergrund.
Abschließend soll das Problem der Datentypen mit dynamischer Länge näher beleuchtet werden. Wie bereits erwähnt, ist die Implementierung des Fuzz-Tests ohne LPM aus Listing 5.10 nicht optimal bzw. realistisch. Hierbei wurde die Länge der Strings explizit so festgelegt, dass diese die richtige Länge der gesuchten Worte haben. Es gilt dabei zu bedenken, dass jeder generierte String mit unpassender Länge einen invaliden Input darstellt. Das bedeutet, dass der zuvor durchgeführte Test viel leichter zum Fehler gelangen konnte, da die Strings immer die richtige Länge hatten. Im Prinzip wird dadurch viel Willkür aus dem Fuzz-Test herausgenommen und diese ist eigentlich gewünscht, da viele Strings unterschiedlicher Länge generiert werden sollen. Passt man den Fuzz-Test ohne LPM nun so an, dass dieser Strings unterschiedlicher Länge generiert, dann wird die Implementierung weiter verkompliziert. Derartige Festlegungen variabler Längen sind kein leichtes Unterfangen und stellen

den Tester vor Probleme, da dies erhebliche Auswirkungen auf den Fuzz-Test hat. Als Beispiel könnte man entsprechend Listing 5.11 die Stringlänge unter Einsatz von Zufallszahlen in einem definierten Bereich festlegen.

```
std::srand(std::time(nullptr));   // #include <ctime>
uint8_t sizeVN = rand() %  4 + 1; // Laenge 1-4  Byte
uint8_t sizeNN = rand() % 11 + 1; // Laenge 1-11 Byte

if (Size >= sizeVN) {
    vorname = std::string(reinterpret_cast<const char*>(Data), sizeVN);
}
if (Size >= sizeNN+sizeVN) {
    nachname = std::string(
                reinterpret_cast<const char*>(Data + sizeVN), sizeNN);
}
if (Size >= 4 + sizeNN + sizeVN) {
    std::memcpy(&id, Data + sizeNN + sizeVN, 4);
}
if (Size >= 4 + 4 + sizeNN + sizeVN) {
    std::memcpy(&groesse, Data + 4 + sizeNN + sizeVN, 4);
}
```

Listing 5.11: Anpassung Fuzz-Test ohne LPM mit variablen Stringlängen

Der Fuzz-Test wurde entsprechend des Listings wiederholt, sodass der Vorname nun zufällig zwischen 1-4 Byte und der Nachname zwischen 1-11 Byte lang ist. Die dargestellte Testimplementierung ist erneut nicht wirklich optimal, da hier der Längenbereich weiterhin stark eingegrenzt ist. Diese starke Eingrenzung ist jedoch nötig, da die Leistungsfähigkeit des Fuzz-Tests ohne LPM dadurch bereits signifikant abnimmt. So sinkt die Zahl der Ausführungen pro Sekunde allein durch die Generierung der Zufallszahlen von durchschnittlich ca. 800.000 auf durchschnittlich ca. 350.000 ab. Die Zeit bis der Fuzz-Test auf diese Weise den Fehler erreicht steigt auf 56 Minuten und 56 Sekunden. Die letzte Rückmeldung über die Zahl der durchgeführten Testfälle erfolgte bei 1,07 Mrd. Ausführungen. Die Abschätzung der ungefähren Zahl der insgesamt durchgeführten Testfälle lautet demnach:

$$T_{gesamt} \approx (56\,min \cdot 60\,\frac{s}{min} + 56\,s) \cdot 350000\,\frac{exec}{s} \approx 1{,}2 \cdot 10^9\,exec \tag{5.3}$$

Es wird deutlich, wieso die Länge der Strings weiterhin so stark eingegrenzt wurde. Mit jeder Vergrößerung dieses Bereichs sinkt statistisch die Chance auf Strings mit der richtigen Länge. Würde der Längenbereich weiter vergrößert werden, dann wäre der Fuzz-Test irgendwann nicht mehr in der Lage, den Fehler in einer angemessenen Zeit zu erreichen. Generell macht dieses Beispiel deutlich, wie groß die Auswirkungen von kleinen Anpassungen in der Testimplementierung auf die Ausführungsgeschwindigkeit und die Chance zur Generierung valider Inputs sein können. Dies gilt für jede Art von Fuzz-Test und bezieht sich auch auf die Implementierung eigener Mutatoren. Vergleicht man das Ergebnis erneut mit dem Fuzz-Test mit LPM, dann hat dieser zwar eine wesentlich langsamere Ausführungsgeschwindigkeit, gleichzeitig ist die gezielte Mutation einzelner Felder sehr effektiv. Darüber hinaus ist die Implementierung durch den Einsatz der Protocol Buffer wesentlich leichter und es entstehen keine Probleme bei der Handhabung von Datentypen mit dynamischer Länge. Diese Vorteile nehmen zu, je größer und komplexer die Inputs des Testobjektes werden.

5.2.3 Corpus- und Crashmanagement mit Protocol Buffern

Der Einsatz von Protocol Buffern hat zusätzlich erhebliche Vorteile bei der Definition eines eigenen Corpuses und beim Aufbau der generierten Crashreports. Grundsätzlich kann man zum Aufbau eines eigenen Corpuses einfach einen Ordner erstellen und dem Fuzz-Test diesen Ordner als Corpus-Ordner mit übergeben. Innerhalb des Ordners können beliebig Dateien platziert werden, die der Binärrepräsentation von validen Inputs entsprechen. Diese können einfach in Dateien mit beliebigem Namen ohne Dateiendung geschrieben werden. Für libFuzzer sieht dies bei der Ausführung wie folgt aus. Hierbei wird ebenfalls jede während des Fuzz-Tests stattfindende Corpuserweiterung in diesem Ordner gespeichert.

```
./my_fuzztest CORPUS_DIR
```

Ohne den Einsatz der Protocol Buffer hat es sich als schwierig herausgestellt, vernünftige Binärrepräsentationen für valide Inputs verschiedener Datentypen manuell zu erstellen. Unter Einsatz von Protocol Buffern und LPM können diese entsprechend der definierten Datenstrukturen erstellt werden. So kann man für die zuvor verwendete `person_data` Datenstruktur (Listing 5.6) des Fuzz-Tests eine leere Datei erstellen und die Inputs nach den Namen der Felder wie in Listing 5.12 eintragen. In diesem Fall würde der Fuzz-Test mit LPM den Fehler sofort entdecken.

```
1 vorname: "Max"
2 nachname: "Mustermann"
3 id: 1
4 groesse: 1.71
```

Listing 5.12: Strukturierte Corpusdefinition mit Protocol Buffern

Ähnliches gilt für den Crashreport. Ohne den Einsatz der Protocol Buffer und LPM ist die generierte Datei mit dem Crashreport eine Binärrepräsentation des beim Testfall mit dem Crash generierten Byte-Arrays. Ein derartiger Crashreport des Fuzz-Tests ohne LPM ist in Abbildung 5.5 dargestellt. Man sieht, dass es für den Tester nicht unmittelbar ersichtlich ist, welche Inputs zum Crash geführt haben. Das manuelle Auslesen des Reports mit einem Hexeditor wird zusätzlich erschwert, da man die Größe bzw. Trennung einzelner Datentypen manuell rekonstruieren muss.

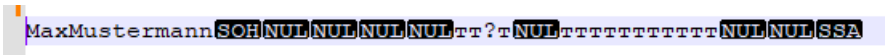

Abbildung 5.5: Crashreport des Fuzz-Tests ohne LPM in Notepad++

Bei Verwendung von Protocol Buffern und LPM sieht der generierte Crashreport exakt so aus, wie die in Listing 5.12 dargestellte Erstellung valider Inputs für den Corpus. Eine Abbildung des Crashreports vom Fuzz-Test mit LPM ist in Abbildung 5.6 dargestellt.

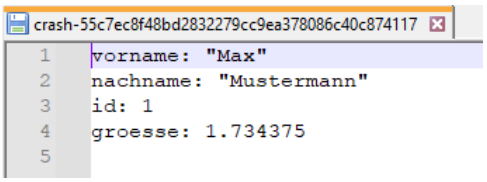

Abbildung 5.6: Crashreport des Fuzz-Tests mit LPM in Notepad++

Insgesamt stellen diese Verbesserungen der Corpusgenerierung und der Crashreports einen enormen Vorteil der Fuzz-Tests mit Protocol Buffern und LPM dar. Stellt man sich ein Testobjekt vor, das sehr viele komplexe/zusammengesetzte Inputs erwartet, dann lässt sich so ein Fuzz-Test wesentlich leichter durchführen und auswerten. Auf diese Weise wird dem Tester viel Arbeitsaufwand erspart.

5.2.4 Mutation Post-Processing

Häufig ist es für die Generierung valider Inputs erforderlich, dass einzelne Felder/Variablen nur bestimmte Werte annehmen dürfen. Derartige Beschränkungen haben enormen Einfluss auf die Effektivität des Fuzz-Tests. Dadurch werden viele der generierten Inputs invalide, auch wenn der Fuzzer über die Zeit in der Lage ist, passende Werte für diese Felder zu generieren. Zur Lösung dieses Problems bietet LPM das sog. `Mutation Post-Processing` an. Dahinter steht die Registrierung einer oder mehrerer Callback-Funktionen, sodass ein Protocol Buffer nach der Mutation weiter modifiziert werden kann. Auf diese Weise ist es möglich einzelne Protocol Buffer nach der Mutation gezielt anzupassen und Felder so mit festgelegten Werten zu mutieren. [59] Die Registrierung einer derartigen Callback-Funktion ist in Listing 5.13 dargestellt.

```
1 static protobuf_mutator::libfuzzer::PostProcessorRegistration<MyMessageType
    > reg = {
2     [](MyMessageType* message, unsigned int seed) {
3       TweakMyMessage(message, seed);
4     }};
5
6 DEFINE_PROTO_FUZZER(const MyMessageType& input) {
7   testobject(input);
8 }
```

Listing 5.13: Implementierung von Mutation Post-Processing mit LPM

Nach dem dargestellten Schema können mehrere Callback-Funktionen registriert werden. Die nachträgliche Manipulation des bereits mutierten Protocol Buffers müsste in der `TweakMyMessage()` Funktion implementiert werden. Damit man sich die Funktionsweise vorstellen kann, wird in Listing 5.14 das `Mutation Post-Processing` für den Fuzz-Test mit LPM implementiert. Hierbei wird der Seed verwendet, um den Vornamen ausschließlich die Worte `Hello` und `World` annehmen zu lassen.

```
1 static protobuf_mutator::libfuzzer::PostProcessorRegistration<person_data>
    reg = {
2         [](person_data* message, unsigned int seed) {
3             if (seed % 2) {
4                 message->set_vorname("Hello");
5             }
6             else {
7                 message->set_vorname("World");
8     }   }   };
```

Listing 5.14: Mutation Post-Processing am Beispiel des Vornamens

Die Implementierung des Fuzz-Tests mit LPM durch das `DEFINE_PROTO_Fuzzer` Makro (Listing 5.7) bleibt hierbei unverändert. Anschließend wird die Konsolenausgabe aktiviert und der Test erneut ausgeführt. Ein Ausschnitt der Konsolenausgabe ist in Abbildung 5.7 dargestellt. Man erkennt, dass der Vorname ausschließlich die entsprechenden Worte annimmt. Durch die Anwendung von `Mutation Post-Processing` können einzelne Felder in gezielter Art und Weise mutiert werden. Je nach Testobjekt kann so die Chance zur Generierung valider Inputs signifikant erhöht werden.

Gleichzeitig ist die Implementierung sehr leicht und einzelne Callback-Funktionen lassen sich schnell deaktivieren oder gegeneinander austauschen. So kann der Fuzzer gezielt an die gewünschten Stellen innerhalb eines Testobjektes geführt werden.

```
Vorname: Hello; Nachname: ; Id: 0; Groesse: 0
Vorname: Hello; Nachname: ; Id: 0; Groesse: 0
Vorname: World; Nachname: ; Id: 0; Groesse: 0
Vorname: World; Nachname: Hello; Id: 0; Groesse: 0
Vorname: World; Nachname: ; Id: 0; Groesse: 2.1715e-18
Vorname: Hello; Nachname: ; Id: 0; Groesse: 2.1715e-18
#10000  DONE   cov: 109 ft: 130 corp: 2/48b lim: 4096 exec/s: 3333 rss: 31Mb
Done 10000 runs in 3 second(s)
```

Abbildung 5.7: Konsolenausgabe des Fuzz-Tests mit Mutation Post-Processing des Vornamens

5.3 Bewertung von Structure-Aware Fuzzing mit Protocol Buffern

Es hat sich gezeigt, dass der Einsatz von Structure-Aware Fuzzing mit Protocol Buffern viele Vorteile gegenüber reinem Coverage-Guided Fuzzing hat. Zumal das Feedback zur Codeabdeckung erhalten bleibt. Der größte Vorteil liegt in der einfachen Implementierung, da die Daten des definierten Protocol Buffers ohne großen Aufwand auf die Inputs des Testobjektes verteilt werden können. Hierbei ist es leicht möglich eine große Anzahl von Inputs, oder auch zusammengesetzte Inputs nachzubilden. Die aufwendige Extraktion der benötigten Daten aus einem generierten Byte-Array entfällt bei dieser Technik. Auf diese Weise lassen sich Fuzz-Tests für Testobjekte mit komplizierten Inputs schnell und einfach implementieren. Besonders vorteilhaft ist es, wenn Protocol Buffer unabhängig von Fuzz-Tests bereits innerhalb einer Codebasis als Datenstrukturen eingesetzt werden. Darüber hinaus lassen sich Datentypen mit dynamischer Länge über den `libprotobuf-mutator` leicht handhaben. Hier hat der Vergleich der Fuzz-Tests verdeutlicht, wie groß die Probleme bei der Handhabung von Datentypen mit dynamischer Länge ohne die Verwendung von LPM sind. Es stellte sich heraus, dass LPM die Ausführungen pro Sekunde stark reduziert, dabei jedoch wesentlich weniger Testfälle benötigt, um den Fehler in der betrachteten `add_person()` Funktion zu finden. Als der Fuzz-Test ohne LPM auf die Generierung von Inputs mit dynamischer Länge erweitert wurde, wobei dabei der Längenbereich weiterhin stark eingeschränkt war, konnte dieser den Fehler nur noch schwer erreichen. Wäre der Längenbereich weiter vergrößert worden, dann wäre der Fuzz-Test nicht mehr in der Lage gewesen, den Fehler in einer angemessenen Zeit zu entdecken. Dies und der Vergleich des Mutationsverhaltens beider Tests verdeutlichen, wie effektiv gezielte Mutationen einzelner Felder sind. Zumal dabei noch keine testobjektspezifische Anpassung der Mutatoren vorgenommen wurde. Diesbezüglich bietet LPM den Vorteil, dass hier Mutatoren einzelner Feldtypen überschreiben werden können. So lassen sich leichter unterschiedliche testobjektspezifische Mutatoren für einzelne Felder implementieren. Zusätzlich ermöglicht der Einsatz von `Mutation Post-Processing` die einfache Nachbearbeitung der Felder. Dadurch können die Felder mit wenig Aufwand so mutiert werden, dass diese ausschließlich festgelegte Werte annehmen. Insgesamt kann so die Chance auf die Generierung von validen Inputs stark erhöht werden und der Fuzzer lässt sich leicht an gewünschte Stellen innerhalb eines Testobjektes lenken. Auch beim Corpusmanagement und der Generierung von Crashreports entstehen beim Einsatz der Protocol Buffer und LPM Vorteile. So können dem Fuzzer valide Inputs im Corpus entsprechend der Definition der Protocol Buffer bereitgestellt werden. Dadurch entfällt

der aufwendige Zusammenbau von Binärdaten valider Inputs, bei denen durch den Aufbau als Byte-Array, keine klare Trennung einzelner Datentypen existiert. Gleiches gilt für generierte Crashreports. Dort werden die Inputs ebenfalls lesbar, in getrennten Feldern, entsprechend der Definition der Protocol Buffer aufgeführt. Auf diese Weise entfällt die aufwendige Rekonstruktion der Inputdatentypen aus den Binärdaten des Crashreports. So wird viel Arbeitsaufwand gespart, da ein Corpus leicht aufgebaut werden kann und der Fuzz-Test einfach auszuwerten ist. Insgesamt nehmen alle genannten Vorteile von Structure-Aware Fuzzing mit Protocol Buffern zu, je komplexer und umfangreicher die Struktur des Inputs des Testobjektes ist. Bei reinem Coverage-Guided Fuzzing würde man bezüglich der Effektivität und des Implementierungsaufwands irgendwann an die Grenze der Machbarkeit stoßen.
Alles in allem kann man an dieser Stelle keine Aussage treffen, welche der Techniken die eindeutig bessere ist. Die Antwort auf diese Frage ist letztendlich immer vom Testobjekt abhängig. Grundsätzlich würde ich Structure-Aware Fuzzing mit Protocol Buffern aus Sicht der kommerziellen Softwareentwicklung im Bereich der Unit-Tests immer bevorzugen. In diesem Bereich werden überwiegend Testobjekte betrachtet, deren Inputs sich aus verschiedenen Standarddatentypen, in mehr oder weniger komplexer/verschachtelter Form zusammensetzen. Diese können über Protocol Buffer leicht nachgebildet werden und die anschließende Implementierung der Fuzz-Tests ist einfach und intuitiv. Auf diese Weise wird Softwareentwicklern ohne große Erfahrung im Bereich von Fuzz-Tests, ein einfacher Einstieg in diese Testtechnik ermöglicht. Da Fuzz-Tests keine anderen Testverfahren im Bereich der Softwaretests vollständig ersetzen können, bedeutet ihr Einsatz immer den Aufwand zusätzlicher Ressourcen. Hierbei ist die schnelle und einfache Implementierung das ausschlaggebende Argument für diese Technik. Nachdem die Struktur in Form eines Protocol Buffers abgebildet ist, läuft die Implementierung immer nach dem gleichen Schema ab und entspricht ausschließlich der Verteilung des Inhalts der definierten Felder auf die Inputs des Testobjektes. Aufgrund dessen ist das Verhältnis von Kosten und Nutzen bei dieser Technik sehr gut, sodass es in jedem Fall Sinn macht, sie an geeigneten Stellen einzusetzen. Insgesamt kann der Aufbau eines dertigen Fuzz-Tests mit den folgenden vier Schritten zusammengefasst werden.

- 1. Inputstruktur des Testobjekts als Protocol Buffer nachbilden und kompilieren
- 2. Implementierung des Fuzz-Tests bzw. Aufteilung der Felder auf Inputs des Testobjektes
- (3.) ggf. valide Inputs für Corpus erstellen und `Mutation Post-Processing` einbauen
- 4. Fuzz-Test ausführen

Der Fuzz-Test ist auf diese Weise nicht vollständig automatisiert aber der Implementierungsaufwand ist auf ein Minimum reduziert. Wird diese Technik standardmäßig im Testprozess eingesetzt, dann sollte die grundlegende Verwendung von Protocol Buffern, zur Definition von komplexen Datenstrukturen in der Codebasis, in Erwägung gezogen werden. Auf diese Weise wären die Definitionen der Strukturen vieler Inputs bereits vor der Implementierung des Fuzz-Tests vorhanden. Die Fuzz-Tests sollten an geeigneten Stellen manuell implementiert werden. In einem nächsten Schritt macht es Sinn, die Ausführung der implementierten Fuzz-Tests in die CI-CD Pipeline aufzunehmen. Die Idee dahinter ist, dass vor jedem Merge die definierten Fuzz-Tests innerhalb des Codes, für eine festgelegte Zeit ausgeführt werden. Der Corpus kann zwischen den Ausführungen der Tests gesammelt werden, sodass der Test bei jedem Merge in der Lage ist dort weiterzumachen, wo er beim letzten Merge aufgehört hat. So steigt die Zahl der insgesamt ausgeführten Testfälle mit jedem weiteren Merge sukzessive. Der Fuzzer ist in der Lage, jedes Mal weiter innerhalb des Testobjektes vorzudringen und neue Fehler zu entdecken.

6 Zusammenfassung und Ausblick

Im Zuge dieser Arbeit wurden Fuzz-Tests zunächst vor dem Hintergrund der Theorie zum Testen von Software betrachtet. Dabei wurde festgestellt, dass sich Fuzzing stark von konventionellen Techniken zum Testen von Software unterscheidet. Der Grund hierfür ist die Willkür dieses Verfahrens, welche aus den zufälligen Mutationen bzw. der zufälligen Testfallgenerierung resultiert. Diese Willkür sorgt dafür, dass sich der exakte Ablauf eines Fuzz-Tests nicht reproduzieren lässt. Im Gegensatz zu konventionellen Testtechniken sind Fuzz-Tests daher nicht geeignet, um die Erfüllung einzelner Anforderungen gezielt und reproduzierbar nachzuweisen. Andere Techniken innerhalb des Testprozesses von Software können somit nicht durch Fuzz-Tests ersetzt werden. Stattdessen kann Fuzzing ausschließlich als Ergänzung zu konventionellen Techniken eingesetzt werden, was zwingend die Aufwendung zusätzlicher Ressourcen erfordert.
In dieser vermeintlich als Nachteil wirkenden Willkür liegt jedoch die Stärke des Verfahrens. Die nach dem Zufallsprinzip generierten Testfälle unterscheiden sich stark von der Art der Testfälle, die ein Entwickler manuell implementieren würde. Häufig führt dies zur Generierung von Testfällen, die ein Mensch gar nicht bedenken würde. Gleichzeitig ist es ein automatisiertes Verfahren, das eine extrem große Anzahl verschiedener Testfälle generiert. Im Gegensatz zu konventionellen Testtechniken werden so mit jeder Durchführung des Tests neue einzigartige Testfälle generiert, statt immer die gleichen, fest implementierten Testfälle abzuprüfen. Durch die große Anzahl der Testfälle eignet sich das Verfahren ideal für die Kombination mit Sanitizern. Dadurch lassen sich viele verschiedene Fehlerklassen entdecken, die ebenfalls nur schwer mit konventionellen Techniken erkannt werden können. Aufgrund dessen birgt der Einsatz von Fuzzing als ergänzende Technik ein großes Potenzial, um die Softwarequalität signifikant zu steigern.
Im Anschluss an diese theoretischen Betrachtungen wurde ein Überblick über die verschiedenen Arten von Fuzz-Tests gegeben. Hierbei wurden besonders die Sachverhalte beleuchtet, die häufig in der Literatur nicht erklärt, oder als gegebenes Wissen angenommen werden. So wurde beispielsweise die Funktionsweise des Feedbacks zur Codeabdeckung beim Coverage-Guided Fuzzing erklärt. Darüber hinaus wurde gezeigt wie Fuzz-Tests klassifiziert werden können und wie einzelne Techniken im Detail funktionieren. Diesbezüglich schließen sich einzelne Fuzz-Techniken nicht gegenseitig aus, sondern sind als Erweiterung zueinander zu verstehen. So kann Coverage-Guided Fuzzing durch Einbezug von Informationen zur Struktur der Inputs auf Structure-Aware Fuzzing erweitert werden. In ähnlicher Art und Weise wurde in einem nächsten Schritt die Funktionsweise von gängigen Sanitizern im Detail erklärt. Durch das so geschaffene Verständnis zur Funktionsweise der einzelnen Werkzeuge und Komponenten eines Fuzz-Tests wird dem Einsteiger der Übergang zu einem erfahrenen Anwender/Entwickler ermöglicht.
Abschließend wurde Structure-Aware Fuzzing mit Protocol Buffern als vielversprechende Technik vorgeführt und untersucht. Dabei wurde im Detail erklärt, was Protocol Buffer sind, wie sie eingesetzt werden können und wie sich diese mit Fuzz-Tests kombinieren lassen. Im Zuge dessen wurde ein Fuzz-Test mit libFuzzer (reines Coverage-Guided Fuzzing) und mit libFuzzer in Kombination mit Protocol Buffern (Structure-Aware Fuzzing) zum Vergleich durchgeführt. Es hat sich gezeigt, dass Structure-Aware Fuzzing mit Protocol Buffern leichter zu implementieren ist und somit auch unerfahrenen Anwendern den einfachen Zugang ermöglicht. Dabei ist diese Technik besonders gut

für Testobjekte mit vielen, komplexen bzw. zusammengesetzten Inputs geeignet. Dies macht sie zur idealen Technik für den Einsatz in Unit-Tests. Darüber hinaus hat sie keine Probleme mit Inputs dynamischer Länge. Alles in allem hat Structure-Aware Fuzzing großes Potenzial als fester Bestandteil im Testprozess der kommerziellen Softwareentwicklung eingesetzt zu werden. Die einfach Implementierung sorgt dafür, dass der zusätzliche Kostenaufwand auf ein Minimum reduziert wird. Im Zuge dessen sollte auch der grundsätzliche Einsatz von Protocol Buffern, für die Definition von Datenstrukturen innerhalb der Codebasis, in Betracht gezogen werden. Dadurch wäre der Grundstein für den Einsatz dieser Technik bereits im Voraus gesetzt.

Für weiterführende Arbeiten bietet es sich auf Basis dieser Arbeit an, sich im Detail mit den Funktionen/der Funktionsweise von AFL++ auseinanderzusetzen. Dieser Fuzzer gehört zum aktuellen Zeitpunkt zu den Modernsten und wird stetig weiterentwickelt. Alle in dieser Arbeit dargelegten Informationen lassen sich grundsätzlich neben libFuzzer auch mit AFL/AFL++ anwenden. In diesem Bereich besteht das gleiche Problem, dass die erhältlichen Informationen, den Übergang vom Einsteiger zum erfahrenen Anwender/Entwickler nicht ermöglichen bzw. nur schwer ermöglichen. AFL++ bietet sehr viele Funktionen und die Dokumentation wird diesen an vielen Stellen nicht gerecht. Viele Dinge, die dem Einsteiger nicht bekannt sind, werden innerhalb der Dokumentation als gegeben angenommen. Daher würde eine detaillierte Auseinandersetzung mit den Funktionen/Funktionsweisen dieses Fuzzers einen enormen Mehrwert bieten. Des Weiteren wäre auf Basis des Structure-Aware Fuzzings mit Protocol Buffern, eine darauf aufbauende Entwicklung von spezifischen Fuzz-Tests denkbar. Beispielsweise wird diese Technik in der Entwicklung von Chromium eingesetzt, um die Struktur von SQL-Queries nachzubilden [64]. Dadurch wird das Fuzzen von jeglichen Testobjekten möglich, welche SQL-Queries in derartiger Form als Inputs erwarten. Die Entwicklung solcher spezifischen aber dennoch weitflächig anwendbaren Fuzz-Tests, eignet sich gut für weiterführende Arbeiten. Solche Entwicklungen sind wiederverwendbar und können vielen Softwareentwicklern einen Mehrwert bieten. Zuletzt sei bezüglich weiterführender Arbeiten auf `FuzzBench` [65] hingewiesen. Wie sich gezeigt hat, ist der Vergleich unterschiedlicher Fuzzer bzw. Fuzz-Techniken nur schwer möglich. Um dieses Problem zu lösen, versucht `FuzzBench` eine Art Benchmark bereitzustellen, um die Leistungsfähigkeit unterschiedlicher Fuzzer und Fuzz-Techniken in der Forschung vergleichen zu können. Auch hier bietet es sich an, diesen Benchmark in weiterführenden Arbeiten genauer zu untersuchen und eventuell mit dem Structure Aware Fuzzing mit Protocol Buffern durchzuführen.

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, dec 1990.

[2] Wikipedia - Heartbleed. https://en.wikipedia.org/wiki/Heartbleed (besucht am 09.06.24).

[3] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Retrieved 2024-01-18 17:28:37+01:00.

[4] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2008.

[5] R. Patton. *Software Testing*. Sams Publishing, 800 E. 96th St., Indianapolis, Indiana, 46240 USA, 2001.

[6] Donald Firesmith. Using V Models for Testing. Carnegie Mellon University, Software Engineering Institute's Insights (blog), Nov 2013. Accessed: 2024-Jun-11.

[7] Felix Hübner. Complete Model-Based Testing Applied to the Railway Domain, 2018.

[8] Deutsche Kommission Elektrotechnik Elektronik Informationstechnik. *DIN EN 50129 (VDE 0831-129)*. VDE, 06/2019.

[9] Deutsche Kommission Elektrotechnik Elektronik Informationstechnik. *DIN EN 50128 (VDE 0831-128)*. VDE, 03/2012.

[10] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software Testing Techniques: A Literature Review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 177–182, 2016.

[11] Edsger W. Dijkstra. The Humble Programmer. *Commun. ACM*, 15(10):859–866, oct 1972.

[12] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[13] Cloudflare Inc. What is penetration testing? https://www.cloudflare.com/learning/security/glossary/what-is-penetration-testing/ (besucht am 21.06.24).

[14] LLVM Compiler Infrastructure. https://llvm.org/ (besucht am 12.07.24).

[15] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[16] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[17] MoldStud. The Impact of Software Vulnerabilities in Today's World. 2024. https://moldstud.com/articles/p-the-impact-of-software-vulnerabilities-in-todays-world (besucht am 09.06.2024).

[18] National Vulnearability Database. https://nvd.nist.gov/general (besucht am 09.06.24).

[19] Google and Alphabet Vulnerability Reward Program. https://bughunters.google.com/about/rules/google-friends/6625378258649088/google-and-alphabet-vulnerability-reward-program-vrp-rules (besucht am 09.06.24).

[20] Esa Jääskelä. Genetic Algorithm in Code Coverage Guided Fuzz Testing. *University of Oulu, Department of Computer Science and Engineering*, 2016. https://oulurepo.oulu.fi/handle/10024/6947 (besucht am 02.06.2024)).

[21] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium*, 2008.

[22] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, feb 2013.

[23] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based Directed Whitebox Fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, 2009.

[24] libfuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html (besucht am 07.06.24).

[25] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[26] Frances E. Allen. Control Flow Analysis. *SIGPLAN Not.*, 5(7):1–19, jul 1970.

[27] Richárd Dévai, Judit Jász, Csaba Nagy, and Rudolf Ferenc. Designing and Implementing Control Flow Graph for Magic 4th Generation Language. *Acta Cybern.*, 21:419–437, 2014.

[28] AFL Reading Notes 1: Instrumentation, Initialization and Fork Server. https://mem2019.github.io/jekyll/update/2019/08/09/AFL-Fuzzer-Notes-1.html (besucht am 08.06.24).

[29] Technical "Whitepaper"for afl-Fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt (besucht am 08.06.24).

[30] AFL++ Documentation - Fuzzing in Depth. https://aflplus.plus/docs/fuzzing_in_depth/ (besucht am 09.06.24).

[31] Clang Documentation - Source-based Code Coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html (besucht am 24.06.24).

[32] Uniform Resource Locator. https://de.wikipedia.org/wiki/Uniform_Resource_Locator (besucht am 22.06.24).

[33] Yuma Jitsunari and Yoshitaka Arahori. Coverage-Guided Learning-Assisted Grammar-Based Fuzzing. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 275–280, 2019.

[34] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1, 12 2018.

[35] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2021.

[36] Harrison Green. Learning About Structure-Aware Fuzzing and Finding JSON Bugs to Boot, 2022. https://www.mayhem.security/blog/learning-about-structure-aware-fuzzing-and-finding-json-bugs-to-boot (besucht am 24.06.24).

[37] Google LLC. Documentation - Protocol Buffers. https://protobuf.dev/ (besucht am 24.06.24).

[38] Code Sanitizer. Wikipedia. https://en.wikipedia.org/wiki/Code_sanitizer (besucht am 25.05.2024).

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Address-Sanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*, 2012.

[40] AddressSanitizer - Documentation. https://github.com/google/sanitizers/wiki/AddressSanitizer (besucht am 25.05.24).

[41] AddressSanitizer - Algorithm. https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm (besucht am 25.05.24).

[42] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: Fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, 2015.

[43] Documentation - Memory Sanitizer. https://github.com/google/sanitizers/wiki/MemorySanitizer (besucht am 27.06.24).

[44] John Regehr. A Guide to Undefined Behavior in C and C++. Embedded in Academia Blog, July 2010. (besucht am 28.06.24).

[45] Chris Lattner. What Every C Programmer Should Know About Undefined Behavior. The LLVM Project Blog, May 2011. (besucht am 28.06.24).

[46] Clang Documentation - Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (besucht am 28.06.24).

[47] Clang Documentation - Memory Sanitizer. https://clang.llvm.org/docs/MemorySanitizer.html (besucht am 29.06.24).

[48] Clang Documentation - Address Sanitizer. https://clang.llvm.org/docs/AddressSanitizer.html (besucht am 29.06.24).

[49] Protocol Buffer Documentation - Overview. https://protobuf.dev/overview/ (besucht am 30.06.24).

[50] Apache2.0-License. https://www.apache.org/licenses/LICENSE-2.0.html (besucht am 30.06.24).

[51] 3-Clause BSD-License. https://opensource.org/license/bsd-3-clause (besucht am 30.06.24).

[52] Protocol Buffer Basics: C++. https://protobuf.dev/getting-started/cpptutorial/ (besucht am 30.06.24).

[53] Protocol Buffer Documentation - proto2 Syntax. https://protobuf.dev/programming-guides/proto2/ (besucht am 30.06.24).

[54] Protocol Buffer Documentation - proto3 Syntax. https://protobuf.dev/programming-guides/proto3/(besucht am 30.06.24).

[55] Proto2 vs Proto3. https://www.hackingnote.com/en/versus/proto2-vs-proto3/ (besucht am 30.06.24).

[56] GitHub - Protobuf. https://github.com/protocolbuffers/protobuf (besucht am 30.06.24).

[57] Protocol Buffer: C++ Generated Code Guide. https://protobuf.dev/getting-started/cpptutorial/ (besucht am 30.06.24).

[58] Robot Operating System - Introduction. http://wiki.ros.org/ROS/Introduction (besucht am 03.07.24).

[59] GitHub - libprotobuf-mutator. https://github.com/google/libprotobuf-mutator/tree/master (besucht am 04.07.24).

[60] Matt Morehouse Kostya Serebryany, Vitaly Buka. Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator, Oktober 2017. https://llvm.org/devmtg/2017-10/slides/Serebryany-Structure-aware

[61] DockerHub - AFL++ Image. https://hub.docker.com/r/aflplusplus/aflplusplus (besucht am 04.07.24).

[62] GitHub - Abseil - C++ Common Libraries. https://github.com/abseil/abseil-cpp (besucht am 04.07.24).

[63] GitHub - libprotobuf-mutator - af3bb18 Branch. https://github.com/google/libprotobuf-mutator/tree/af3bb18749db3559dc4968dd85319d05168d4b5e (besucht am 04.07.24).

[64] Fuzzing SQL Queries with Protocol Buffers. https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md#example-sqlite (besucht am 12.07.24).

[65] FuzzBench: Fuzzer Benchmarking As a Service. https://github.com/google/fuzzbench (besucht am 12.07.24).