Fachhochschule Nordwestschweiz
Hochschule für Technik

Entwicklung eines Token-basierten LibAFL Fuzzers für Bildverarbeitungsbibliotheken

IP6-Arbeit

Windisch, April 2025

Studentin/Student	Hoang Viet Nguyen Alessandro Lenti
Betreuer/in	Christopher Scherb Alexander Stiermer

Fachhochschule Nordwestschweiz, Hochschule für Technik

Abstract

Diese Arbeit stellt einen neuartigen Token-basierten Fuzzer vor, der mithilfe der Open-Source-Fuzzing-Bibliothek **LibAFL** automatisch wiederkehrende Byte-Sequenzen ("Tokens") aus binären Eingaben extrahiert und im Fuzzing-Prozess gezielt einsetzt. Grundlage ist ein speziell entwickelter **Token-Discovery-Algorithmus**, der stabile Byte-Bereiche anhand von Coverage-Feedback identifiziert. Ergänzende Optimierungsmethoden wie Token-Cleaning, Filtering, maximale Token-Größe und Token-Aging steigern Effizienz und Relevanz der erkannten Tokens.

Die Implementierung wurde anhand der Bildverarbeitungsbibliotheken **libpng** und **libmozjpeg** evaluiert. Für libpng konnten sowohl vordefinierte Tokens (z. B. IHDR, PLTE) als auch neue, bisher nicht berücksichtigte Strukturen erfolgreich erkannt werden. Die Ergebnisse zeigen, dass der Fuzzer mit automatisch ermittelten Tokens in Codeabdeckung und Corpus-Größe mit manuell konfigurierten Token-Fuzzern konkurrieren kann. Tests mit libmozjpeg bestätigten zudem die Übertragbarkeit des Ansatzes auf andere Formate und ermöglichten die Identifikation zusätzlicher potenzieller Schwachstellen (Objectives).

Damit liefert diese Arbeit einen Beitrag zur Automatisierung des tokenbasierten Fuzzings für strukturierte Binärdaten, reduziert den manuellen Vorbereitungsaufwand und eröffnet Perspektiven für die effiziente Testung weiterer komplexer Dateiformate.

Inhaltsverzeichnis

1 Danksagung

Wir danken herzlich Herrn Christoph Scherb, der uns als Betreuer dieses Projekts stets tatkräftig unterstützt und mit wertvollen Anregungen begleitet hat. Ebenso gilt unser Dank der FHNW, ohne deren Unterstützung die umfassenden Tests nicht möglich gewesen wären.

2 Einleitung

Die stetig wachsende Komplexität moderner Software und der hohe Anspruch an deren Zuverlässigkeit erfordern effiziente und automatisierte Testverfahren. Eine bewährte Methode zur Erkennung von Sicherheitslücken und Robustheitsproblemen ist das Fuzzing – ein Prozess, bei dem Programme mit einer Vielzahl zufällig oder gezielt veränderter Eingaben getestet werden, um unerwartetes Verhalten auszulösen. Während sich klassische Fuzzer auf Byte- oder Grammatik-basierte Mutationen stützen, gewinnt das Token-Level-Fuzzing zunehmend an Bedeutung. Dieser Ansatz nutzt wiederkehrende Strukturen innerhalb der Eingaben, um gezieltere und oft effektivere Tests zu ermöglichen.

Im Rahmen dieser Arbeit wird ein neuartiger Token-basierten Fuzzer entwickelt, der auf der Open-Source-Bibliothek LibAFL aufbaut und speziell für Bildverarbeitungsbibliotheken wie libpng und libmozjpeg konzipiert ist. Herzstück der Lösung ist ein Token-Discovery-Algorithmus, der relevante Token automatisch aus binären Daten extrahiert und in den Fuzzing-Prozess integriert. Ziel ist es, die Effizienz und Tiefe der Testabdeckung zu steigern, ohne auf vordefinierte Token angewiesen zu sein.

Neben der Konzeption und Implementierung des Algorithmus werden in dieser Arbeit auch verschiedene Optimierungsstrategien vorgestellt, die den Entdeckungsprozess beschleunigen und die Qualität der identifizierten Token verbessern. Die Wirksamkeit des Ansatzes wird anhand umfangreicher Experimente evaluiert und mit etablierten Fuzzing-Strategien verglichen.

Um die Arbeit vollständig einordnen zu können, werden zunächst zentrale Grundlagen des Fuzzings erläutert. Dazu zählen die Mechanismen, mit denen ein Fuzzer Eingaben durch die Programmstruktur verfolgt, die gängigen Ansätze zur Programmanalyse sowie die Methoden zur gezielten Mutation von Eingaben, um Abstürze oder unerwartetes Verhalten zu provozieren. Darüber hinaus wird ein Überblick über die wichtigsten Werkzeuge und Komponenten von **LibAFL** gegeben, die für die Umsetzung dieser Arbeit genutzt wurden.

Abschließend wird die Performanz unseres Fuzzers im Vergleich zu den LibAFL-Beispielfuzzern untersucht. Dabei zeigen wir, dass unser Ansatz nicht nur mithalten kann, sondern in bestimmten Bereichen sogar bessere Ergebnisse erzielt. Zur objektiven Bewertung stellen wir zudem zwei Formeln vor, mit denen sich die Effektivität eines Fuzzers quantifizieren lässt.

3 Grundlagen

Im folgenden Kapitel werden die Grundlagen des Fuzzing behandelt. Zunächst klären wir, was unter einem Fuzzer zu verstehen ist. Bevor wir zur Problemstellung übergehen, erläutern wir den aktuellen Stand der Forschung. Dabei werden unter anderem Konzepte wie Coverage Tracking und tokenbasiertes Fuzzing vorgestellt. Abschliessend werfen wir einen Blick auf die LibAFL-Bibliothek. Sie ist in Rust implementiert und wurde zur Lösung der Problemstellung eingesetzt.

3.1 Grundlagen des Fuzzing

Fuzzing ist eine automatisierte Testmethode zur Identifikation von Softwarefehlern und Sicherheitslücken. Ein Fuzzer verändert Input Daten gezielt nach bestimmten Strategien oder durch zufällige Mutation. Diese neu generierten Inputs werden einem Zielprogramm übergeben und anschliessend wird das Programmverhalten überwacht. Ziel ist es, durch unerwartete oder fehlerhafte Inputs unerwünschtes Verhalten wie Programmabstürze auszulösen.

Die Verbindung zwischen dem Fuzzer und dem Zielprogramm erfolgt über ein sogenanntes *Harness*. Dieses Hilfsprogramm nimmt die generierten Inputs entgegen, verarbeitet sie gegebenenfalls vor, und übergibt sie in geeigneter Form an die zu testende Komponente.

Fuzzing-Tools speichern Inputs, die zu Abstürzen oder Fehlverhalten führen, um diese später zu analysieren und als Grundlage für Verbesserungen der Robustheit und Sicherheit der Software nutzen zu können. Abbildung 3.1 zeigt den typischen Ablauf eines Fuzzing-Prozesses.

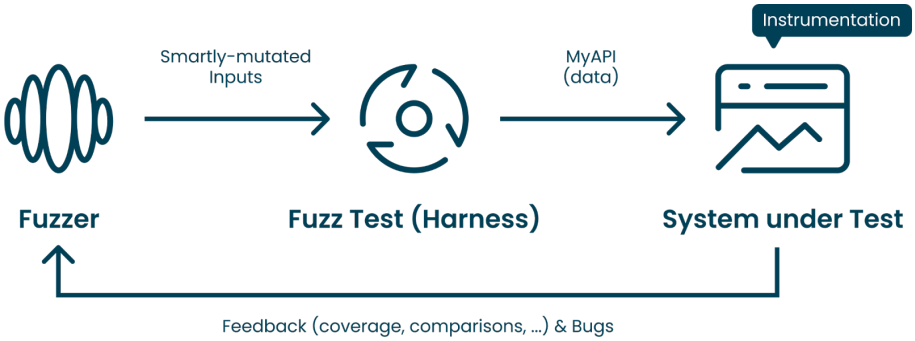

Abbildung 3.1: Ablauf eines Fuzzing-Prozesses. Bildquelle: [1]

Eine der frühesten bekannten Arbeiten zum Thema Fuzzing wurde 1988 von Barton Miller und seinem Team an der University of Wisconsin veröffentlicht. Ziel der Studie war es, die Robustheit von Unix-Dienstprogrammen zu analysieren, indem zufällig generierte Inputs auf die Programme angewendet wurden. Zu diesem Zweck entwickelten sie ein einfaches Tool zur Erzeugung randomisierter Zeichenfolgen und beobachteten das Verhalten der getesteten Software unter diesen Bedingungen [2], [3].

Moderne Fuzzer verwenden heute eine Vielzahl unterschiedlicher Strategien zur Generierung von Inputs. Parallel dazu hat sich das Forschungsfeld des Fuzzings erheblich weiterentwickelt, sodass inzwischen auch komplexe Systeme wie Betriebssystemkerne automatisiert auf ihre Robustheit gegenüber unerwarteten Inputs getestet werden können. Die folgende Sektion gibt einen Überblick über den aktuellen Stand der Forschung.

3.2 Stand der Forschung

Die Forschung zu Fuzzern lässt sich in drei zentrale Bereiche gliedern: Input Generierung, Programmanalyse und Coverage Goals. Während sich der erste Bereich auf Methoden zur systematischen oder zufälligen Erstellung von Testeingaben konzentriert, untersucht die Programmanalyse, wie Informationen über das Verhalten des Zielprogramms gewonnen werden können. Coverage Goals definieren schliesslich, welche Programmteile durch Fuzzing erreicht werden sollen, und wie Mutationen gesteuert werden müssen, um neue Ausführungspfade zu entdecken.

3.2.1 Input Generierung

Ein zentraler Bereich in der Forschung zu Fuzzing ist die systematische Generierung von Inputs für ein Zielprogramm. Die Strategien zur Input-Generierung lassen sich grob in drei Kategorien unterteilen: Byte-Level-Mutationen, Grammatik-basiertes Fuzzing und Token-Level-Fuzzing. Byte-Level-Mutationen operieren auf der rohen Repräsentation der Input Daten, typischerweise als Bytestreams. Sie sind unabhängig von semantischen oder strukturellen Eigenschaften des Inputformats und eignen sich deshalb gut für binäre Formate mit unbekannter Struktur.

Ein prominentes Beispiel für diese Strategie ist die sogenannte Havoc-Mutation, welche erstmals in AFL eingeführt wurde [4]. Havoc kombiniert eine Reihe einfacher, aber vielseitiger Mutationen, um Inputs auf verschiedene Weise zu verändern. Abbildung 3.2 zeigt ihnen das Grundkonzept von Havoc.

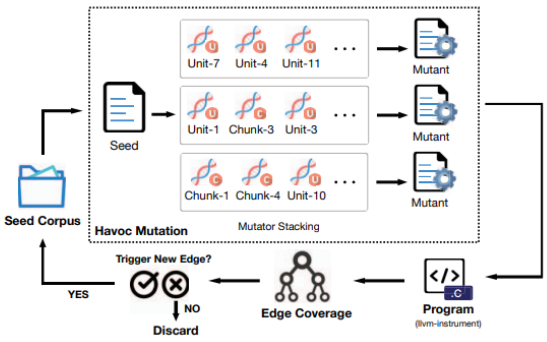

Figure 1: The framework of *Havoc*

Table 1: Mutation operators defined by *Havoc*

Type	Meaning	Mutator
bitflip	Flip a bit at a random position.	`bitflip 1`

interesting values	Set bytes with hard-coded interesting values.	`interest 8`
		`interest 16`
		`interest 32`

arithmetic increase	Perform addition operations.	`addition 8`
		`addition 16`
		`addition 32`

arithmetic decrease	Perform subtraction operations.	`decrease 8`
		`decrease 16`
		`decrease 32`

random value	Randomly set a byte to a random value.	`random byte`
delete bytes	Randomly delete consecutive bytes.	`delete chunk bytes`
clone/insert bytes	Clone bytes in 75%, otherwise insert a block of constant bytes.	`clone/insert chunk bytes`
overwrite bytes	Randomly overwrite the selected consecutive bytes.	`overwrite chunk bytes`

Abbildung 3.2: Grundidee und Mutations Strategien von Havoc. Bildquelle: [5]

Die Stärke von Havoc liegt in der Diversität der Mutationen und der Tatsache, dass diese rekursiv und in zufälligen Kombinationen angewendet werden, wodurch eine hohe Entropie der generierten Inputs entsteht. Trotz ihrer scheinbar simplen Natur erzielen Havoc-Mutationen in der Praxis häufig eine bemerkenswerte Codeabdeckung. In der vergleichenden Studie von Wu et al. [5] outperformte eine rein Havoc-basierte Mutationsstrategie sogar viele moderne Fuzzer. In diesem Projekt wird Havoc ebenfalls verwendet um in der frühen Fuzzing-Phasen Mutationen zu erzeugen.

Byte-Level-Mutationen haben eine zentrale Schwäche: Sie können nicht effektiv mit strukturierten Inputformaten umgehen. Wird beispielsweise versucht, eine interpretierte Sprache wie Python mit einem rein Havoc-basierten Fuzzer zu testen, enden die meisten Mutationen in trivialen Syntaxfehlern, die den Parser nicht überstehen.

An dieser Stelle setzt Grammatik basiertes Fuzzing an. Es generiert Inputs gezielt auf Basis einer formalen Grammatik, wodurch syntaktisch gültige Programme erzeugt werden. Dies ermöglicht tiefere Codeabdeckung und führt oft zur Entdeckung von Bugs in tiefer liegenden Komponenten des Zielprogramms [6].

Ein Nachteil ist jedoch der initiale Aufwand zur Definition einer geeigneten Grammatik. Zudem bleiben Fehler, die nur durch ungültige oder grenzwertige Syntax ausgelöst werden, häufig unentdeckt, da sie ausserhalb der definierten Grammatik liegen [7].

Einen hybriden Ansatz zwischen Byte-Level-Mutationen und Grammatik basiertem Fuzzing stellt die Methode des Token-Level-Fuzzing dar, wie sie von Salls et al. vorgestellt wurde [7]. Dabei wird die Eingabe mit einem Tokenizer in sprachspezifische Einheiten wie Schlüsselwörter, Literale, Operatoren und Trennzeichen zerlegt. Abbildung 3.3 veranschaulicht den neuartigen Fuzzing Prozess.

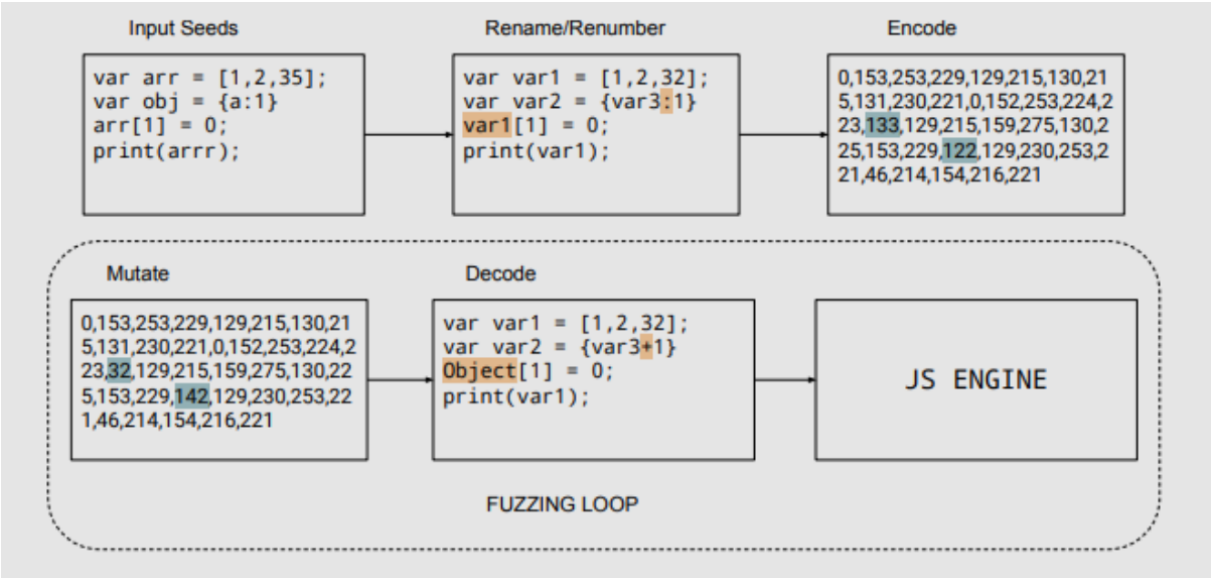

Abbildung 3.3: Token-Level Fuzzing Konzept, Bildquelle:[7]

Als Tokens werden allgemein die folgende Dinge gesehen:

- Zahlen
- Sonderzeichen
- Schlüsselwörter (z.B. return).
- Variablen

Diese Tokens werden anschliessend auf numerische Repräsentationen abgebildet, sodass klassische Mutationen nicht auf rohen Bytes, sondern auf der numerischen Repräsentation des Inputs erfolgt. So lassen sich strukturierte, aber dennoch vielfältige Inputs generieren, die häufig valide Syntax aufweisen, aber ungewöhnliche semantische Kombinationen testen. Dieser Ansatz bildet die Grundlage für den in dieser Arbeit entwickelten Fuzzing-Ansatz und wird ausführlicher im Abschnitt 3.2.4 erläutert.

In dieser Arbeit soll untersucht werden, wie der Fuzzing-Prozess für Bildverarbeitungsbibliotheken optimiert werden kann. Als Grundlage dient dabei Token-Level Fuzzing. Der Einsatz Grammatik basierten Fuzzings ist für dieses Projekt nicht geeignet, da Bilddaten keine feste, formale Struktur aufweisen, die sich sinnvoll in einer Grammatik abbilden liesse.

Zur Initialisierung der Eingaben kommen in der ersten Phase Havoc-Mutationen zum Einsatz, um eine breite Varianz an Inputs zu erzeugen. Aufbauend auf dieser Grundlage wird in dieser Arbeit untersucht, ob sich innerhalb der Inputs wiederkehrende Strukturen identifizieren lassen, die gezielt manipuliert werden können. Ziel ist es, zu evaluieren, ob eine Token basierte Mutation den Fuzzing-Prozess effizienter gestaltet und zur Aufdeckung tieferliegender Fehler beiträgt.

3.2.2 Programmanalyse

Ein weiterer zentraler Aspekt moderner Fuzzing-Systeme ist die Art und Weise, wie Informationen über das Zielprogramm gesammelt werden – dieser Bereich wird oft unter dem Begriff *Programmanalyse* bzw. *Program Access Level* zusammengefasst. In der Literatur wird typischerweise zwischen drei Ansätzen unterschieden:

- **White-Box-Fuzzing** verwendet vollständigen Zugriff auf den Quellcode. Dadurch lassen sich Mutationen gezielt anhand von Codepfaden oder Variablenabhängigkeiten ableiten.
- **Black-Box-Fuzzing** arbeitet ohne jeglichen Einblick in den Quellcode. Entscheidungen über Mutationen beruhen ausschliesslich auf beobachtbarem Verhalten wie Rückgabewerten, Laufzeit oder Abstürzen.
- **Grey-Box-Fuzzing** stellt einen praktischen Mittelweg dar. Obwohl der Quellcode nicht direkt genutzt wird, sammelt das Programm während der Ausführung zusätzliche Informationen, z.B. welche Codeabschnitte ausgeführt wurden. Diese helfen dem Fuzzer, gezielter neue Inputs zu erzeugen.

Grey-Box-Ansätze, wie sie etwa in AFL oder libFuzzer umgesetzt sind, dominieren heute viele praxisnahe Fuzzer, da sie eine gute Balance zwischen Aufwand und Effektivität bieten.

In diesem Projekt wird ein Grey-Box-Ansatz verfolgt, da LibAFL als Fuzzing-Bibliothek zum Einsatz kommt. LibAFL bietet ein flexibles Framework zur Beobachtung verschiedener Zielprogramme während des Fuzzing-Prozesses und ermöglicht die Integration unterschiedlicher Feedback-Mechanismen. Es basiert konzeptionell auf AFL++, einer Weiterentwicklung des populären AFL-Fuzzers, der ebenfalls dem Grey-Box-Paradigma folgt. Durch diese Wahl lassen sich Informationen über die Programmausführung effizient nutzen, ohne direkten Zugriff auf den Quellcode zu benötigen. Weitere Informationen finden Sie in Abschnitt 3.4.

3.2.3 Coverage Goals

Um robuste, zuverlässige und sichere Software zu entwickeln, muss diese sorgfältig auf eine Vielzahl möglicher Inputs und Outputs geprüft werden. Unit-Tests und Integrationstests stellen sicher, dass das Programm bei kontrollierten Eingaben die korrekten Ausgaben liefert. In der Praxis stösst man jedoch schnell an Grenzen, da das vollständige Testen eines Programms wegen der grossen Anzahl potentieller Inputs kaum möglich ist.

Fuzzer bieten hier eine vielversprechende Alternative. Sie ermöglichen es, Programme automatisiert mit einer grossen Bandbreite an Eingaben zu testen. Ein Nachteil einfacher Fuzzer besteht jedoch darin, dass sie oft sehr lange benötigen, um Eingaben zu generieren, die tieferliegende oder komplexe Programmteile tatsächlich ausführen.

Um die Qualität der generierten Eingaben zu verbessern, benötigen Fuzzer eine Strategie, die sich am internen Zustand des Programms orientiert. Dieser Ansatz ist unter dem Begriff *Coverage-Guided Fuzzing* bekannt. Dabei wird das Programmverhalten während der Ausführung analysiert, um Eingaben gezielt so zu mutieren, dass bislang unerreichte Pfade oder Zustände erkundet werden. Es existieren verschiedene Methoden zur Messung der Programmausführung und Programmabdeckung [8], unter anderem:

- **Statement Abdeckung**: Misst wie viele und wie oft Statements ausgeführt werden.
- **Zeilen Abdeckung**: Misst wie viele Zeilen Code ein Input Abdeckt.
- **Funktionsabdeckung**: Misst wie oft Funktionen aufgerufen werden.
- **Basis Block Abdeckung**: Zählt die Anzahl ausgeführter Programmblöcke – jeweils eine Sequenz aufeinanderfolgender Instruktionen.
- **Kanten Abdeckung**: Misst mögliche Interaktionen zwischen den Programmblöcken
- **Pfad Abdeckung**: Ermittelt zur Laufzeit die ausgeführten Pfade durch das Programm zur Analyse des Kontrollflusses.

Im Rahmen unseres Projekts verwenden wir die LLVM-Toolchain zur Instrumentierung der zu testenden Programme. Sowohl AFL als auch AFL++ bieten darüber hinaus zusätzliche Möglichkeiten zur Instrumentierung [9]. Auch LibAFL nutzt die LLVM-Infrastruktur, und viele Benchmark-Fuzzer basieren auf den Funktionen dieser Bibliothek. Aus diesem Grund konzentrieren wir uns in unserer Arbeit auf die Instrumentierungsmechanismen, die direkt durch LLVM bereitgestellt werden.

Die **LLVM-Toolchain** ist eine modulare Sammlung von Werkzeugen, die primär zur Entwicklung von Compilern eingesetzt wird. Darüber hinaus stellt sie Komponenten zur Verfügung, die es ermöglichen, Programme zu instrumentieren, zu optimieren und plattformabhängigen Maschinencode zu generieren. Die in LLVM integrierte Instrumentierung wird unter dem Namen *SanitizerCoverage* bereitgestellt. Diese Komponente ermöglicht die gezielte Instrumentierung benutzerdefinierter Funktionen und arbeitet auf *Funktions-*, *Basisblock-* und *Kantenebene* [10]. Abbildung 3.4 zeigt die drei Hauptphasen eines Compilers und verdeutlicht die Rolle, die LLVM dabei spielt.

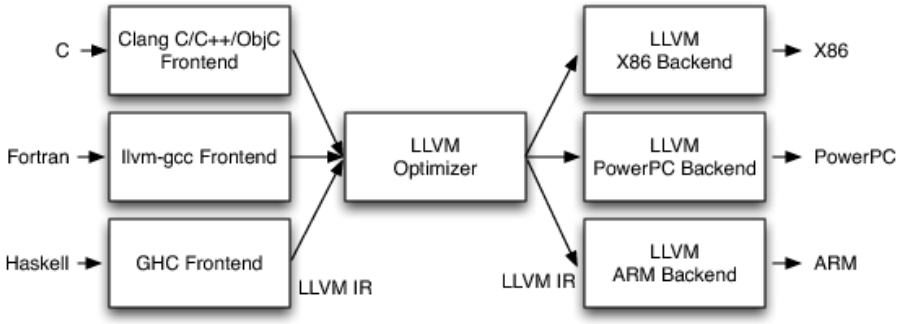

Abbildung 3.4: Nutzung der LLVM-Toolchain, Bildquelle:[11]

Um den eigenen Code zu instrumentieren, muss LLVM als Compiler für das Programm verwendet werden. Zusätzlich muss der Compiler-Flag −fsanitize−coverage=trace−pc−guard angegeben werden. LLVM fügt daraufhin zusätzliche Instruktionen in der *Intermediate Representation* (IR) ein, um Abdeckungsdaten zu sammeln. So werden beispielsweise am Anfang und Ende von Funktionen zusätzliche Instruktionen eingefügt, um deren Aufruf und Ablauf zu protokollieren.

Basic Blocks sind ein zentrales Konzept der Compilertechnik. Ein Basic Block ist eine Folge von Instruktionen mit genau einem Eintrittspunkt und einem Austrittspunkt, wobei es innerhalb des Blocks keine Verzweigungen gibt. Das bedeutet, dass die Ausführung immer am Anfang beginnt und bis zum Ende ohne Unterbrechung oder Sprünge fortläuft [12].

Damit der Quellcode in Basic Blocks unterteilt werden kann, wird er während der Kompilierung in eine Zwischenrepräsentation nach dem Konzept des Three-Address Code (TAC) überführt. Dabei werden komplexe Anweisungen in einfachere Anweisungen zerlegt, sodass jede Anweisung höchstens drei Operanden enthält: eine Zieladresse (für das Ergebnis) und zwei Quelladressen (für die Operanden) [13]. Abbildung 3.5 zeigt ein Beispiel wie C Code in TAC-Instruktionen umgewandelt werden.

```
// Ursprünglicher C-Ausdruck
x = (a + b) * (c - d);

// Three-Adress Code
t1 = a + b;
t2 = c - d;
x = t1 * t2;
```

Abbildung 3.5: Beispiel Umwandlung von C zu TAC

Abbildung 3.6 zeigt, anhand welcher Regeln ein Basic Block identifiziert wird. Drei zentrale Kriterien bestimmen dabei den Eintrittspunkt eines Blocks:

- Die erste TAC Instruktion gilt als Eintrittspunkt.
- Instruktionen welche das Ziel von jump/goto Statements sind gelten als Eintrittspunkt.
- Instruktionen welche nach jump/goto Statements folgen gelten als Eintrittspunkt.

```
1)  i=1        //Leader 1 (First statement)
2)  j=1             //Leader 2 (Target of 11th statement)
3)  t1 = 10 * i     //Leader 3 (Target of 9th statement)
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1                    //Leader 4 (Immediately following Conditional goto statement)
11) if i <= 10 goto (2)
12) i = 1                        //Leader 5 (Immediately following Conditional goto statement)
13) t5 = i - 1                   //Leader 6 (Target of 17th statement)
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Abbildung 3.6: Zwischenrepräsentation um eine 10x10 Matrix in eine Identitäts Matrix zu Transformieren, Bildquelle: [12]

Kanten in einem Programm beschreiben den Fluss von Daten zwischen Basic Blocks. Dabei erfolgt der Datenfluss jeweils vom Austrittspunkt eines Basic Blocks zum Eintrittspunkt eines anderen. Abbildung 3.7 veranschaulicht dieses Prinzip: Die gerichteten Pfeile können dabei als Kanten im Kontrollflussgraphen interpretiert werden [14].

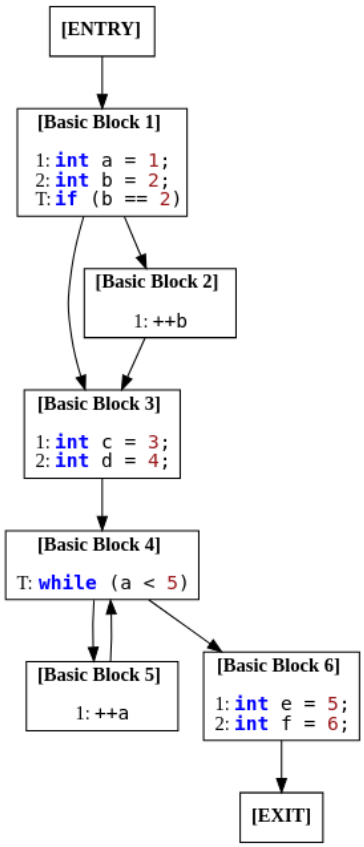

Abbildung 3.7: Kontrollfluss zwischen Basic Blöcken, Bildquelle: [15]

3.2.4 Token-Level Fuzzing

Token-Level Fuzzing wurde erstmals im Jahr 2021 von Salls et al. vorgestellt [7]. Der Ansatz zielt darauf ab, die jeweiligen Schwächen von rein Byte-Level, sowie Grammatik basierten Fuzzern zu überwinden. Das Prinzip wird anhand eines JavaScript-Fuzzers veranschaulicht. Dabei werden Variablen, Zahlen, Schlüsselwörter und Sonderzeichen als sogenannte Tokens betrachtet. Tokens stellen eine Zuordnung (Mapping) zwischen einem Wort und einer Zahl dar. Die Mutationen erfolgen auf der Ebene dieser Tokens, wobei einzelne Tokens vertauscht oder verändert werden. Bevor der Code an die JavaScript-Engine übergeben wird, werden die Tokens in ihre ursprünglichen Werte zurücktransformiert. Mögliche Mutationen sind in Abbildung 3.8 dargestellt.

```
while (bar.x) ⟶ if (bar.x)
              ⟶ Number (bar.x)
              ⟶ while (bar+x)
              ⟶ while (while.x)
```

Abbildung 3.8: Beispiel Mutationen, Bildquelle: [7]

Allerdings gibt es einige Herausforderungen bei diesem Ansatz. Zum einen existiert eine unendliche Anzahl möglicher Zahlen und Variablennamen, die als Tokens behandelt werden. Um den Fuzzing-Prozess nicht durch das Codieren und Decodieren unnötig zu verlangsamen, werden Zahlen auf die nächstgrössere Zweierpotenz gerundet, wobei ein Limit von 2^{32} gesetzt wird. Laut der zugrunde liegenden Studie reichen bereits wenige Variablennamen aus, um Bugs zu identifizieren. Daher wurde die Anzahl der verwendeten Variablen auf 15 begrenzt. Abbildung 3.9 zeigt die vollständige Architektur des Fuzzers.

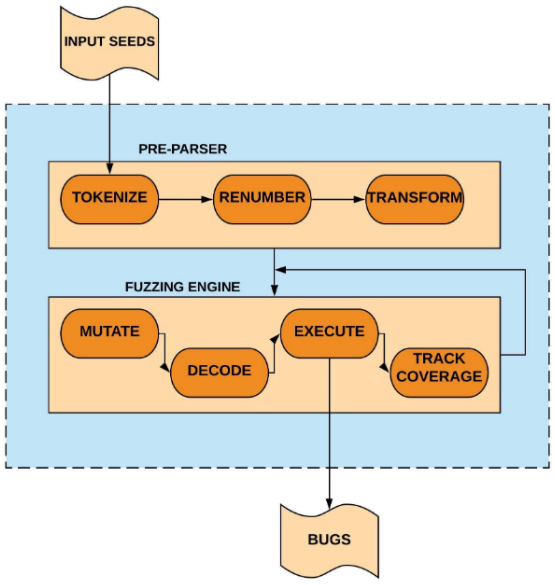

Abbildung 3.9: Token-Level Fuzzer Architektur, Bildquelle: [7]

Salls et al. konnten in ihrer Studie zeigen, dass ihr Ansatz 29 Fehler aufdeckte, die weder durch Byte-Level, noch durch Grammatik basierte Fuzzer gefunden wurden [7]. Aufbauend auf diesem Verfahren untersucht die vorliegende Arbeit, inwieweit das Konzept des Token-Level Fuzzing auch auf binäre Formate übertragbar ist.

3.3 Problemstellung

Dieses Projekt stellt eine Erweiterung des Konzepts *Token-Level Fuzzing* dar. Im Fokus steht die Untersuchung, ob sich aus binären Eingaben aussagekräftige Tokens extrahieren lassen und wie diese automatisiert identifiziert werden können.

Hierzu wird zunächst ein Verfahren zur Token-Extraktion aus binären Daten entwickelt. Basierend auf diesen Erkenntnissen wird ein Algorithmus entworfen, der unter Verwendung der Open-Source-Fuzzing-Toolchain *LibAFL* potenzielle Tokens identifiziert.

Die entdeckten Tokens werden anschliessend in den Fuzzing-Prozess integriert, um den Einfluss des Token-Discovery-Mechanismus auf die Fuzzing-Performance zu evaluieren. Als erste Testumgebung dient die weit verbreitete Bildverarbeitungsbibliothek *libpng*, für die in *LibAFL* bereits funktionierende Beispiel-Fuzzer mit vordefinierten Tokens existieren. Das Ziel dieser Arbeit besteht darin, die gegebenen Tokens zu rekonstruieren und damit die Grundlage für ein automatisiertes tokenbasiertes Fuzzing binärer Formate zu schaffen.

3.4 LibAFL

Ein moderner Fuzzer setzt sich aus mehreren modularen Komponenten zusammen. Dazu zählen unter anderem eine Mutations-Engine zur Erzeugung neuer Testfälle, ein Instrumentierungsmechanismus zur Überwachung der Programmausführung sowie ein Feedback-System, das auf Basis gesammelter Metriken geeignete Mutationen auswählt. Zusätzlich müssen Testfälle, die zu Abstürzen oder unerwartetem Verhalten führen, für eine nachgelagerte Analyse persistiert werden.

Um diese Komponenten nicht selbst implementieren zu müssen und gleichzeitig auf ein flexibles, erweiterbares Framework zurückgreifen zu können, wird in dieser Arbeit die Open-Source-Bibliothek *LibAFL* verwendet. LibAFL bietet eine umfassende Infrastruktur für die Entwicklung und Ausführung benutzerdefinierter Fuzzer und unterstützt dabei sowohl konventionelle als auch experimentelle Fuzzing-Strategien.

Die Architektur von LibAFL basiert auf acht zentralen, modular aufgebauten Komponenten. Jede dieser Komponenten ist individuell konfigurierbar und kann bei Bedarf erweitert oder ersetzt werden. Die Bibliothek ist in *Rust* implementiert, nutzt jedoch unter anderem auch die *LLVM*-Engine, um Laufzeitverhalten zu analysieren [16].

Abbildung 3.10 zeigt die systematische Struktur des Frameworks, wie sie in der ursprünglichen Studie von Fioraldi et al. eingeführt wurde. Sie veranschaulicht die Interaktion der Kernkomponenten innerhalb eines typischen Fuzzing-Workflows.

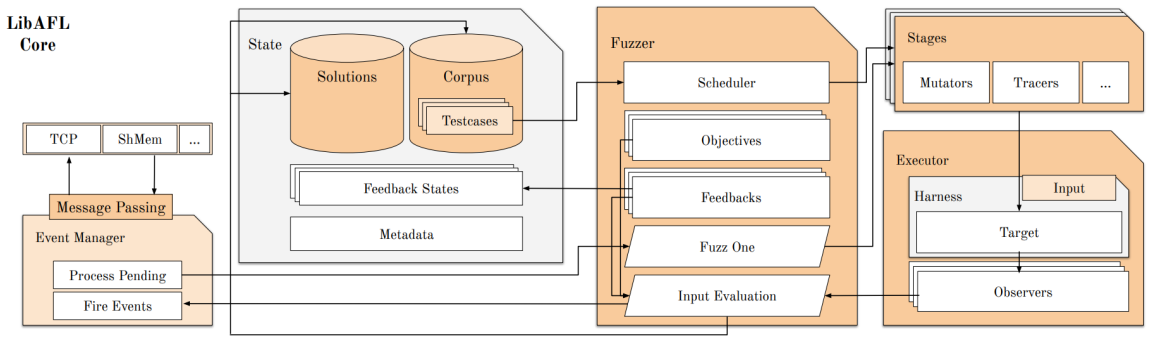

Abbildung 3.10: Kernkonzepte und Interaktionen von LibAfl. Bildquelle: [16]

Die folgenden Komponenten bilden das Kernsystem von LibAFL:

- **Observer**: Beobachtet und sammelt Informationen während der Programmausführung. Beispiele hierfür sind Coverage-Metriken, Laufzeitverhalten oder bestimmte Speicherzugriffe.
- **Executor**: Führt das Zielprogramm mit einem gegebenen Input aus. Er sorgt für die Übergabe des Inputs und die Erfassung des Laufzeitergebnisses, zum Beispiel Abstürze oder Exceptions.
- **Feedback**: Bewertet die Ausführungsergebnisse und entscheidet, ob ein Input als „interessant“ gilt. Diese Bewertung erfolgt typischerweise auf Grundlage der von einem oder mehreren Observern bereitgestellten Daten.
- **Input**: Repräsentiert die Testeingaben für das Zielprogramm. In den meisten Fällen handelt es sich um einfache Byte-Arrays, es können aber auch komplexere Strukturen wie Syntaxbäume verwendet werden – insbesondere bei grammatikbasierten Fuzzern.
- **Corpus**: Bezeichnet die Sammlung aller als interessant bewerteten Inputs. Zusätzlich enthält das Corpus auch sogenannte *Solutions* – also Inputs, die zum Absturz des Zielprogramms oder zu anderem sicherheitsrelevantem Fehlverhalten geführt haben.
- **Mutator**: Verändert bestehende Inputs durch gezielte oder zufällige Mutationen, um neue Testfälle zu generieren.
- **Generator**: Erzeugt neue Eingaben unabhängig von einem initialen Corpus. Besonders bei format- oder grammatikbasierten Fuzzing-Ansätzen wird der Generator zur systematischen Exploration des Eingaberaums eingesetzt.
- **Stage**: Definiert eine konkrete Verarbeitungseinheit für einen Input, zum Beispiel eine Mutationsrunde oder eine Analyseoperation. Im Rahmen dieser Arbeit wird eine spezielle *Token-Discovery-Stage* implementiert, welche versucht, wiederkehrende Token innerhalb binärer Eingaben zu identifizieren.

4 Konzept

In diesem Kapitel wird das Konzept des Kern-Algorithmus vorgestellt. Ziel des Algorithmus ist es, durch Analyse der Coverage Maps relevante Tokens aus den Eingaben zu extrahieren. Um eine Verlangsamung des Fuzzing-Prozesses zu vermeiden, werden ergänzend Verfeinerungsmethoden präsentiert, die den Algorithmus effizienter gestalten.

4.1 Konzept: Token-Discovery Algorithmus

Der Algorithmus dient dazu, zusammenhängende Byte-Sequenzen im Input zu identifizieren. Eine Byte-Sequenz gilt dann als zusammenhängend, wenn kleinere Veränderungen innerhalb dieser Sequenz keine Änderungen in der Coverage Map verursachen. Um solche Sequenzen zu erkennen, werden zunächst die Positionen (Indizes) ermittelt, an denen mutierte Eingaben vom Original abweichen.

Im nächsten Schritt werden diese Mutationen gezielt in das Original eingefügt, um die resultierende Coverage Map zu analysieren. Der Algorithmus verschiebt dabei die Mutation iterativ nach links und rechts, bis eine Veränderung der Coverage Map auftritt. Die dabei gefundene stabile Sequenz wird anschliessend als Token abgespeichert.

Abbildung 4.1 veranschaulicht die einzelnen Schritte des Token-Discovery Algorithmus: Zunächst wird ein zufälliges Byte im Input verändert (a). Danach wird schrittweise das links benachbarte Byte modifiziert, wobei jeweils geprüft wird, ob die Coverage Map unverändert bleibt (b). Dieser Vorgang wird solange fortgesetzt, bis eine Modifikation zu einer veränderten Coverage Map führt – das entsprechende Byte gehört dann nicht mehr zum Token (c). Der gleiche Prozess wird anschliessend für die rechts benachbarten Bytes durchgeführt (d).

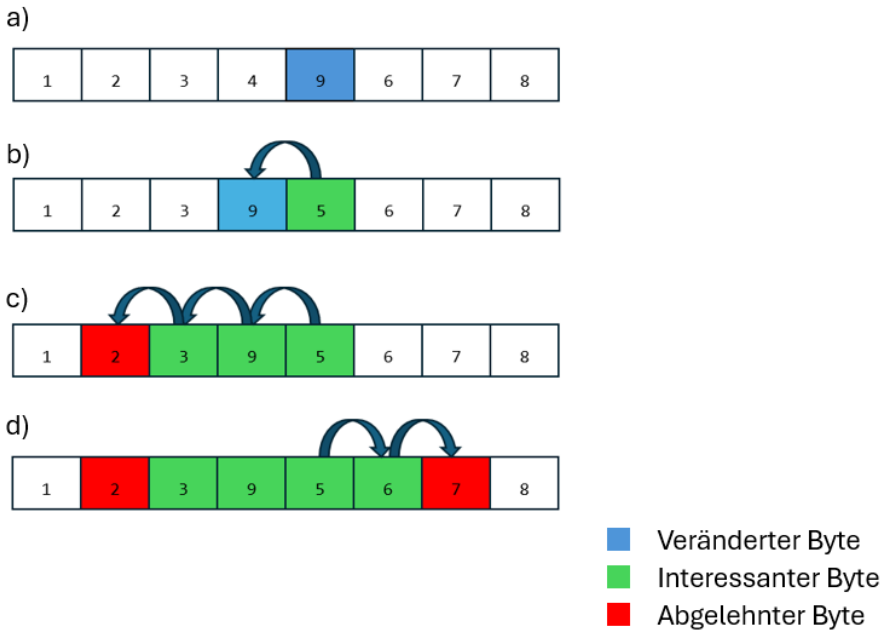

Abbildung 4.1: Schritte des Token-Discovery Algorithmus: (a) Ein zufälliges Byte wird verändert. (b) Der linke Nachbar wird verändert, und es wird überprüft, ob dieselbe Coverage Map entsteht. (c) Schritt b wird wiederholt, bis eine neue Coverage Map erzeugt wird. (d) Der Prozess aus c wird auch auf der rechten Seite durchgeführt.

4.2 Verfeinerungsmethoden

Der oben aufgeführte Algorithmus ist ein simpler aber effektiver Weg Tokens zu finden. Um die Effizienz dieses Algorithmus zu verbessern wurden Verfeinerungsmethoden hinzugefügt.

4.2.1 Token-Cleaning

Durch Parallelisierung können Tokens generiert werden, welche den zuvor gefundenen Tokens ähneln oder sich teilweise auch überlappen. Dies kann potenziell zu Speicherproblemen führen, da mehr Tokens gespeichert werden als notwendig sind. Zudem kann es die Effektivität des Fuzzers beeinträchtigen, da im Mutator unvollständige Tokens verwendet werden. Um dem entgegenzuwirken, werden die Tokens während des Fuzzing Prozess miteinander verglichen und überlappende Tokens werden entfernt.

4.2.2 Filter

Der Discovery-Algorithmus verändert sequentiell benachbarte Bytes vom original Input. Dies ist ineffizient, wenn veränderte Positionen im mutierten Inputs bereits einem Token zugeordnet werden. Zur Vermeidung doppelter Ausführungen wird deshalb eine Liste bereits besuchten Indizes gespeichert. Wird eine bereits gesehene Position ausgewählt, kann dieser dann ignoriert werden.

4.2.3 Maximale Token-Grösse

Da der Token-Discovery-Algorithmus den Input sequenziell nach links und anschliessend nach rechts verändert und ausführt, um zu prüfen, ob dieser zum Token gehört, können unter Umständen sehr grosse Tokens identifiziert werden. Dies kann dazu führen, dass der Fuzzer erheblich verlangsamt wird. Durch die Einführung einer maximalen Token-Grösse lässt sich der Prozess potenziell beschleunigen, wobei gleichzeitig der Fokus gezielt auf kleinere Tokens gelegt wird.

4.2.4 Token Aging

Der sogenannte Discovery-Algorithmus verfolgt das Ziel, relevante Token im Input auf Grundlage der Coverage Map zu identifizieren. Um ein Token der Länge N zu extrahieren, sind jedoch N Ausführungen des Zielprogramms erforderlich. Dieser Prozess ist äusserst rechenintensiv und führt zu einer erheblichen Verlangsamung des gesamten Fuzzing-Vorgangs. Darüber hinaus besteht die Gefahr, dass der Algorithmus ineffiziente Token – beispielsweise sehr kurze oder übermässig lange – extrahiert, was die Effektivität des Fuzzers negativ beeinflussen kann.

Zur Steigerung der Effizienz und Effektivität wurde daher ein einfaches Token Aging-Verfahren implementiert. Der Fuzzer operiert in zwei Phasen: In der ersten Phase identifiziert er potenziell relevante Token und speichert diese zur späteren Wiederverwendung. Nach etwa 1000 Iterationen werden die gespeicherten Token verworfen, und der Erkennungsprozess beginnt erneut.

Wurden jedoch bereits 100 Token vor Ablauf der 1000 Iterationen gesammelt, wechselt der Fuzzer vorzeitig in die zweite Phase. In dieser Phase nutzt er die gesammelten Token zur Generierung neuer Eingaben. Während dieser Phase wird der Discovery-Algorithmus deaktiviert, bis der nächste Token-Erkennungszyklus beginnt.

5 Implementierung

Dieses Kapitel erläutert die Implementierung der in Kapitel 4 vorgestellten Konzepte anhand von Pseudo-Algorithmen und der Nutzung von LibAFL-Werkzeugen. Ziel ist, die praktische Umsetzung verständlich darzustellen.

5.1 Token-Discovery Algorithmus

Beim Token-Discovery Algorithmus wird zunächst die originale Coverage Map P^O an Position i sowie die linken und rechten Nachbarindizes l und r festgelegt. In zwei Schleifen werden zuerst die linken, dann die rechten Nachbarn überprüft, wobei der originale Input geklont wird.

In jeder Iteration wird das mutierte Byte B^M eingesetzt und die Coverage Map neu berechnet. Bleibt sie unverändert, verschiebt sich der Zeiger weiter. Verändert sich die Map ist die Token-Grenze erreicht und die andere Seite wird geprüft. Der Wert an Position i wird wegen der Coverage Map nach jeder Iteration auf den Originalwert zurückgesetzt. Nach Erreichen beider Grenzen wird die Byte-Sequenz aus dem Originalinput extrahiert. Der Pseudocode ist in Algorithmus 1 dargestellt.

Algorithm 1 Token-Discovery Algorithmus

```
1:  function SearchToken(original, mutated)
2:      diffIndices ← ExtractDiffPositions(original, mutated)
3:      input ← Clone
4:      for each i in diffIndices do
5:          B^M ← mutated[i]
6:          l ← i − 1
7:          r ← i + 1
8:          input[i] ← B^M
9:          P^O ← GetCoverage(input)
10:         while l > 0 do
11:             input[l + 1] ← original[l + 1]
12:             input[l] ← B^M
13:             P^N ← GetCoverage(input)
14:             if P^O != P^N then
15:                 break
16:             end if
17:             l ← l − 1
18:         end while
19:         while r < Length(original) do
20:             input[r − 1] ← original[r − 1]
21:             input[r] ← B_M
22:             P^N ← GetCoverage(input)
23:             if P^O != P^N then
24:                 break
25:             end if
26:             r ← r + 1
27:         end while
28:         S.Tokens.Add(original[l..r])
29:     end for
30: end function
```

5.2 Verfeinerungsmethoden

Zur Verbesserung der Token-Qualität wurden ergänzende Methoden implementiert. Die folgenden Abschnitte beschreiben diese Verfeinerungen im Detail.

5.2.1 Token-Cleaning

Diese Implementierung benutzt ein Sliding-Window Prinzip, welches jeden Token miteinander vergleicht ob dieser in einem anderen Token vorkommt. Falls dies der Fall ist wird dieser aus der Token Liste entfernt. Das Sliding-Window ermöglicht hier, dass verschieden grosse Token miteinander verglichen werden können. Da dieser Vorgang, vor allem bei einer grossen Anzahl von Tokens, zeitintensiv ist, wird die Token-Cleaning Methode nur nach einer bestimmten Anzahl von Executions ausgeführt.

5.2.2 Filter

Der Filter wurde so implementiert, dass die Indexes veränderten Bytes in einer Liste abgespeichert werden. Zusätzlich alle Indexes welche während dem Token-Discovery Algorithmus durchgegangen werden, werden auch in dieser Liste abgespeichert. Falls bei späteren Iterationen der gleiche Index aufgerufen wird, können diese ignoriert werden.

5.2.3 Maximale Token-Grösse

Der erste Algorithmus 2 ist eine statische maximale Grösse, welche für die linke und die rechte Seite die gleiche Distanz zur Mitte hat.

Algorithm 2 statische Maximale Grösse

```
function searchToken(original, mutated)
    ...
    l ← i − 1
    r ← i + 1
    ...
    while l > 0  &  i − l + 1 < Size/2 do
        ...
    end while
    while r < C.len  &  r − i − 1 < Size/2 do
        ...
    end while
    ...
end function
```

Zu diesem Zweck wird eine zusätzliche Variable mit dem Namen Size definiert, welche die maximal zulässige Grösse eines Tokens vorgibt. Sowohl in der linken als auch in der rechten Schleife wird überprüft, ob der aktuelle Index eine maximale Distanz von $\frac{\text{Size}}{2}$ nicht überschreitet. Auf diese Weise kann sichergestellt werden, dass die definierte maximale Token-Grösse nicht überschritten wird.

5.2.4 Token Aging

LibAFL stellt flexible Schnittstellen zur Verfügung, über die sich eigene Verfahren nahtlos integrieren lassen. In den sogenannten Stages erhält man Zugriff auf verschiedene Analysewerkzeuge, darunter die Coverage Map sowie den Mutator. In unserer eigens entwickelten Stage werden die Eingaben zunächst mutiert; interessante Ergebnisse – also solche, die neue Pfade oder Verhalten im Zielprogramm auslösen – werden in einer Menge zwischengespeichert.

Auf Basis dieser interessanten Eingaben wird anschliessend der Discovery-Algorithmus angewendet. Zu Beginn des Algorithmus wird überprüft, wie viele Token bereits extrahiert wurden. Überschreitet die Anzahl 100, wird der Algorithmus vorzeitig abgebrochen, um eine Überlastung zu vermeiden.

Darüber hinaus wird in der Stage die Anzahl der bisher ausgeführten Iterationen mitgezählt. Sobald etwa 1000 Iterationen erreicht sind, wird das Token-Set vollständig gelöscht. Dies ermöglicht dem Discovery-Algorithmus, erneut mit der Identifikation relevanter Token zu beginnen.

6 Evaluation und Diskussion

Dieses Kapitel präsentiert die Testergebnisse der implementierten Fuzzer-Lösung. Dabei werden sowohl die Anzahl der durchlaufenen Codepfade als auch die Grösse des generierten Corpus analysiert. Der Discovery-Algorithmus wurde als benutzerdefinierte Stage in LibAFL integriert.

Zur Auswertung der untenstehenden Testfälle wird zunächst die dafür entwickelte Monitoring-Infrastruktur vorgestellt. Anschliessend bewerten wir die Effizienz des Fuzzers anhand definierter Metriken über einen festen Zeitraum. Zur Beurteilung der Effektivität wird eine entsprechende Bewertungsformel eingeführt.

Abschliessend vergleichen wir die entdeckten Tokens mit den von LibAFL bereitgestellten, um zu prüfen, ob ähnliche Tokens identifiziert wurden und welche neuen Muster gegebenenfalls erkannt werden konnten.

6.1 Testfälle

Tabelle 6.1: Vergleichs Testfälle

ID	Name
V1	LibAFL Token-Mutator
V2	LibAFL Byte-Mutator

In der oben stehenden Tabelle 6.1 stehen die Vergleichs Testfälle. Diese verwenden die jeweilige Implementation von LibAFL und dienen als Vergleich zu der neuen Implementierung.

Tabelle 6.2: Ausgeführte Testfälle mit den Verfeinerungs Methoden

ID	Max. Token Grösse	Filtering	Token-Cleaning	Token-Aging
T1	100	X	X	-
T2	20	X	X	-
T3	8	X	X	-
T4	4	X	X	-
T5	best	X	-	-
T6	best	-	X	-
T7	best	-	-	X
T8	best	X	X	X

Wie Tabelle 6.2 zeigt, werden als erstes die Testfälle T1-T4 mit den Verfeinerungsmethoden Filtering und Token-Cleaning durchgeführt. Diese Tests wurden zur Identifikation der optimalen Token Grösse genutzt. Dabei wurden die Anzahl gefundener Corpora und die Code Coverage als Metriken berücksichtigt. Die weiteren Testfälle T5-T8 wurden dann mit der optimalen Token Grösse weiter durchgeführt. Die Testfälle T5-T8 dienen dazu, die Effektivität der Verfeinerungsmethoden zu analysieren. Die Zweiteilung der Testfälle wurde beschlossen, da nur eine begrenzte Anzahl von VMs (Virtual Machines) zur Verfügung standen.

6.2 Monitoring Infrastruktur

LibAFL stellt einen Prometheus-Monitor zur Verfügung, über den Laufzeitstatistiken erfasst werden können. Mithilfe dieser Prometheus-Schnittstelle sowie dem Monitoring-Tool Grafana konnten wir umfassende Dashboards zur Visualisierung der Fuzzer-Statistiken erstellen.

Sowohl die Prometheus-Schnittstelle als auch das Grafana-Dashboard wurden auf Basis von Docker-Images implementiert. Die gesamte Konfiguration erfolgt über eine docker-compose.yml-Datei. Beide Services laufen als eigenständige Microservices in separaten Containern.

Zur Gewährleistung einer konsistenten und vergleichbaren Datenerhebung wurde eine dedizierte Maschine bereitgestellt, die ausschliesslich für das Monitoring und die Darstellung der Fuzzer-Statistiken zuständig ist.

6.3 Resultate Libpng Fuzzer

Die Testfälle T1–T4 wurden jeweils über einen Zeitraum von sieben Tagen ausgeführt. Diese Laufzeit wurde gewählt, um den Einfluss zufälliger Schwankungen zu minimieren. Solche Zufälligkeiten entstehen während des Fuzzing-Prozesses, da der Mutator des Fuzzers zufällige Änderungen an den Eingaben vornimmt.

Unter den getesteten Einstellungen erzielte eine maximale Token-Grösse von 4 die beste Code Coverage und generierte während des Testzeitraums die meisten Corpora. Daher wurde diese Einstellung auch für die nachfolgenden Testfälle T5–T8 übernommen. Abbildung 6.1 stellt die Ergebnisse grafisch dar.

Abbildung 6.1: Code Coverage und Corpora von Testfällen T1-T4 und den Benchmarks V1, V2.

Die Testfälle T5 bis T8 wurden jeweils über einen Zeitraum von sechs Tagen ausgeführt. In allen Fällen kam eine maximale Token-Grösse von 4 zum Einsatz. Abbildung 6.2 veranschaulicht die Ergebnisse dieser Tests.

Abbildung 6.2: Code Coverage und Corpora von Testfällen T5-T8 und die Benchmarks V1, V2.

Die Konfiguration best_configuration (T8) erzielte mit einer Code Coverage von 18% das beste Ergebnis und übertraf im zweiten Testdurchlauf sogar die LibAFL-Variante ohne Tokens (V2). Zudem zeigt Abbildung 6.2, dass best_configuration hinsichtlich der Anzahl generierter Corpora den zweiten Platz belegte.

Es ist jedoch zu betonen, dass die Ergebnisse mit einer gewissen Vorsicht zu interpretieren sind. Aufgrund der inhärenten Zufälligkeit des Fuzzing-Prozesses kann es auch bei identischer Konfiguration und Testdauer zu signifikanten Unterschieden in den Resultaten kommen. Dies wird exemplarisch deutlich beim Vergleich der jeweiligen Ergebnisse von V1 und V2 in den ersten beiden Durchläufen, welche teils stark voneinander abweichen.

Unser Fuzzer erzielt mit sämtlichen implementierten Verfeinerungsmethoden eine Performanz, die mit dem des Benchmarks mit vorgegebenen Tokens entspricht. Die automatisierte Token Extraktion bietet darüber hinaus die Möglichkeit, bei längerer Ausführung eine tiefere strukturelle Analyse des Zielprogramms zu ermöglichen.

Da sowohl die Code Coverage als auch die Anzahl interessanter Corpora zu einem gewissen Grad zufallsbedingt sind, wurde zusätzlich die Effektivität der Implementierungen analysiert. Die **Effektivität pro Sekunde** (Gleichung 6.1) misst, wie viele interessante Inputs c pro Sekunde zum Corpus hinzugefügt werden. Um Unterschiede in den Startzeiten der virtuellen Maschinen zu berücksichtigen, wurde der Kehrwert der durchschnittlichen Executions pro Sekunde $exec/s$ mit der Gesamtzahl an Executions e multipliziert.

Darüber hinaus wurde die **Execution-Aufwand pro Fund** untersucht (Gleichung 6.2), die angibt, wie viele Executions erforderlich sind, um einen interessanten Input zum Corpus hinzuzufügen.

$$\text{Effektivität pro Sekunde} = NC/s = \frac{c}{\left(\text{avg}\left(\frac{exec}{s}\right)^{-1} \cdot e\right)} \tag{6.1}$$

$$\text{Execution-Aufwand pro Fund} = \frac{e}{NC} \tag{6.2}$$

Tabelle 6.3: Resultate der Effektivität

Testfall	NC/s	Exec/NC	Edge%
T1	0.00710154	665762	15.7
T2	0.00714611	903336	15.7
T3	0.00644553	1055357	15.7
T4	0.00694653	1706563	17.1
T5	0.0067	9298038	17.1
T6	0.0055	1123013	15.8
T7	0.0067	11337659	17.3
T8	0.0069	12641515	17.6
V1	0.0077	9755037	17.6
V2	0.0065	11640996	17.0

In Tabelle 6.3 sind die Resultate der beiden untersuchten Metriken dargestellt. Bei der Effektivität pro Sekunde (Gleichung 6.1) gilt: Je höher der Wert, desto besser. Daraus lässt sich erkennen, dass die Token-Implementation von LibAFL (V1) in dieser Hinsicht effektiver ist als die Byte-Implementation (V2).

Die zweite Metrik, der Execution-Aufwand pro Fund (Gleichung 6.2), gibt an, wie viele Executions im Durchschnitt erforderlich sind, um einen interessanten Input zum Corpus hinzuzufügen. Hier gilt: Ein kleinerer Wert ist besser. Auch bei dieser Metrik zeigt V1 eine bessere Performance als V2.

Ein detaillierter Blick auf die Testfälle T5 bis T8 zeigt, dass T8 zwar die höchste Effektivität pro Sekunde erreicht, jedoch den höchsten Execution-Aufwand pro Fund aufweist. Dies lässt sich durch die insgesamt höhere Anzahl an Executions erklären.

Interessant ist zudem der Vergleich zwischen den Testfällen T1, T2 und T4: Obwohl T1 und T2 eine geringere Code Coverage aufweisen als T4, erzielen sie sowohl bei der Effektivität pro Sekunde als auch beim Execution-Aufwand pro Fund bessere Werte.

6.4 Resultate LibMozjpeg Fuzzer

Die Bildverarbeitungsbibliothek Libpng diente massgeblich zur Entwicklung und Evaluierung des vorgestellten Algorithmus. Obwohl der Discovery-Algorithmus anhand von PNG-Bildern getestet wurde, ist er nicht auf dieses Dateiformat beschränkt. Vielmehr ist er so konzipiert, dass er generisch auf beliebige Programme und Dateiformate anwendbar ist – insbesondere solche, die strukturierte Binärdaten verarbeiten. Ziel ist es, Strukturen direkt in Rohdaten (Bytefolgen) zu erkennen, ohne Vorwissen über das konkrete Eingabeformat zu benötigen.

Um die Generalisierbarkeit des Algorithmus weiter zu untersuchen, wurde zusätzlich eine zweite Bildverarbeitungsbibliothek herangezogen, die mit JPEG-Dateien arbeitet. JPEG-Bilder unterscheiden sich strukturell deutlich von PNG-Dateien, insbesondere im Hinblick auf Kompression und Segmentierung. Auch für dieses Format stellt LibAFL geeignete Beispiel-Fuzzer bereit, die als Benchmark zur Evaluierung des Algorithmus dienen.

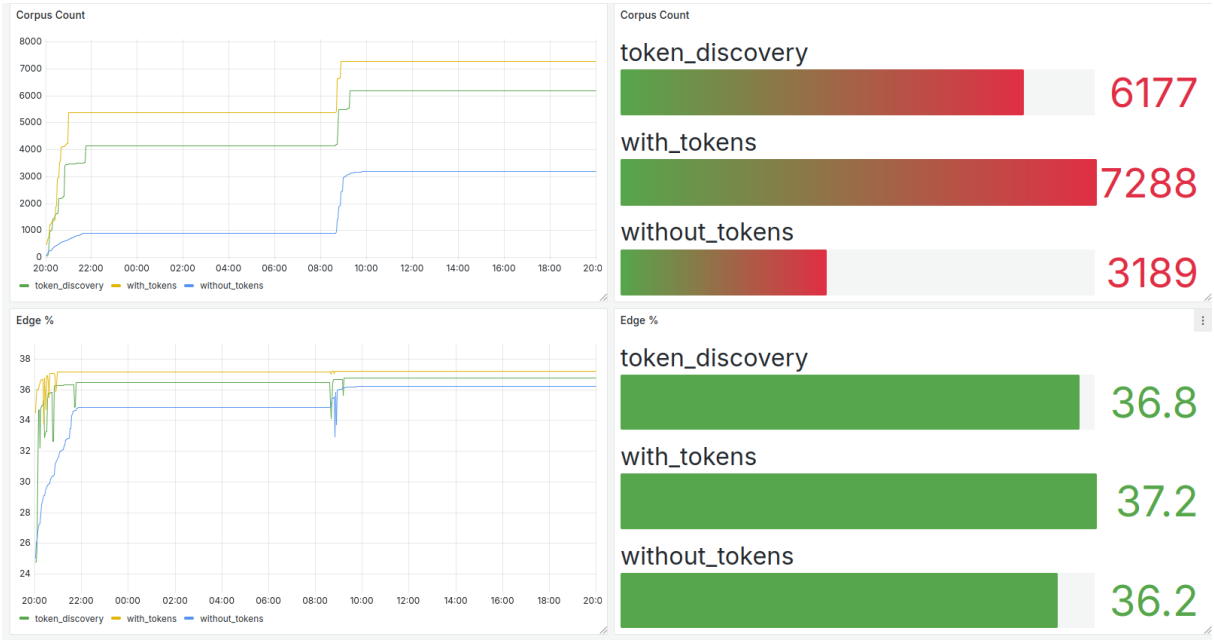

Abbildung 6.3: Code Coverage und Corpus Count des Fuzzes und der gegebenen Benchmarks

Die Tests wurden jeweils über einen Zeitraum von 24 Stunden durchgeführt. Abbildung 6.3 zeigt, dass unser Fuzzer besser performt als der gegebene Benchmark-Fuzzer ohne Tokens. Innerhalb des Testzeitraums konnte unser Ansatz sowohl mehr Corpora erzeugen als auch eine höhere Code Coverage erzielen. Im Vergleich zur Libpng-Testreihe konnten mit mozjpeg zusätzlich sogenannte Objectives gefunden werden. Abbildung 6.4 zeigt die Anzahl der gefundenen Objectives pro Fuzzer.

Das Fuzzing der Libpng-Bibliothek führte hingegen zu keinem einzigen gefundenen Objective, weshalb diese Resultate in der entsprechenden Diskussion nicht weiter berücksichtigt wurden. Objectives bezeichnen in diesem Kontext Eingaben, die zu einem Programmabsturz (Crash) oder einem Timeout führen und somit Hinweise auf potenzielle Schwachstellen oder Fehler im Zielprogramm liefern.

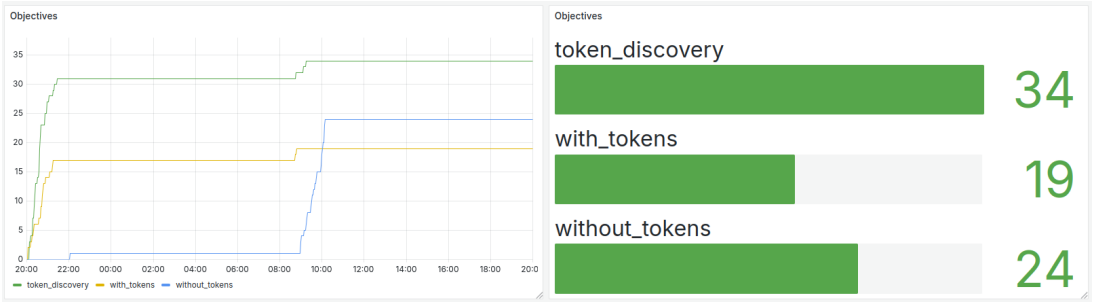

Abbildung 6.4: Anzahl gefundene Objectives

Auffällig ist, dass die Code Coverage in einigen Fällen abnimmt. Dies ist darauf zurückzuführen, dass der Fuzzer während des Tests abgestürzt ist und keinen neuen Client-Prozess zur Fortsetzung des Fuzzings erzeugt hat. Abschliessend lässt sich festhalten, dass unser Fuzzer zwar insgesamt weniger Corpora generiert hat als der Vergleichsbenchmark mit Tokens, jedoch mehr Objectives entdecken konnte – und damit potenziell kritischere Schwachstellen aufgedeckt wurden als durch beide Referenz-Fuzzer.

6.5 Vergleich gefundener und vorgegebener Tokens von Libpng

Zur Bewertung der Fähigkeit des Algorithmus, gültige Tokens innerhalb von PNG-Dateien zu erkennen, wurden alle gefundenen Tokens analysiert. Der Fuzzer lief dafür 15 Minuten ohne vordefinierte Begrenzung der Token Grösse. In dieser Zeit wurden die folgenden zentralen Chunk-Typen (Tokens) identifiziert:

- **IHDR**, **PLTE**, **IDAT**, **IEND**

Diese Chunk-Typen gehören zu den sogenannten *critical chunks*, die erforderlich sind, um eine PNG-Datei korrekt zu dekodieren. Ein valider PNG-Datenstrom beginnt zwingend mit dem IHDR-Chunk, enthält eine oder mehrere IDAT-Chunks und endet mit dem IEND-Chunk. Zusätzlich wurden auch sogenannte *ancillary chunks* erkannt, welche optionale Metadaten enthalten (z.B. Zeitstempel oder Softwareinformationen). Beispiele dieser metadaten Tokens sind:

- `Adobe ImageReady`
- `2015-02-11T03:00:09+01:00`
- `tdate:create`, `tdate:modify`, `tSoftware`

Metadaten wie Erstellungszeit oder verwendete Software werden typischerweise in Text-Chunks wie `tEXt`, `iTXt` oder `zTXt` gespeichert – Schlüsselwörter wie `Creation Time` oder `Software` gehören zu den vordefinierten Keywords :contentReference[oaicite:1]index=1.

Diese Tokens wurden sowohl als einzelne, abgegrenzte Tokens erkannt als auch als Bestandteile umfangreicher Tokenstrukturen, teilweise grösser als 100 Bytes. Dies deutet darauf hin, dass der Fuzzer nicht nur einfache, sondern auch komplex strukturierte Chunk-Daten erfolgreich identifiziert [17], [18], [19].

Zusammenfassung

Erkannte Token-Kategorie	Funktion/Aussagekraft
Critical Chunks (IHDR, PLTE, IDAT, IEND)	Essentiell für korrektes Decodieren
Ancillary/Text-Chunks	Metadaten (Datum, Software, etc.)
Token-Grösse (>100 Bytes)	Komplexe Tokenstrukturen

6.6 Diskussion & und weitere Schritte

Tokens: Die Implementierung erkannte erfolgreich sowohl vordefinierte als auch potenziell relevante zusätzliche Tokens.

Maximale Token Grösse: Eine Token Grösse von 4 erweist sich in Bezug auf die Corpus Grösse und die Code Coverage als besonders geeignet für libpng. Möglicherweise hängt dies damit zusammen, dass im PNG-Format die Felder für Chunk-Länge, -Typ und -CRC jeweils eine Länge von 4 Byte aufweisen [20]. Andere Token Grössen erzielten hingegen eine höhere Effektivität pro Sekunde sowie eine grössere Anzahl an Ausführungen. Daher sollten auch diese weiterhin untersucht werden.

Verfeinerungsmethoden: Die Verfeinerungsmethoden zeigen, dass die Effektivität pro Sekunde gesteigert werden kann. Zusätzlich verschlechterten sich aber die Execution-Aufwand pro Fund, also müssten sich die Implementierung neu überdacht werden.

Dynamische Token-Grösse: Anstelle fester Token-Grössen kann eine dynamische Zuweisung verwendet werden. Wird beispielsweise eine Schleife vorzeitig beendet, sollten die verbleibenden Ressourcen – etwa freie Positionen – der anderen Schleife zur Verfügung gestellt werden. Auf diese Weise lassen sich auch grössere Token vollständig erkennen, selbst wenn ein frühzeitiger Abbruch auf einer Seite der Schleife erfolgt.

7 Fazit

Wir haben einen Token-Level-Fuzzer vorgestellt, der mithilfe unseres Token-Discovery Algorithmus automatisch neue Tokens auf Basis von Coverage-Feedback identifizieren und nutzen kann. Die Wirksamkeit wurde exemplarisch an der Bildverarbeitungsbibliothek libpng demonstriert, wobei sowohl vordefinierte Tokens wie IHDR und PLTE als auch neue, nicht-vordefinierte Tokens erkannt wurden. Darüber hinaus zeigte sich die Übertragbarkeit des Ansatzes durch erfolgreiche Tests an einer weiteren Bibliothek, libmozjpeg. Obwohl die Optimierung der Effizienz nicht im Vordergrund stand, übertraf unsere Implementierung den Byte-Level-Fuzzer hinsichtlich Code Coverage und Corpus-Grösse.

Quellenverzeichnis

[1] C. Intelligence, *Illustration zum Fuzzing in C++ (aus „SecureCodingC++ using Fuzzing“)*, https://www.code-intelligence.com/blog/secure-coding-cpp-using-fuzzing, Zugriff am 09. Juli 2025, 2025.

[2] B. P. Miller, L. Fredriksen und B. So, „An Empirical Study of the Reliability of UNIX Utilities“, University of Wisconsin-Madison, Computer Sciences Department, Techn. Ber. CS-TR-736, 1988, Accessed: 2025-07-10. Adresse: https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf.

[3] B. P. Miller, G. Cooksey und F. Moore, „Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services“, University of Wisconsin-Madison, Computer Sciences Department, Techn. Ber. CS-TR-1268, 1995, Accessed: 2025-07-10. Adresse: https://www.paradyn.org/papers/fuzz.pdf.

[4] M. Zalewski, *American Fuzz Lop*, https://github.com/google/AFL, Accessed: 2025-07-14, 2020.

[5] M. Wu u. a., „One fuzzing strategy to rule them all“, in *Proceedings of the International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, USA: Association for Computing Machinery, Apr. 2022, S. 1634–1645. DOI: 10.1145/3510003.3510174.

[6] P. Godefroid, A. Kiezun und M. Y. Levin, „Grammar-based whitebox fuzzing“, in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, ACM, 2008, S. 206–215. DOI: 10.1145/1375581.1375607.

[7] C. Salls, C. Jindal, J. Corina, C. Kruegel und G. Vigna, „Token-Level Fuzzing“, in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, S. 2795–2809, ISBN: 978-1-939133-24-3. Adresse: https://www.usenix.org/conference/usenixsecurity21/presentation/salls.

[8] Code Intelligence. „The Magic Behind Feedback-Based Fuzzing“. Accessed: 2025-08-04, Code Intelligence. Adresse: https://www.code-intelligence.com/blog/the-magic-behind-feedback-based-fuzzing.

[9] AFL++ Team. „Fuzzing in Depth“. Accessed: 2025-08-04, AFL++. Adresse: https://aflplus.plus/docs/fuzzing_in_depth/.

[10] LLVM Project. „SanitizerCoverage — Clang 18 documentation“. Zugriff am 4. August 2025. Adresse: https://clang.llvm.org/docs/SanitizerCoverage.html.

[11] A. Brown und G. Wilson. „The Architecture of Open Source Applications: Volume 1 – LLVM“. Zugriff am 4. August 2025. Adresse: https://aosabook.org/en/v1/llvm.html.

[12] GeeksforGeeks. „Basic Blocks in Compiler Design“. Zuletzt aktualisiert am 12. Juli 2025, abgerufen am 5. August 2025, GeeksforGeeks. Adresse: https://www.geeksforgeeks.org/compiler-design/basic-blocks-in-compiler-design/.

[13] GeeksforGeeks. „Three-Address Code - Compiler Design“. Zuletzt abgerufen am 5. August 2025, GeeksforGeeks. Adresse: https://www.geeksforgeeks.org/compiler-design/three-address-code-compiler/.

[14] GNU Compiler Collection. „Edges — GCC Internals“. Zuletzt abgerufen am 5. August 2025, Free Software Foundation. Adresse: https://gcc.gnu.org/onlinedocs/gccint/Edges.html.

[15] A. Rehn. „Matching source-level CFG basic blocks to LLVM IR basic blocks“. Bildquelle, zuletzt abgerufen am 5. August 2025, adamrehn.com. Adresse: https://adamrehn.com/articles/matching-cfg-blocks-to-basic-blocks/.

[16] A. Fioraldi, D. Maier, D. Zhang und D. Balzarotti, „LibAFL: A Framework to Build Modular and Reusable Fuzzers“, in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, USA: ACM, 2022. DOI: 10.1145/3548606.3560552. Adresse: https://www.s3.eurecom.fr/docs/ccs22_fioraldi.pdf.

[17] W3C, *Portable Network Graphics (PNG) Specification, Third Edition*, https://www.w3.org/TR/png/, Kapitel zu Critical Chunks: IHDR, IDAT, IEND, 2022.

[18] W3C, *Portable Network Graphics (PNG) Specification, Third Edition*, https://www.w3.org/TR/png/, Kapitel zur Definition von Critical und Ancillary Chunks, 2022.

[19] W3C, *Portable Network Graphics (PNG) Specification (Overview)*, https://www.w3.org/TR/png/Overview.html, Kapitel zu vordefinierten Keywords in Text-Chunks, 2022.

[20] „PNG Specification: File Structure“, besucht am 12. Aug. 2025. Adresse: https://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html.

Ehrlichkeitserklärung

Ich (wir) erkläre(n) hiermit, dass ich (wir) den vorliegenden Leistungsnachweis selber und selbständig verfasst habe(n),

- dass ich (wir) sämtliche nicht von mir (uns) selber stammenden Textstellen und anderen Quellen wie Bilder etc. gemäss gängigen wissenschaftlichen Zitierregeln[1] korrekt zitiert und die verwendeten Quellen klar sichtbar ausgewiesen habe(n);
- dass ich (wir) in einer Fussnote oder einem Hilfsmittelverzeichnis alle verwendeten Hilfsmittel (KI-Assistenzsysteme wie Chatbots[2], Übersetzungs-[3] Paraphrasier-[4] oder Programmierapplikationen[5]) deklariert und ihre Verwendung bei den entsprechenden Textstellen angegeben habe(n);
- dass ich (wir) sämtliche immateriellen Rechte an von mir (uns) allfällig verwendeten Materialien wie Bilder oder Grafiken erworben habe(n) oder dass diese Materialien von mir (uns) selbst erstellt wurde(n);
- dass das Thema, die Arbeit oder Teile davon nicht bei einem Leistungsnachweis eines anderen Moduls verwendet wurden, sofern dies nicht ausdrücklich mit der Dozentin oder dem Dozenten im Voraus vereinbart wurde und in der Arbeit ausgewiesen wird;
- dass ich mir (wir uns) bewusst bin (sind), dass meine (unsere) Arbeit auf Plagiate und auf Drittautorschaft menschlichen oder technischen Ursprungs (Künstliche Intelligenz) überprüft werden kann;
- dass ich mir (wir uns) bewusst bin (sind), dass die Hochschule für Technik FHNW einen Verstoss gegen diese Eigenständigkeitserklärung bzw. die ihr zugrundeliegenden Studierendenpflichten der Studien- und Prüfungsordnung der Hochschule für Technik verfolgt und dass daraus disziplinarische Folgen (Verweis oder Ausschluss aus dem Studiengang) resultieren können.

Windisch, 14. August 2025

Name: Alessandro Lenti

Unterschrift: 

Name: Nguyen Hoang Viet

Unterschrift: 

[1] IEEE
[2] ChatGPT
[3] Deepl
[4] Quillbot
[5] Github Copilot