Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Analyse von nicht-funktionalem Programmverhalten mit evolutionärem Fuzzing

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Gergö Miklovics
geboren am: 31.07.1994
geboren in: Pécs, Ungarn

Gutachter/innen: Prof. Lars Grunske
Dr. Thomas Vogel

eingereicht am: ..... verteidigt am: .....

Inhaltsverzeichnis

1 Abstract

Leistungsprobleme können in Software unerwartet auftreten, wenn Programme mit Inputs versorgt werden, die ein Worst-Case-Verhalten aufweisen. Ein großer Teil der Forschung konzentriert sich auf die Diagnose solcher Probleme mittels statistischer Profiling-Techniken [9]. Aber wie erkennt man diese Inputs überhaupt?

Solche Inputs können in der SUT (**S**oftware **U**nder **T**est) komplett problemlos durchlaufen und man würde erkennen, dass sie eine Menge Ressourcen verbraucht haben. Andererseits wäre es auch problematisch, wenn man einen Input nutzt, der eigentlich nicht viel Speicher braucht, aber dafür sehr lange berechnet werden muss. Bei großen Programmen muss auch auf den Ressourcenverbrauch geachtet werden, insbesondere wenn man zum Beispiel tausende von Inputs pro Sekunde bearbeiten möchte. Manche würden davon aber einen Großteil unserer Ressourcen blocken und/oder so lange mit der Berechnung brauchen, dass man erst einmal Mid-Production stehen bleiben muss. Dieser Fall darf nicht eintreten.

In dieser Arbeit werde ich Fuzzing und insbesondere das bereits bestehende EvoGFuzz (***Evo***lutionary ***G***rammar-Based ***Fuzz***ing) [4] vorstellen und dafür weitere sogenannte Fitnessfunktionen entwickeln.

Im Rahmen dieser Arbeit wurden Funktionen entwickelt, um potenzielle Schwachstellen in Bezug auf Speicherverbrauch und Zeitverbrauch aufzudecken. Zusätzlich wurden Fitnessfunktionen für Codeabdeckung, spezifische Exceptions und allgemeine Abstürze entwickelt.

Es ist jedoch wichtig klarzustellen, dass im Rahmen der Testfälle ein Programm entwickelt wurde, bei dem bestimmte Inputs spezifische Ergebnisse auslösen können. Dies ermöglicht eine gezieltere Prüfung der Funktionen.

Die Ergebnisse der Testläufe zeigen einen vielversprechenden Trend an. Im Vergleich zum Baseline-EvoGFuzz wurden mit dem erweiterten EvoGFuzz statistisch wesentlich bessere Ergebnisse erreicht. Verglichen mit Performance Fuzzing hat der erweiterte EvoGFuzz signifikante Verbesserungen bei der Aufdeckung von hoher CPU-Zeit- und hohem Speicherverbrauch erzielt. Es wurde jedoch bei der Wanduhrzeit kein bedeutsamer Fortschritt erreicht. Des Weiteren wurde mein Ansatz zwar auf einem synthetischen Beispiel ausgewertet, aber die Ergebnisse könnten womöglich bei echten Programmen ganz anders aussehen.

2 Einführung

Sicherheitslücken können in Software oft extrem kostspielig sein, aber nicht nur wegen des direkten Geschäftsverlusts oder der Nichtverfügbarkeit der verletzten Ressourcen. Die permanente Schädigung oder auch langfristige Kosten durch Sicherheitslücken haben auch weitreichendere Auswirkungen auf den künftigen Umsatz eines betroffenen Unternehmens. Diese Kosten hängen mit dem Verlust von Kunden zusammen, die zur Konkurrenz abwandern, oder mit der Unfähigkeit, neue Kunden zu gewinnen, weil die Sicherheit als zu schlecht wahrgenommen wird [2] .

Deswegen müssen solche Sicherheitslücken in einer sicheren Umgebung getestet und aufgedeckt werden, bevor sie in der 'realen Welt' Schaden anrichten können. Dafür gibt es bereits seit Jahren verschiedene Arten von Tests und Testumgebungen.

Eines davon ist Whitebox-Testing. Es bezieht sich auf eine Testmethode, bei der der Tester über detaillierte Kenntnisse über die interne Struktur und Implementierung des zu testenden Systems verfügt. Beim Whitebox-Testing werden Tests basierend auf der internen Logik, dem Datenfluss, den Kontrollstrukturen und den Rahmenbedingungen des Codes erstellt [8].

Ferner unterscheiden wir noch das Graybox-Testing, bei dem der Tester begrenztes Wissen über die interne Struktur des Systems hat, typischerweise eine Teilmenge der Informationen, die ein Whitebox-Tester hat [8].

Falls dem Tester der Zugriff auf den Quellcode der Software nicht gegeben ist, ist Whitebox- oder Graybox-Testing nicht möglich. Dies kann bei proprietären Softwaren der Fall sein, deren Quellcode nicht öffentlich verfügbar ist, oder bei Sicherheitsbeschränkungen, die den Zugriff auf den Quellcode aus sensiblen Gründen verbieten. Es kann auch vorkommen, dass der Quellcode bei älteren Systemen nicht mehr verfügbar ist und somit die interne Implementierung nicht bekannt ist. In solchen Szenarien wird es schwierig oder unmöglich, diese Testingmethoden anzuwenden.

Und aus diesem Grund existiert das Blackbox-Testing, bei dem keine Kenntnisse über die interne Funktionsweise der Anwendung vorhanden sind [8] oder benötigt werden. Es werden lediglich grundlegende Aspekte des Systems von außen untersucht, und es besteht keine oder nur geringe Verbindung zur internen logischen Struktur des Systems [8]. Zu diesem gehört das so genannte Blackbox-Fuzzing, das sich bei der Aufdeckung kritischer Schwachstellen bewährt hat, die auf andere Weise nicht geprüft werden können [15].

Fuzzing ist ein Ansatz zum Softwaretesten, bei dem das zu testende System mit Testfällen bombardiert wird, die von einem anderen Programm generiert wurden. Das System wird überwacht, um Fehlverhalten festzustellen, welches durch die Verarbeitung dieser Inputs aufgedeckt wird [10]. Auch wenn ein solch simpler Ansatz naiv klingen

mag, hat sich in der Vergangenheit gezeigt, dass Fuzzing effektiv bei der Aufdeckung von Fehlern in einer Vielzahl von Softwaresystemen ist [10].

Während sich die Grundprinzipien des Fuzzings seit der ersten Prägung des Begriffs [12] nicht geändert haben, hat die Komplexität der Mechanismen, die zum Antreiben des Fuzzing-Prozesses verwendet werden, erhebliche evolutionäre Fortschritte gemacht [10].

Bestimmte Inputs können auch zu einer sicherheitskritischen Schwachstelle werden, wenn Angreifer die Kontrolle über diese Schwachstellen übernehmen, um eine große Menge an Speicher zu verbrauchen und einen Denial-of-Service-Angriff (DoS) zu starten [9]. Die Erkennung solcher Schwachstellen ist jedoch eine Herausforderung, da sich die gängigen Fuzzing-Techniken bisher eher auf die Codeabdeckung, nicht aber auf den Speicherverbrauch konzentrieren [19].

Eine der neuesten Techniken ist EvoGFuzz. Der ***Evo***lutionary ***G***rammar-Based ***Fuzz***er. EvoGFuzz konzentriert sich auf die Generierung von Testinputs zur Aufdeckung von Fehlern und unerwünschtem Verhalten [4]. Ob das Programm abstürzt und bei welchen Inputs es das tut, sind jedoch nicht die einzigen Eigenschaften, die getestet werden sollten, sondern auch, welches unerwünschte Verhalten der Input produziert.

Solches Verhalten können zum Beispiel erhöhter Ressourcenverbrauch (Memory) oder unerwünscht lange Laufzeiten sein. Diese Funktionen befinden sich bisher nicht im EvoGFuzz, daher werde ich diese beiden Eigenschaften in dieser Arbeit untersuchen und Fitnessfunktionen zur Erweiterung von EvoGFuzz entwickeln.

3 Verwandte Arbeiten

Dieser Abschnitt gibt einen Überblick über verwandte Arbeiten und den Stand der Technik in Bezug auf die Ziele dieser Arbeit mit besonderem Schwerpunkt auf derzeit verfügbare Methoden zum Testen und Auffinden von Zeit- und ressourcenbasierten Schwachstellen.

3.1 AFL- American Fuzzy lop

American Fuzzy Lop ist ein sicherheitsorientierter Fuzzer, der eine neue Art der Instrumentalisierung zur Kompilierzeit und genetische Algorithmen einsetzt, um automatisch saubere, interessante Testfälle zu finden, die neue interne Zustände in der Zieldatei auslösen [22].

Dadurch wird die funktionale Abdeckung des gefuzzten Codes erheblich verbessert. Die kompakten und synthetischen Datensätze, die das Tool erzeugt, sind auch für andere, arbeitsintensivere oder ressourcenintensivere Testverfahren von Nutzen [22].

Etwas vereinfacht lässt sich der Algorithmus von AFL folgendermaßen zusammenfassen [1]:

1. Laden der vom Benutzer bereitgestellten anfänglichen Testfälle in die Warteschlange.
2. Entnahme der nächsten Eingabedatei aus der Warteschlange.
3. Versuch, den Testfall auf die kleinste Größe zu trimmen, die das gemessene Verhalten des Programms nicht verändert.
4. Wiederholte Mutation der Datei unter Verwendung einer ausgewogenen und gut untersuchten Auswahl an traditionellen Fuzzing-Strategien, wie zufällige Bit-Flips oder arithmetische Mutationen.
5. Wenn eine der erzeugten Mutationen zu einem neuen, von der Instrumentalisierung aufgezeichneten Zustandsübergang führt, füge den mutierten Output als neuen Eintrag in die Warteschlange ein.
6. Weiter zu 2.

Die entdeckten Testfälle werden regelmäßig überprüft, um diejenigen zu eliminieren, die durch neuere, umfassendere Funde überflüssig geworden sind, und durchlaufen mehrere andere instrumentengesteuerte Schritte zur Aufwandsminimierung [1].

3.2 Performance Fuzzing

Leistungsprobleme können in Software unerwartet auftreten, wenn Programme mit Inputs versorgt werden, die ein Worst-Case-Verhalten aufweisen [9]. Außerdem sind solche Inputs auch sehr schwer zu generieren. Also wie macht man sie am besten ausfindig? Zu den am häufigsten gewählten Quellen gehören:

- manuell geschriebene Inputs
- oft vorkommende Inputs
- standardisierte Testsets
- Inputs, die von den Nutzern eingeschickt wurden, weil diese seltsames Verhalten ausgelöst haben [9]

Diese Quellen von Inputs betonen jedoch entweder nur das Durchschnittsverhalten oder können erst dann entdeckt werden, wenn der Schaden bereits eingetreten ist.

Um dieses Problem zu lösen, wurde PerfFuzz von Lemieux et al. entwickelt, um jede Programmstelle mit einem Input zu verknüpfen, der diese Stelle am meisten beansprucht [9].

Das Ziel von Performance-Fuzzing besteht darin, Leistungsengpässe zu identifizieren, die Ressourcennutzung zu optimieren und sicherzustellen, dass Programme große Datenmengen oder eine hohe Parallelität verarbeiten können, ohne langsamer zu werden oder abzustürzen [9].

3.2.1 Fachbegriffe

Mutation: Bei Fuzzing bezieht sich die Mutation auf den Prozess der Veränderung oder Manipulation von Eingabedaten, um neue Testfälle zu generieren. Dabei werden bestehende Inputs durch zufällige Modifikationen wie das Ändern von Werten, Hinzufügen oder Entfernen von Daten oder das Verzerren von Strukturen verändert [24].

Programmkomponenten: Eine Programmkomponente ist eine unabhängig verwendbare und austauschbare Einheit innerhalb einer Anwendung. Sie stellt eine bestimmte Funktionalität bereit und kann in verschiedenen Kontexten wiederverwendet werden. Eine Komponente kann beispielsweise eine Klasse, ein Modul, eine Bibliothek oder ein Service sein, der spezifische Aufgaben oder Dienste innerhalb des Gesamtsystems erfüllt [16].

Der PerfFuzz-Algorithmus kann leicht angepasst werden, um eine Vielzahl von Werten für verschiedene Programmkomponenten zu maximieren: die Anzahl der bei malloc-Anweisungen zugewiesenen Bytes, die Anzahl der Cache-Misses oder Page Faults bei Speicherlade-/Speicheranweisungen, die Anzahl der E/A-Operationen über Systemkomponenten hinweg, usw.

3.2.2 Die Funktionsweise von PerfFuzz

Der Algorithmus von PerfFuzz basiert auf dem abdeckungsorientierten Greybox-Fuzzer AFL. Der Algorithmus benötigt als Input ein Programm und eine Reihe Start-Inputs.

Der Algorithmus initialisiert zunächst einen Satz von Parent-Inputs basierend auf den Start-Inputs. Jeder Parent-Input wird dann für das Mutations-Fuzzing in Betracht gezogen, und zwar mit einer Wahrscheinlichkeit, die sich nach ihrer aktuellen Günstigkeit richtet. Child-Inputs werden durch Mutation ausgewählter Parent-Inputs mithilfe von implementierungsspezifischen Heuristiken generiert, die von AFL übernommen wurden.

Das Programm wird mit jeder neu generierten Child-Input ausgeführt. Dabei werden Rückmeldungen gesammelt, wie Informationen über die Codeabdeckung und Werte, die mit den ausgewählten Programmkomponenten verbunden sind.

Wenn die Ausführung zu einer neuen Codeabdeckung führt oder den Wert einer Programmkomponente maximiert, wird der entsprechende Input dem zukünftigen Fuzzing als Parent-Input hinzugefügt.

Der Algorithmus iteriert weiter über die Parent-Inputs, bis das angegebene Zeitbudget abgelaufen ist. Dieser Vorgang wird wiederholt, um leistungsstarke Eingaben zu erzeugen [9].

3.3 Zest

Zest ist eine Technik, die entwickelt wurde, um die semantische Analysephase von Testprogrammen zu verbessern [13].

Es basiert auf der Idee, zufällige Inputgeneratoren, ähnlich wie bei QuickCheck [3], automatisch anzupassen, um die Untersuchung der semantischen Analysestufen effizienter zu gestalten.

Zest erkennt, dass die kleinste Einheit, das Bit, eine wichtige Rolle bei der Darstellung und Konstruktion der generierten Inputs spielt. Zest wandelt zuerst QuickCheck-ähnliche Generatoren mit zufälliger Eingabe in deterministische parametrische Generatoren um, die eine Folge von nicht typisierten Bits, die so genannten "Parameters", auf eine syntaktisch gültige Eingabe abbilden. Auf dieser Grundlage wendet Zest einen Algorithmus ähnlich wie bei den CGF-Tools (Coverage Guided Fuzzer [22]), auf die Parameterregion an [13].

Indem Zest Änderungen auf Bit-Ebene identifiziert und darauf reagiert, kann es die Generierung von Testinputs in Richtung semantischer Gültigkeit und erhöhter Codeabdeckung lenken. Zest ist jedoch nicht in der Lage, Leistungsprobleme aufzudecken.

3.4 Bonsai Fuzzing

Bonsai Fuzzing ist eine Technik zur automatischen Erzeugung eines prägnanten und umfassenden Testkorpus für strukturell komplexe Inputs [17].

Bonsai Fuzzing erzeugt kleine Inputs durch einen iterativen evolutionären Algorithmus. In der ersten Runde werden winzige Inputs erstellt, und in jeder nachfolgenden Runde werden Inputs etwas größer, indem bereits generierte Inputs mutiert werden. Dabei werden die Ergebnisse aus vorherigen Runden berücksichtigt.

Es wird zunächst ein Verfahren zur Auswahl syntaktisch gültiger Inputs aus einer Grammatikspezifikation definiert, indem man die Anzahl der Identifikatoren, linearen Wiederholungen und verschachtelten Expansionen in den resultierenden Ableitungsbäumen begrenzt.

Anschließend definiert man eine partielle Ordnung für fuzzerbasierte Grammatiken, die von der Testabdeckung geleitet werden. Für jede gewünschte Größenbeschränkung ergibt sich daraus ein Raster von fuzzerbasierten Grammatiken. Die unterste Ebene

des Rasters hat eine minimale Größenbeschränkung - ein Fuzzer mit wenigen oder keinen guten Start-Inputs - und die oberste Ebene hat die maximal wünschenswerte Größenbeschränkung - ein Fuzzer, der das endgültige Testkorpus erzeugt[17].

3.5 MemLock - Memory usage guided fuzzing

Um Fehler im Speicherverbrauch aufzuspüren, haben Wen et al. mit dem Ziel der Verbesserung von Grey-Box-Fuzzing MemLock entwickelt. MemLock arbeitet in zwei Schritten.

Zuerst führt es eine statische Analyse durch, um die für den Speicherverbrauch relevanten Anweisungen und Operationen zu identifizieren.

Es wird qualitativ der Aufrufgraph analysiert, der den Stack-Speicherverbrauch bestimmt, und quantitativ die Speicherverbrauchsoperationen, die den Heap-Speicherverbrauch bestimmen. Außerdem wird der Kontrollflussgraph des Programms analysiert, der eine Verzweigungsabdeckung für die Erkundung verschiedener Programmpfade bietet. Anhand des analysierten Speicherverbrauchs werden die Informationen zur Verzweigungsabdeckung und zum Speicherverbrauch von MemLock genutzt, um den Fuzzing-Prozess zu steuern [19].

Anschließend steuert MemLock den Fuzzing-Prozess basierend auf den Informationen zur Verzweigungsabdeckung und zum Speicherverbrauch. Wenn ein Input einen neuen Zweig abdeckt oder zu einem höheren Speicherverbrauch führt, wird dieser als interessant angesehen und in die Seed-Warteschlange aufgenommen. Dieser Input wird weiter mutiert, bis MemLock voraussichtlich einen Input erzeugt, bei dem der Speicherverbrauch den verfügbaren Speicher übersteigt.

3.6 Inputs from Hell - Learning Input Distributions for Grammar-Based Test Generation

Zum Fuzzing gehört nicht nur das tatsächliche Testen (SUT mit diversen Inputs durchlaufen lassen), sondern man muss auch bestimmen, wie diese Inputs generiert werden sollen. Eine Möglichkeit davon sind - wie von Soremekun et al. beschrieben - durch probabilistische Grammatiken generierte Inputs.

Ein probabilistischer Input Generator ist ein Ansatz zur Generierung von Testinputs, bei dem Wahrscheinlichkeiten verwendet werden, um die Verteilung der generierten Inputs zu steuern [14]. Anstatt zufällige Inputs zu erzeugen, werden bestimmte Wahrscheinlichkeiten festgelegt, um gezielte Inputs mit spezifischen Eigenschaften zu generieren.

Es gibt drei Haupteingenschaften zu diesen Inputs:

1. Gewöhnliche Inputs: Inputs, die im typischen Betrieb auftreten

2. Ungewöhnliche Inputs: In der Produktion weniger auftretende Inputs, die möglicherweise weniger häufig verwendet, weniger getestet oder weniger verstanden werden.

3. Fehlerverursachende Inputs: Inputs, die gut funktionieren sollen, die früher zu Fehlern geführt haben, bei denen der Fehler bereits behoben ist.

Die Generierung dieser Inputs erfolgt durch:

1. Gewöhnliche Inputs: Um dieses Verhalten abzudecken, gibt es in der Regel eine Reihe spezieller Tests (manuell geschrieben oder generiert).

2. Ungewöhnliche Inputs: die Warscheinlichkeiten aus den gewöhnlichen Inputs werden gelernt, und diese Warscheinlichkeiten werden invertiert, damit man die Ungewöhnlichen Inputs generieren kann.

3. Fehlerverursachende Inputs: es wird von Proben gelernt, die in der Vergangenheit zu Fehlern geführt haben. Dadurch können wir ähnliche Inputs erzeugen, um die Umgebung der ursprünglichen Inputs zu testen [14].

3.7 Evolutionary Grammar-Based Fuzzing

Der Nachteil von PerfFuzz, MemLock und dem Größteil der Coverage-Guided Fuzzers wie AFL ist, dass sie den ursprünglichen Input nur mutieren.

Anders gesagt: Ich nehme als Beispiel eine Software für einen Taschenrechner. Man hat bei der Software diverse Operatoren, wie zum Beispiel der Input für die Wurzel von 2: $sqrt(2)$. Wenn man nur den Input $sqrt(2)$ nehmen würde, könnte man daraus mit den vorhandenen Möglichkeiten von zum Beispiel PerfFuzz, entweder ein Byte ändern: $sXrt(2)$, oder hinzugeben/entfernen: $sXqrt(2)$ / $srt(2)$. Aber wie man sieht, könnte das dann auch nicht valide Inputs erzeugen.

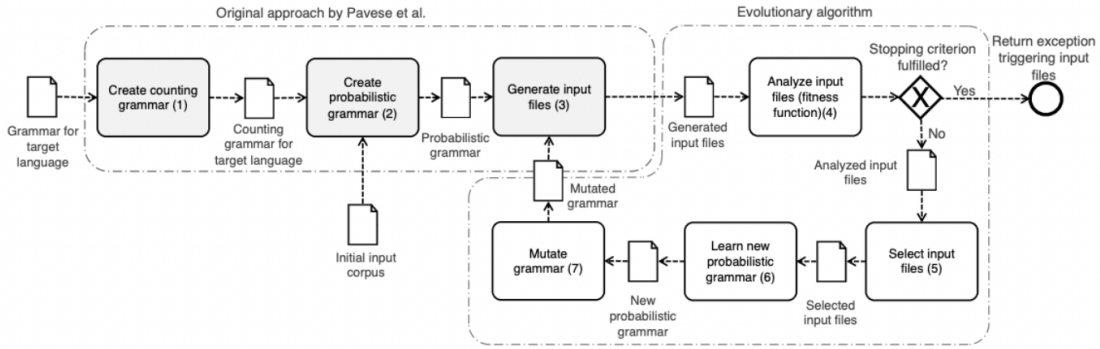


Abbildung 1: Der Aufbau von EvoGFuzz [4]

Das kann man mit EvoGFuzz umgehen. Wie auf Abbildung 1 zu sehen ist, beginnt der von Eberlein et al. entwickelte Fuzzer mit der Definition einer Grammatik, die die Syntax und Struktur gültiger Inputs in einem Programm bestimmt [4]. Diese Grammatik wird dann verwendet, um einen zufälligen, aber syntaktisch korrekten Satz von Inputs zu erzeugen. Der Fuzzer wertet dann das Verhalten des Programms aus, wenn diese Inputs in das Programm eingespeist werden, und protokolliert auftretende Fehler oder Abstürze [4].

Auf der Grundlage dieser Bewertung wählt der Fuzzer die leistungsfähigsten Inputs aus und verwendet sie, um durch das Neulernen der Grammatik eine neue Generation von Inputs zu erzeugen.

3.8 Unterschiede

Die Hauptunterschiede meiner erweiterten EvoGFuzz-Variante im Vergleich zu den oben genannten Fuzzern liegen in verschiedenen Aspekten.

Erstens, wie beim EvoGFuzz bereits erwähnt, arbeiten alle vorgestellten Fuzzer außer Bonsai Fuzzing mithilfe von Mutation, und dadurch können auch nicht valide Inputs erstellt werden. Das ist bei meiner Variante ausgeschlossen. In meiner Variante hingegen werden die Inputs gemäß einer bestimmten Grammatik generiert, die im Vorfeld an die Grammatik des zu testenden Systems angepasst werden.

Des Weiteren besteht bei den genannten Fuzzern die Möglichkeit, dass die gefundenen interessanten Inputs die nachfolgenden Inputs in eine bestimmte Richtung lenken. Das bedeutet, wenn ein Fuzzer einen interessanten Input findet, werden die nächsten Inputs in ähnlicher Weise mutiert, was dazu führen kann, dass nur bestimmte Teile des Systems getestet werden.

Diese problematische Eigenschaft wird bei EvoGFuzz vermieden. Die Inputs werden generiert und anschließend werden die interessanten Inputs analysiert, um eine neue Grammatik abzuleiten. Diese Grammatik kann bei Bedarf mutiert werden, um das sogenannte 'Drifting' zu vermeiden.

Bei einer solchen Drift würde die Grammatik nur Inputs mit spezifischen Merkmalen erzeugen, die den ausgewählten Individuen ähneln, von denen die Grammatik gelernt wurde [4].

Darüber hinaus ist eine der Hauptunterschiede die Suche nach Leistungsproblemen. Diese Eigenschaft ist teilweise bei PerfFuzz und MemLock vorhanden, aber alle anderen vorgestellten Fuzzer suchen nach Inputs mit besserer Codeabdeckung. Das bedeutet, wie zum Beispiel bei den Fuzzern AFL und Bonsai Fuzzing, je mehr Codes abgedeckt werden, desto 'interessanter' ist dieser Input. Bei Memlock ist der Input 'interessant', wenn er mehr Speicher verbraucht hat, und bei PerfFuzz, wenn er mehr CPU-Leistung

gebraucht hat, um diesen Input zu verarbeiten.

Also kurz zusammengefasst: Meine erweiterte EvoGFuzz Variante ist besser im Bezug zu Inputgenerierung, da keine invaliden Inputs generiert werden können, wie bei den mutationsbasierten Fuzzern. Außerdem hat diese die Eigenschaften von MemLock und PerfFuzz, indem sowohl Speicherverbrauch- als auch Leistungsprobleme aufgedeckt werden. Des weiteren besitzt sie die Eigenschaft von Bonsai Fuzzing, weil sie mit einem grammatikbasierten Inputgenerator arbeitet, wie Zest.

4 Herangehensweise

In diesem Abschnitt werde ich meine Herangehensweise zur Erweiterung von einem evolutionären Fuzzer vorstellen. Der Schlüssel dieser Arbeit ist, mehrere Fitnessfunktionen einzubauen, die bisher nicht in dem Fuzzer vorgekommen sind, um diverse Schwachstellen aufzudecken. Dadurch möchte ich erreichen, dass verschiedene Aspekte eines Programms getestet werden können, ohne großen Aufwand mit den Testfällen zu betreiben.

4.1 Übersicht

Die eingebauten Fitnessfunktionen dienen dazu, Schwachstellen zu finden, die in der SUT bereits drin sind, aber noch nicht aufgedeckt wurden.

Aktuell sucht der Fuzzer ausschließlich nach Abstürzen. Das sind aber nicht die einzigen Probleme, die man in einer Software finden kann. Ich möchte deshalb die folgenden neuen Fitnessfunktionen zu diversen Fällen vorstellen.

4.2 Zeitverbrauch

Eine Wartezeit bei der Software ist die Zeitspanne, die die CPU auf eine bestimmte Aktion oder ein bestimmtes Ereignis wartet, bevor sie mit der nächsten Anweisung fortfährt.

Es gibt mehrere Gründe, warum die CPU warten muss. Zum Beispiel könnte sie darauf warten, auf den Direktzugriffsspeicher (RAM) zuzugreifen, um Daten zu lesen oder zu schreiben. Eine andere mögliche Ursache ist, dass auf den Abschluss einer Eingabe-/Ausgabeoperation gewartet wird.

Eine CPU-Wartezeit kann ein Engpass für die Computerleistung sein, da die CPU während dieser Zeit nicht produktiv ist und stattdessen Ressourcen verschwendet. Daher ist es wichtig, die Ursache der CPU-Latenz zu identifizieren und zu minimieren, um die Gesamtleistung des Systems zu verbessern.

4.2.1 Wanduhrzeit

Die **Wanduhrzeit** - auch als 'wall clock time' bezeichnet - bei der Software bezieht sich auf die tatsächliche Zeit, die benötigt wird, um eine bestimmte Aufgabe zu erledigen. Es ist die Zeit, die von einer herkömmlichen Uhr abgelesen werden kann [18].

4.2.2 CPU-Zeit

Die **CPU-Zeit** (ab jetzt CPUT genannt) stellt die Zeit dar, die die CPU mit der Ausführung einer bestimmten Aufgabe verbringt. Diese kann als 'reine' Ausführungszeit bezeichnet werden, da sie andere Faktoren wie Ein- und Ausgabevorgänge, Wartezeiten oder Pausen nicht berücksichtigt [18].

4.3 Speicherverbrauch

Speicherlecks, oder memory leaks treten in der Softwareentwicklung dann auf, wenn ein Programm Speicher allokiert, aber diesen Speicher nach der Nutzung nicht ordnungsgemäß freigibt. Wenn dies über einen längeren Zeitraum geschieht, kann dies dazu führen, dass der verfügbare Speicherplatz knapp wird, und es das Programm schließlich zum Absturz bringt, oder unerwartete Ergebnisse liefert [25].

Bei **unnötiger Speicherzuweisung** wird gegebenenfalls mehr Speicher zugewiesen als notwendig. [25] Anschließend, bei **schlechten Speicherstrategien** wird der vom Programm benötigte Speicher ineffizient benutzt, beispielsweise wenn es häufig Speicher allokiert und freigibt, anstatt einen Pool von Speicherbereichen zu verwenden [25].

Um einen hohen Speicherverbrauch zu reduzieren, ist es wichtig sicherzustellen, dass das Programm den verfügbaren Speicher effizient nutzt und dass Speicherlecks o.ä. vermieden werden.

4.4 Exception tracking

Das Verfolgen von einzelnen Exceptions in der Software ist wichtig, da diese unerwartetes Verhalten darstellen, das durch Codefehler oder fehlerhaftes Verhalten verursacht werden kann. Die Verfolgung von einzelnen Exceptions hilft Entwicklern, diese Fehler zu identifizieren und zu beheben, bevor sie ernsthaftere Probleme verursachen oder die Anwendung zum Absturz bringen.

Mit der Exceptionverfolgung können Entwickler auch bestimmen, wie oft bestimmte Exceptions in der Anwendung auftreten und welche Exceptions am häufigsten vorkommen. Durch Fuzzing ist es ferner auch möglich, dass man herausfindet, welche Inputs diese Exceptions verursachen, und dadurch wird es einfacher zu identifizieren, an welcher Stelle das geschieht.

Diese Informationen können dazu beitragen, die Anwendungsstabilität, Anwendungszuverlässigkeit und die Benutzererfahrung zu verbessern.

4.4.1 Diverse Exceptions

Derzeit ist es möglich, den Fuzzer einzusetzen, um generelle Abstürze, die von diversen Exceptions ausgelöst worden sind, zu verfolgen. Der Fuzzer testet die SUT mit den Inputs durch, und untersucht, ob es einen Absturz gegeben hat. Wenn ja, wird die nächste Grammatik in der Richtung mutiert, dass man mehr, ähnliche Inputs generieren kann, die diesen Absturz auslösen.

4.5 Code Coverage

Die Codeabdeckung ist ein Indikator dafür, wie viel Quellcode während der Ausführung der SUT tatsächlich abgedeckt wurde. Ein wichtiger Faktor beim Fuzzing ist das Messen der Codeabdeckung, um sicherzustellen, dass Testinputs eine ausreichende Anzahl von Codepfaden durchlaufen, um so viele Fehlerquellen wie möglich zu entdecken.

Eine höhere Codeabdeckung bedeutet im Allgemeinen eine höhere Chance, potenzielle Fehler zu finden, und ist daher ein wichtiger Faktor bei der Bewertung der Qualität von Fuzzing.

5 Änderungen bei EvoGFuzz

Im Rahmen meiner Arbeit an EvoGFuzz habe ich einige Änderungen vorgenommen, um die Leistung und Bedienbarkeit des Tools zu verbessern.

Eine wesentliche Verbesserung betrifft die Initialisierung des Fuzzers. Früher musste eine separate Funktion erstellt werden, um den Inputs den Wert 'True' zuzuweisen, wenn ein Absturz auftrat, und 'False', wenn die SUT ohne Probleme ausgeführt wurde. Dazu musste ein Typ von EvoGFuzz importiert und die Funktion zur Wertzuweisung erstellt werden.

Um die Programme benutzerfreundlicher zu gestalten, habe ich mich entschieden, die Fitnesswerte direkt innerhalb von EvoGFuzz zu berechnen. Daher wird die SUT nun bei der Initialisierung direkt übergeben, anstatt eine separate Funktion zu verwenden, und die Fitnesswerte werden direkt in EvoGFuzz berechnet.

Schließlich wurden auch die Fitnessfunktionen für die oben genannten Funktionen implementiert.

6 Implementierung der Fitnessfunktionen

6.1 Zeitverbrauch

Bei der Implementierung der Fitnessfunktion für die Wanduhrzeit musste ich mehrere Sachen beachten. Erstens kann die tatsächliche Fitnessfunktion für beide dieselbe sein. Ich nehme bei beiden die Laufzeit (entweder CPUT oder WCT) und gebe sie dem Fuzzer weiter. Dank dem Aufbau vom Fuzzer ist es möglich, die Zeiten einfach weiterzugeben, ohne diese vorher zu sortieren.

Die besten Inputs werden dann durch Tournament Selection [4] ausgewählt und zu Grammar-Neugenerierung weiterverarbeitet.

Die Berechnung der Zeiten ist dank dem Python eigenen 'time' [6] Package relativ einfach gestaltet.

6.1.1 Wanduhrzeit

Bei der Wall Clock Time (WCT) habe ich die Zeit genommen, genau bevor ich die SUT gestartet habe, dann habe ich diese ausgeführt, und die Zeit danach wieder gemessen. Aus der Differenz der beiden Zeiten ergab sich dann die WCT - die Zeit, die gebraucht wurde, um die SUT mit einem Input auszuführen.

6.1.2 CPU-Zeit

Bei der CPUT - ähnlich wie bei der WCT - wurde mit einer kleinen Änderung auch die Zeit genommen. Dem Package 'time' kann man direkt mitteilen, dass man nur die CPU-Zeit messen möchte, also die Zeit, die die CPU damit verbringt, die SUT auszuführen, ohne Wartezeiten oder In-, Outputs.

Nachdem ich die Zeit **x**, die verbraucht wurde, extrahiert habe, habe ich **x** an die Fitnessfunktion weitergegeben. Dabei wurde das mit 10000 multipliziert, damit ich einen Wert bekomme, womit der Fuzzer dann besser arbeiten kann.

6.1.3 Abstürze beim Zeitverbrauch

Inputs, die einen Fehler erzeugt haben, haben den Wert **BUG** und diejenigen, die keinen Fehler erzeugt haben, den Wert **NO BUG** zugewiesen bekommen.

Ich habe mich dafür entschieden, dass die Fitness von Inputs, die den Wert BUG haben, ignoriert werden. Die Begründung dafür ist, dass der Fuzzer dann nicht die komplette SUT durchlaufen hat, sondern gegebenenfalls nur eine Zeile (im Fall, dass die SUT gleich am Anfang ein Bug hat, das diese zum Abstürzen bringt).

Deswegen habe ich beschlossen, dass dieser Wert ignoriert wird, damit ich eine sauberere Reihe an Daten habe.

6.1.4 Beispiel

Man möchte die CPU-Zeit (oder Wanduhrzeit) für einen Durchlauf mit dem Input X und dem Input Y in der SUT messen. In dem Fall wird zuerst -bevor die SUT mit dem Input gestartet wird- die aktuelle Zeit gespeichert. Danach wird die SUT mit dem Input gestartet. Nachdem es durchlaufen ist, wird die Zeit wieder geprüft, und diese Zeit wird aus der Startzeit subtrahiert und gespeichert. Die Zeitfunktion prüft bei der CPU-Zeit nur den Zeitverbrauch, wobei die CPU tatsächlich gearbeitet hat. Bei der Wanduhrzeit werden auch Eingabe-/Ausgabeoperationen betrachtet. Des Weiteren untersucht die Funktion, ob es einen Absturz gab.

In Unserem Fall gab es bei Input X keinen Absturz, aber bei Input Y hat sich ein Absturz ergeben. Diese Inputs werden an die Fitnessfunktion mit der gespeicherten Laufzeit und der Absturzvariable weitergegeben. Dann wird die Variable X in dieser Funktion mit dem Fitnesswert (die gespeicherte Laufzeit) eingeordnet, und die Variable Y ignoriert (da es einen Absturz gab).

Nachdem alle Variablen durchlaufen sind, werden die Inputs, die keinen Absurz verursacht haben, mit EvoGFuzz verarbeitet.

6.2 Speicherverbrauch

Bei der Implementierung der Fitnessfunktion für Speicherverbauch habe ich mich für das Python-eigene 'tracemalloc' [7] Package entschieden. Dieses Package bietet Rückverfolgung, wo ein Objekt zugeordnet wurde, Statistiken über zugewiesene Speicherblöcke pro Dateiname und pro Zeilennummer mit Gesamtgröße, Anzahl und durchschnittlicher Größe der zugewiesenen Speicherblöcke und Berechnung der Unterschiede zwischen zwei Snapshots, um Speicherlecks zu erkennen [7].

Tracemalloc funktioniert so, dass man den höchsten Speicherverbrauch während der Laufzeit am Ende der Ausführung der SUT erhält. Dieser höchste Wert bleibt für jede Wiederholung gespeichert. Bei der Erkennung vom Speicherverbrauch muss ich deswegen tracemalloc vor jeder Ausführung zurücksetzen.

Danach wird tracemalloc gestartet und die SUT mit einem Input ausgeführt. Nach der Ausführung bekomme ich den 'peak' Wert zurück, stoppe tracemalloc und gebe der Fitnessfunktion den Wert weiter.

Bei der Fitnessfunktion wird dann dieser Wert als Fitness der Inputs gespeichert und mit dem Fuzzer weiterbearbeitet.

6.2.1 Abstürze beim Speicherverbrauch

Hier muss ich ebenso wie beim Zeitverbrauch auf die Abstürze aufpassen. Wenn ein Absturz während dem Testen passiert, darf dieser Input nicht betrachtet werden, da dieser die Ergebnisse verfälschen könnte.

Falls ein Absturz geschieht, bekommt dieser Input den Wert BUG, und dadurch kann man schnell und effizient entscheiden, ob dies zur Weiterverabeitung gehört oder entsorgt wird.

6.2.2 Beispiel

Die Funktionsweise möchte ich hier ähnlich erläutern, wie sie beim Zeitverbrauch vorgestellt wurde. Man hat zwei Inputs, X und Y und die sollen in der SUT auf Speicherverbrauch untersucht werden.

Der Aufbau der Tracemalloc funktioniert so, dass alles, was zwischen dem Start und Ende des Tracing in dieser Funktion gestartet wird und Speicher verbraucht, in eine Variable gespeichert wird. Es wird des Weiteren eine Variable erstellt, in der gespeichert wird, wenn es einen Absturz gab.

Da es im Beispiel bei der Variable Y einen Absturz gibt, wird diese wieder ignoriert, und X in der Fitnessfunktion verarbeitet. Danach bekommt X den Speicherverbrauch als Fitnesswert und wird schließlich an EvoGFuzz weitergegeben.

6.3 Exception Tracking

Für das Exception-Tracking habe ich eine zusätzliche Option in der Input-Klasse implementiert, die Möglichkeit, Fehlermeldungen mit einzugeben. Diese Funktion ermöglicht es, aufgetretene Exceptions bei einer späteren Überprüfung des Inputs an die Fitnessfunktion zu übergeben. Es wird also überprüft, ob eine Exception aufgetreten ist und wenn ja, welche.

Für die Fitnessfunktion muss man angeben, nach welcher Exception gesucht werden soll, dies ist nämlich im Code festgeschrieben. Sie prüft dann, welche Exceptions ausgelöst wurden, und weist diesem Input einen Fitnesswert von 1 zu, wenn es die gesuchte Exception ist und ansonsten Null (um zu vermeiden, dass andere mögliche Exceptions das Fuzzing verfälschen). Dies ermöglicht eine gezielte Suche nach Exceptions und verbessert die Softwarequalität durch die Aufspürung und Korrigierung von Fehlern.

6.4 Codeabdeckung

Um die Genauigkeit des Trace [23] zu verbessern, habe ich dem Tracer-Code eine Zeile hinzugefügt, die den Namen der überwachten Funktion überprüft. Dadurch wird

sichergestellt, dass nur relevante Funktionen überwacht werden und keine anderen Funktionen oder Unterpakete versehentlich erkannt werden. Wenn sich jedoch der Funktionsname ändert oder eine andere SUT verwendet wird (da der Name derzeit im Code festgelegt ist), muss diese Zeile manuell angepasst werden.

6.4.1 Abstürze bei der Codeabdeckung

Abstürze können zu verzerrten Fuzzing-Ergebnissen führen, da der Fuzzer in diesem Fall möglicherweise nicht alle möglichen Pfade und Codezweige durchlaufen ist.

Wenn ein Absturz auftritt, ist der Fitnesswert dieser Inputs 0. Wenn kein Absturz auftritt, wird der Fitnesswert basierend auf dem vom Tracer zugewiesenenen Wert berechnet. Dieser Wert gibt an, wie viele Codezeilen durchlaufen wurden.

6.4.2 Beispiel

Die Funktionsweise ist hier ebenso ähnlich wie bei den vorhin vorgestellten Funktionen, wie Zeit- und Speicherverbrauch. Wir nehmen zwei Inputs, X und Y. Wir haben bereits den Namen der SUT in den Fuzzer eingetragen.

Nach dem Start des Fuzzers wird erstens der Tracer mit dem Namen der Funktion initialisiert. Zuerst wird eine Variable erstellt, wo der Tracer die maximale Anzahl der Codezeilen speichert. Danach wird die SUT mit den Inputs aufgerufen, und der Tracer zählt dabei, wie viele Zeilen Code durchlaufen werden. Nachdem die Inputs durchlaufen sind, wird geprüft, wie viele von den maximalen Codezeilen durchlaufen wurden, und diese werden prozentual ausgerechnet (z.B es wurden 87 Zeilen von 100 durchlaufen, also 87%). Dieser Prozentsatz wird dann als der Fitnesswert gespeichert.

Wie vorhin auch, ist der Input X ohne Probleme durchgelaufen, Input Y hatte jedoch einen Absturz. Dementsprechend wird nur an X weitergearbeitet, und Y wird ignoriert.

6.5 Anmerkung

Im Fuzzer kann man nach den Inputs suchen, die die oben genannten Eigenschaften vorweisen. Es werden jedoch **keine** weiteren Einzelheiten über die Inputs angezeigt. Beispielsweise findet man bei der Codeabdeckung die Inputs, die das meiste des Codes abgedeckt haben, jedoch nicht, wie viele davon. Oder beim CPU-Zeitverbrauch bekommt man die Inputs, die die meiste Zeit verbraucht haben, um verarbeitet werden zu können, jedoch nicht, wie viele.

7 Evaluation

7.1 Testsubjekt

Diese Funktion wurde von mir entwickelt, um den Fuzzern eine optimale Testumgebung zu bieten. Wie bei Algorithmus 1 zu sehen ist, basiert sie auf einem Input und entscheidet je nachdem, welcher Input übergeben wurde, welche spezifische Funktion ausgeführt werden soll. Die möglichen Inputs sind 'cov' für Codeabdeckung, 'time' für Wanduhrzeit, 'cpu' für CPU-Zeit, 'mem' für Speicherverbrauch, 'exception' für einen ValueError und 'error' für einen TypeError. Außerhalb dieser sechs Inputs werden alle anderen ignoriert und nicht verarbeitet [11] .

Algorithm 1 Testsubjekt

```
Require: inp
  inp ← str(inp)
  bigNumber ← 1
  if mem in inp then
      bigList ← sum(list(range(10000))
  if cpu in inp  then
      i ← 1
      while i ≤ 10000 do
          bigNumber ← bigNumber × i
          i ← i + 1
  if time in inp  then
      time.sleep(0.1)
  if cov in inp then
      X ← 1
      Y ← 1
      Z ← 1
  if error in inp then
      throw TypeError
  if exception in inp then
      throw ValueError
```

7.2 Aufbau der Evaluation

Die Tests bezüglich des Baseline EvoGFuzz und meiner erweiterten Variante wurden mit dem selben Programm durchgeführt [11]. Diese Tests wurden auf einem MacBook Air (M1, 2020) mit Apple M1 Chip und 8GB Speicher ausgeführt.

Zuerst wurde der Baseline EvoGFuzz mit meiner erweiterten EvoGFuzz-Variante verglichen. Für jedes Testfall-Szenario wurde ein Diagramm erstellt, bei dem auf der

vertikalen (Y) Achse die Anzahl der Vorkommnisse des passenden Inputs und auf der horizontalen (X) Achse der Zeitverlauf dargestellt wurde. Es wurden insgesamt 30 Durchläufe mit jeweils 31 Iterationen durchgeführt.

Jede blaue Linie ist ein Durchlauf und die schwarze Linie stellt den Median aller Durchläufe dar.

Da beide Fuzzer dieselben Anfangsbedingungen haben, verwende ich dieselbe zufällig ausgewählte Eingabedatei, um den ursprünglichen Input zu erstellen. Die ersten Inputs bestanden aus den folgenden Elementen:

Bezeichnung	Input	Anzahl
Wanduhrzeit	time(<Zahlen>)	17
CPU-Zeit	cpu(<Zahlen>)	18
Bestimmte Exception	exception(<Zahlen>)	18
Absturz	error(<Zahlen>)	8
Speicherverbrauch	mem(<Zahlen>)	19
Codeabdeckung	cov(<Zahlen>)	20

Um sicherzustellen, dass die generierten Inputs als 'frisch generiert' wirken, wurde ein Input-Generator verwendet. Dadurch können Inputs erzeugt werden, die vielfältig und unterschiedlich sind. Das bedeutet aber auch, dass die generierten Inputs nicht alle gleichverteilt sind, sondern eine Vielzahl von Variationen aufweisen.

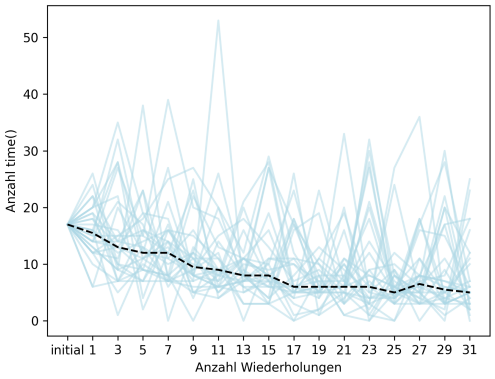
8 Ergebnisse

8.1 Zeitverbrauch

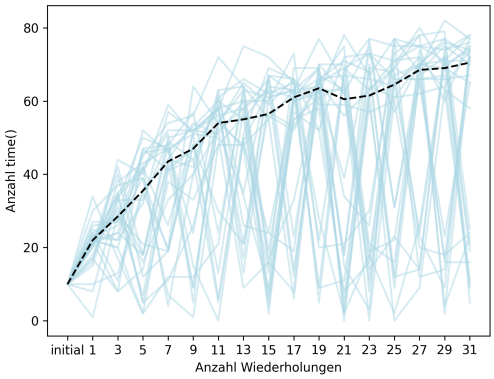
Eine detaillierte Analyse des Vergleichs zwischen dem Basis-EvoGFuzz und meiner erweiterten EvoGFuzz-Version zeigt deutlich, dass der Baseline nicht speziell darauf ausgelegt ist, verschiedene Probleme wie die Wanduhrzeit und die CPU-Zeit zu berücksichtigen.

Nach der Integration der neuen Funktionen in meine erweiterte Version von EvoGFuzz verbesserte sich die Leistung jedoch erheblich. Neue Funktionen sollen sich bei der Generierung von Inputs auf die Identifizierung von Problemen im Zusammenhang mit der Wanduhrzeit und der CPU-Zeit konzentrieren.

8.1.1 Wanduhrzeit



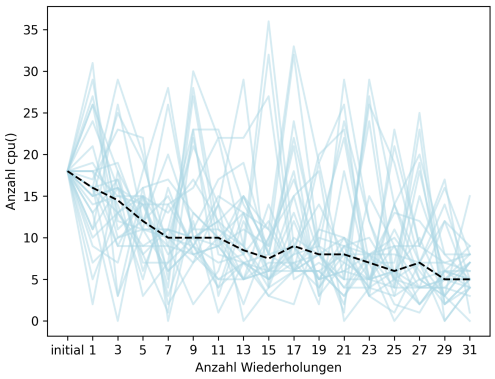
Baseline EvoGFuzz



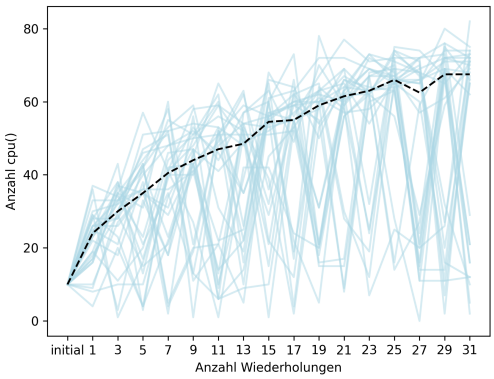
Erweiterter EvoGFuzz

Abbildung 2: Anzahl der time() Vorkommnisse im Baseline- und erweiterten EvoGFuzz

8.1.2 CPU-Zeit



Baseline EvoGFuzz



Erweiterter EvoGFuzz

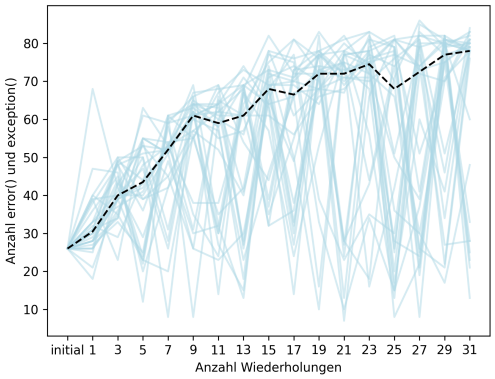
Abbildung 3: Anzahl der cpu() Vorkommnisse im Baseline- und erweiterten EvoGFuzz

8.2 Abstürze

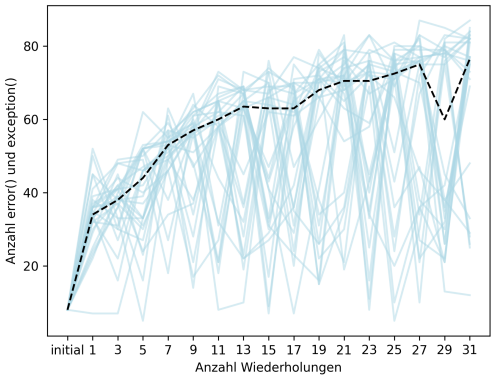
Bei Abstürzen ist die Dynamik etwas anders, da der Baseline darauf ausgelegt ist, solche Fehler zu erkennen und die Inputgenerierung entsprechend anzupassen. Allerdings hat der Baseline seine Grenzen, wenn es darum geht, bestimmte Exceptions zu finden und die Inputs zu generieren, die diese Exceptions auslösen.

Mit meiner erweiterten EvoGFuzz-Variante ist es nun möglich, nach bestimmten Exceptions zu suchen und die Inputs zu generieren, die zu diesen Exceptions führen.

8.2.1 Genereller Absturz



Baseline EvoGFuzz

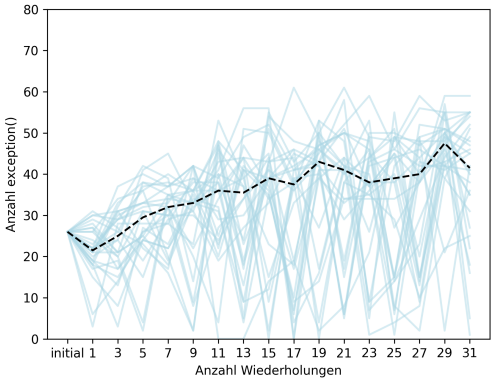


Erweiterter EvoGFuzz

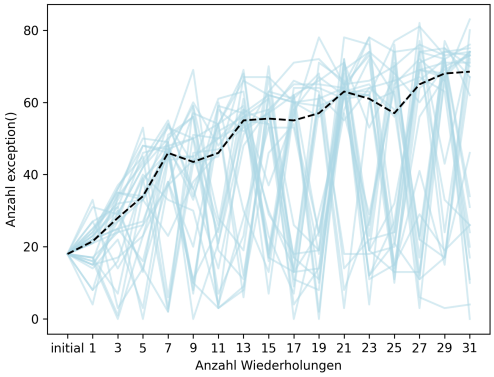
Abbildung 4: Anzahl der error() und exception() Vorkommnisse im Baseline- und erweiterten EvoGFuzz

Wie man auf Abbildung 4 sieht, haben beide Fuzzer die gleichen Ergebnisse geliefert, wenn es darum geht, einen Absturz zu erkennen.

8.2.2 Absturz durch eine bestimmte Exception (Exception tracking)



Baseline EvoGFuzz



Erweiterter EvoGFuzz

Abbildung 5: Anzahl der exception() Vorkommnisse im Baseline- und erweiterten EvoGFuzz

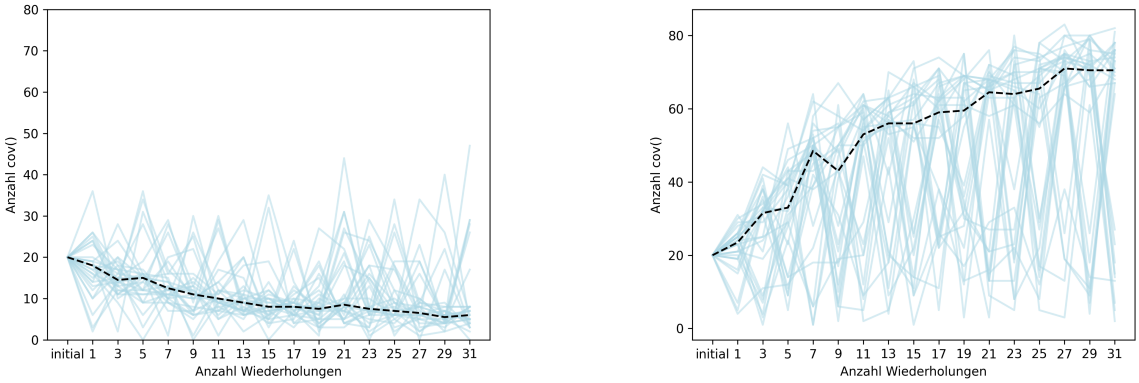
Beim Vergleich zwischen dem einfachen Absturz tracking -das der Baseline bieten kann- und dem bestimmten Exception Tracking wird es jedoch interessant.

Der Baseline-Ansatz verfolgt eine allgemeinere Herangehensweise und berücksichtigt

die Art der ausgelösten Exception nicht explizit. Dadurch werden sowohl error() als auch exception() erfasst und deswegen wird die Anzahl von exception() im Vergleich zu meiner Variante niedriger.

Bei meiner Variante ist es jedoch sichtbar, dass sie auf eine bestimmte Exception fokussiert. Durch diese Fokussierung wird eine signifikante Verbesserung in Bezug auf die Generierung von Inputs erzielt, die genau diese eine Exception hervorrufen.

8.3 Codeabdeckung



Baseline EvoGFuzz

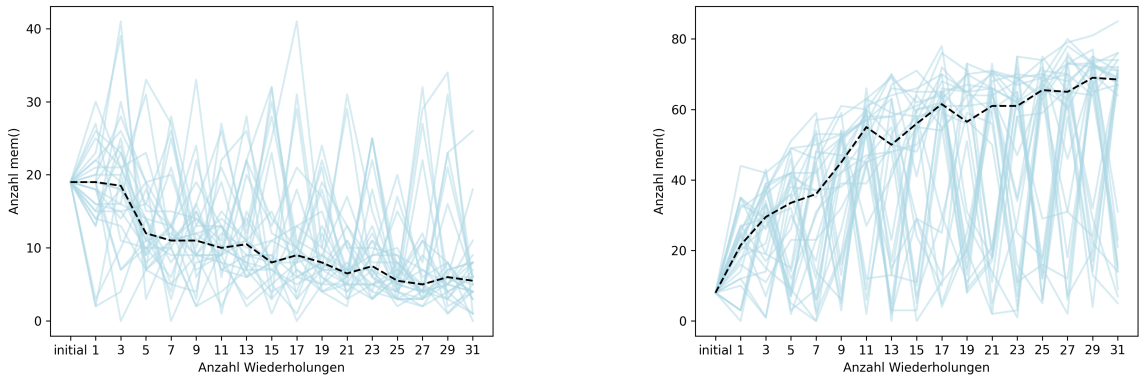
Erweiterter EvoGFuzz

Abbildung 6: Anzahl der cov() Vorkommnisse im Baseline und erweitertem EvoGFuzz

Bei der Codeabdeckung wird in der Fitnessfunktion des Fuzzers überprüft, wie viele Zeilen des Codes abgedeckt wurden. Für diesen Testfall wurde die Testfunktion leicht angepasst. Anstelle einer klaren Trennung zwischen den Variablen 'time', 'cpu' usw. wurde überall eine zweite Möglichkeit hinzugefügt, und zwar '<variable> **oder** cov'. Dadurch konnte der Fuzzer besser erkennen, dass sich die Anzahl der bearbeiteten Codezeilen bei der Variable 'cov' erheblich erhöht hat.

Da sich der Baseline-Fuzzer auf das Auffinden von Exceptions konzentriert, lag der Fokus auf den Variablen 'error' und 'exception'. Wie bereits in der Implementierung der Funktion erwähnt wurde, wurden die Inputs, die einen Fehler ausgelöst haben, ignoriert. Dadurch lässt sich ein klarerer Ablauf des Programms beobachten und es werden mehrere Zeilen durchlaufen.

8.4 Speicherverbrauch



Baseline EvoGFuzz Erweiterter EvoGFuzz

Abbildung 7: Anzahl der mem() Vorkommnisse im Baseline- und erweitertem EvoGFuzz

Im Hinblick auf die Speicherverwaltung zeigt sich erneut (siehe Abbildung 7), dass die Baseline-Methode nicht speziell darauf ausgelegt ist, problematische Speicherverbrauchsmuster zu identifizieren. Andererseits zeigte die erweiterte Variante wesentliche Unterschiede in Bezug auf die identifizierten Variablen, die zu einem hohen Speicherverbrauch führen.

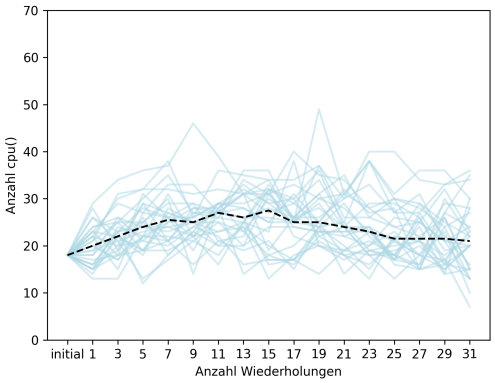
Darüber hinaus ist in der Baseline-Variante eine konsequente Reduzierung der Variablenanzahl zu beobachten, da der Fokus hauptsächlich auf der Anzahl der Abstürze liegt.

8.5 Ergebnisse der Testläufe von EvoGFuzz mit und ohne Mutation der Grammatiken

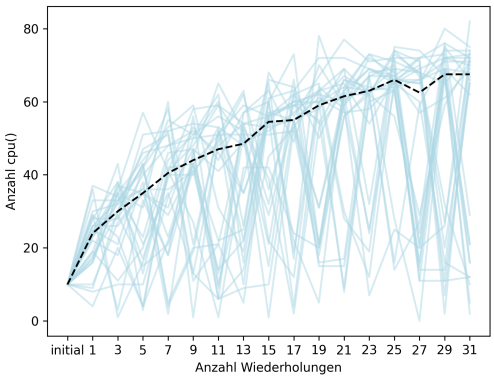
EvoGFuzz selbst wurde einer umfangreichen Testreihe unterzogen, um zu untersuchen, ob eine Verbesserung der Ergebnisse möglich ist, ohne die Grammatik zu mutieren. Das Ziel war es herauszufinden, ob die neu gelernten Grammatiken bereits eine optimale Lösung bieten oder ob durch Mutationen noch bessere Ergebnisse erzielt werden können.

Die Mutation bezieht sich **nicht** auf das Mutieren der Inputs, sondern das Mutieren der Grammatik. Die gelernte probabilistische Grammatik wird mutiert, um eine genetische Drift zu bestimmten Merkmalen der ausgewählten Individuen zu vermeiden. Bei einer solchen Drift würde die Grammatik nur Inputs mit spezifischen Merkmalen erzeugen, die den ausgewählten Individuen ähneln, von denen die Grammatik gelernt wurde [4]. Durch die Mutation entsteht eine mutierte Grammatik, die syntaktisch gültige Inputs erzeugt, deren Merkmale den ausgewählten Individuen ähneln, aber auch andere unerforschte Merkmale aufweisen können [4].

Die folgenden Abbildungen zeigen die Ergebnisse dieser Tests.

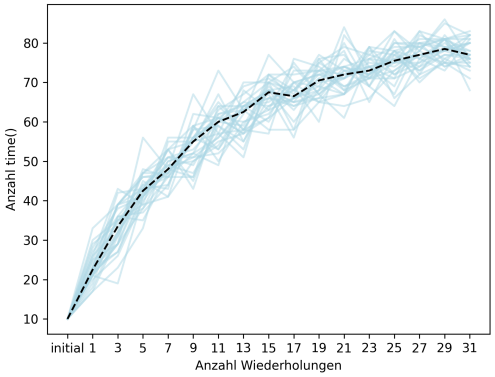


Erweiterter EvoGFuzz ohne Mutation

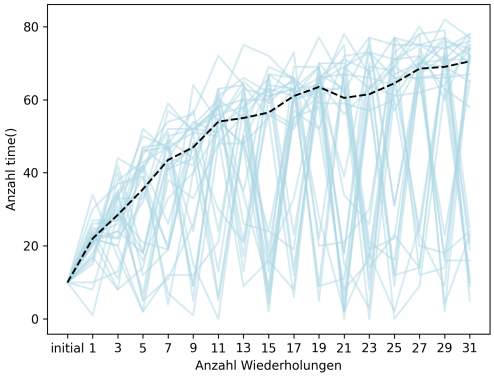


Erweiterter EvoGFuzz mit Mutation

Abbildung 8: Anzahl der cpu() Vorkommnisse im Baseline- und erweiterten EvoGFuzz



Erweiterter EvoGFuzz ohne Mutation



Erweiterter EvoGFuzz mit Mutation

Abbildung 9: Anzahl der time() Vorkommnisse im erweiterten EvoGFuzz ohne und mit Mutation

Die Abbildungen 9-12 zeigen deutlich, dass es bei der Verwendung von EvoGFuzz ohne Mutation kaum wesentliche Unterschiede im Endergebnis im Vergleich zu EvoGFuzz mit Mutation gibt. Allerdings fällt auch auf, dass der Fuzzing-Prozess ohne Mutation wesentlich weniger Versuche unternommen hat, um verschiedene Pfade mithilfe der Grammatiken zu erkunden.

Das bedeutet, bei EvoGFuzz ohne Mutation hat sich der Fuzzer die Inputs gemerkt, die dieses bestimmte Ereignis ausgelöst haben, und hat sich darauf konzentriert, diese Inputs zu generieren. Diese sogenannte genetische Drift ist in unserem Fall wegen des Testprogramms tatsächlich akzeptabel, aber wenn man mehrere Probleme in der SUT

hat, könnten diese gegebenenfalls nicht gefunden werden.

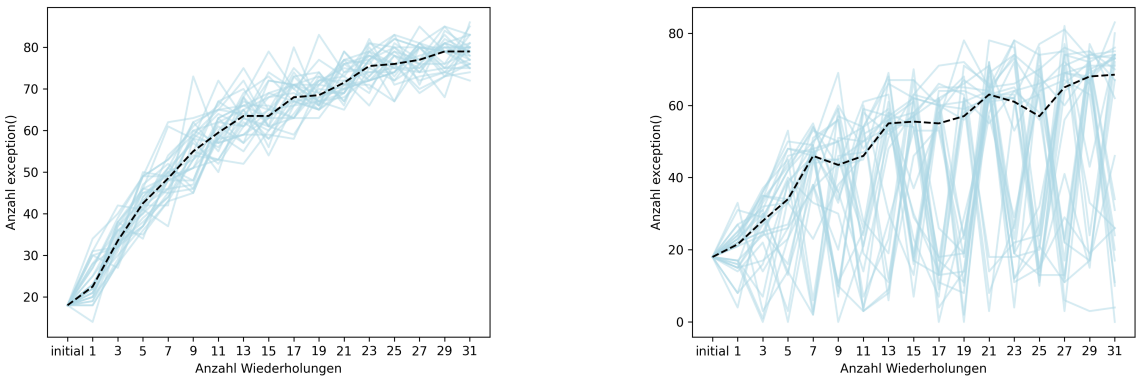
Zur Veranschaulichung der genetischen Drift und der Probleme von EvoGFuzz ohne Mutation werde ich ein kleines Beispiel vorstellen. Angenommen:

1. der Fuzzer hat von jedem möglichen Input einen Fall bekommen. Das heißt, die Initialen Inputs bestehen aus cov(<zahl), mem(<zahl>), cpu(<zahl>), time(<zahl>), exception(<zahl>) und error(<zahl>)

2. Man möchte nach Problemen beim Speicherverbrauch suchen

In diesem Fall findet der Fuzzer nach dem ersten Durchlauf heraus, dass bei mem(<zahl>) ein Speicherproblem liegt. Für die nächste Runde wird eine der anderen Inputs nicht mehr generiert, weil stattdessen ein mem(<zahl>) generiert wird. Und so geht es auch dann weiter, bis man gegebenenfalls nur noch mem(<zahl>) erhält.

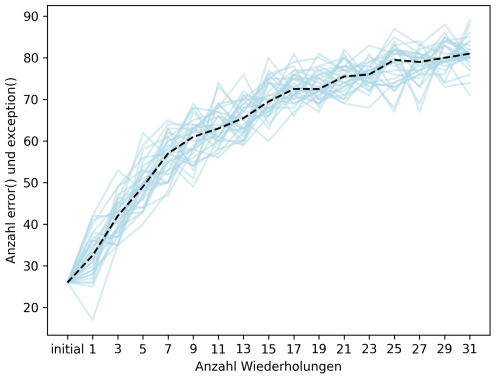
In dem Fall, wenn die SUT zum Beispiel mehrere Speicherprobleme hat, kann es zu einem Problem führen. Es ist nämlich möglich, dass die anderen Speicherprobleme gar nicht erst gefunden werden. Deswegen ist es unbedingt nötig, die Mutation bei der Grammatik beizubehalten.



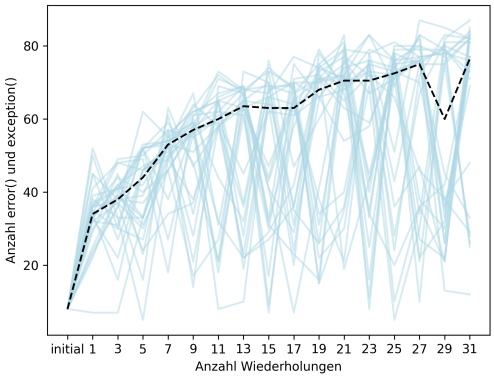
Erweiterter EvoGFuzz ohne Mutation

Erweiterter EvoGFuzz mit Mutation

Abbildung 10: Anzahl der exception() Vorkommnisse im erweiterten EvoGFuzz ohne und mit Mutation

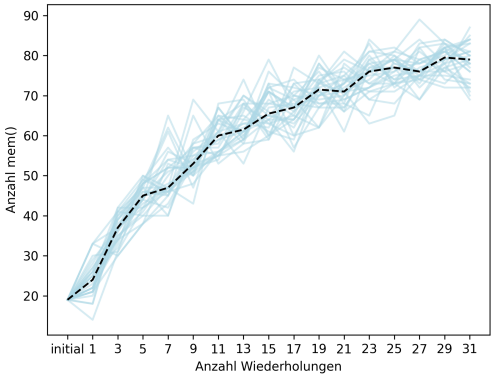


Erweiterter EvoGFuzz ohne Mutation

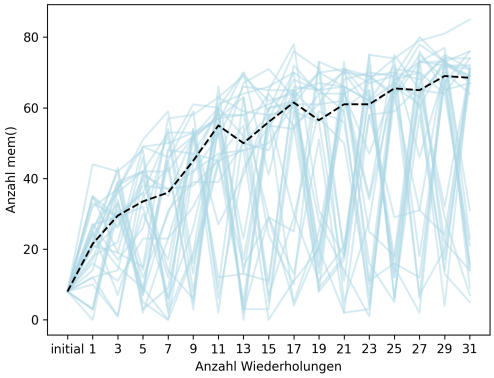


Erweiterter EvoGFuzz mit Mutation

Abbildung 11: Anzahl der error() und exception() Vorkommnisse im erweiterten EvoGFuzz ohne und mit Mutation



Erweiterter EvoGFuzz ohne Mutation

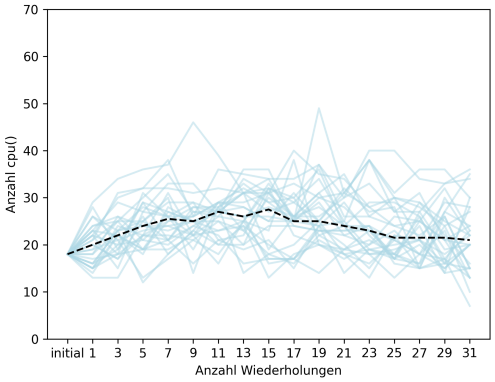


Erweiterter EvoGFuzz mit Mutation

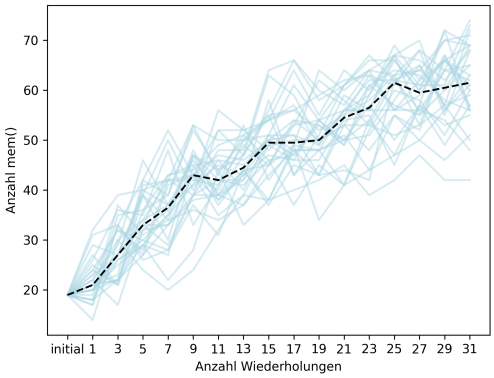
Abbildung 12: Anzahl der mem() Vorkommnisse im erweiterten EvoGFuzz ohne und mit Mutation

8.6 Interessante Erkenntnisse

Jedoch ist ein interessanter Fall vorgekommen. Bei der CPU-Zeit hat man zirka nur die Hälfte der Anzahl von cpu() Vorkommnissen bei den Outputs (siehe Abbildung 8). Die andere Hälfte der 'fehlenden Fälle' findet man nämlich unter mem(). Wie es sich herausstellte, hat der Code zur Speicherbelastung teilweise ungefähr denselben (oder sogar teilweise mehr) CPU-Verbrauch aufgewiesen, wie der Code zur CPU-Belastung (siehe Abbildung 13).



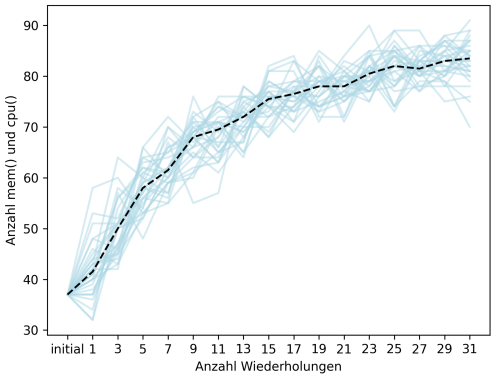
Erweiterter EvoGFuzz ohne Mutation



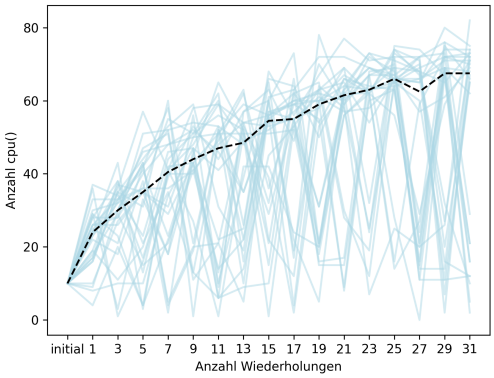
Erweiterter EvoGFuzz ohne Mutation

Abbildung 13: Anzahl der cpu()(links) und mem()(rechts) Vorkommnisse im erweiterten EvoGFuzz ohne Mutation bei der Suche nach hohem CPU-Verbrauch

Und wenn man sowohl die CPU als auch die Speicherbelastung zusammenrechnet, kommt man schließlich auf die selben Ergebnisse wie EvoGFuzz mit Grammatikmutation.



Erweiterter EvoGFuzz ohne Mutation



Erweiterter EvoGFuzz ohne Mutation

Abbildung 14: Anzahl der cpu() und mem() Vorkommnisse im erweiterten EvoGFuzz ohne Mutation (links) und cpu() Vorkommnisse im erweiterten EvoGFuzz mit Mutation

9 Vergleich mit PerfFuzz

9.1 Auswahl des Fuzzers

PerfFuzz ist ein auf Mutationen basierender Fuzzer, der darauf abzielt, Leistungsprobleme aufzudecken. Aus diesem Grund habe ich mich entschieden, meinen erweiterten Fuzzer mit PerfFuzz zu vergleichen.

Die Funktionsweise von PerfFuzz besteht darin, die anfänglichen Inputs durch Mutationen zu verändern, um potenzielle Leistungsprobleme zu finden. Das bedeutet, dass PerfFuzz die vorhandenen Testinputs modifiziert, indem er verschiedene Änderungen an den Daten vornimmt, wie z.B. das Ändern von Werten, das Hinzufügen oder Entfernen von Zeichen oder das Durchführen anderer Transformationen [9]. Dadurch werden verschiedene Varianten der Inputs erzeugt, um mögliche Leistungsprobleme im Testprogramm aufzudecken.

9.2 PerfFuzz Setup

Beim Starten von PerfFuzz bin ich jedoch auf ein Problem gestoßen. Man muss beim Aufrufen der Funktion die maximale Größe der Inputs angeben. Diese waren derzeit bei über 2 MB (100 Testfälle) und dadurch wurde PerfFuzz unbenutzbar langsam. Deswegen wurde der Initial Input auf 21 Fälle reduziert, damit diese unter 256 KB bleiben, nämlich:

Bezeichnung	Input	Anzahl
Wanduhrzeit	time(<Zahlen>)	5
CPU-Zeit	cpu(<Zahlen>)	2
Bestimmte Exception	exception(<Zahlen>)	4
Absturz	error(<Zahlen>)	8
Speicherverbrauch	mem(<Zahlen>)	1
Codeabdeckung	cov(<Zahlen>)	2

Bei PerfFuzz kamen dann immer nur diese 21 Werte zurück. Jedoch wurden bei EvoGFuzz wie immer 100 Werte zurückgeliefert. Deswegen wurden die Ergebnisse beim Berechnen so angepasst, damit sie die gleiche Gewichtung haben.

9.3 Testprogramm

Ich habe das gleiche Testprogramm sowohl für PerfFuzz als auch für das erweiterte EvoGFuzz verwendet. Allerdings habe ich das Testprogramm für das erweiterte EvoGFuzz in die Programmiersprache 'C' übersetzt, da PerfFuzz nur C-Programme als SUT akzeptiert.

Für das erweiterte EvoGFuzz musste ich das Testprogramm als 'subprocess' [5] starten und die Ergebnisse der Aufrufe auswerten. Dies bedeutet, dass ich das Testprogramm als separaten Prozess gestartet habe und dann die Outputs und Rückgabewerte des Prozesses analysiert habe, um Informationen über das Verhalten der SUT zu erhalten.

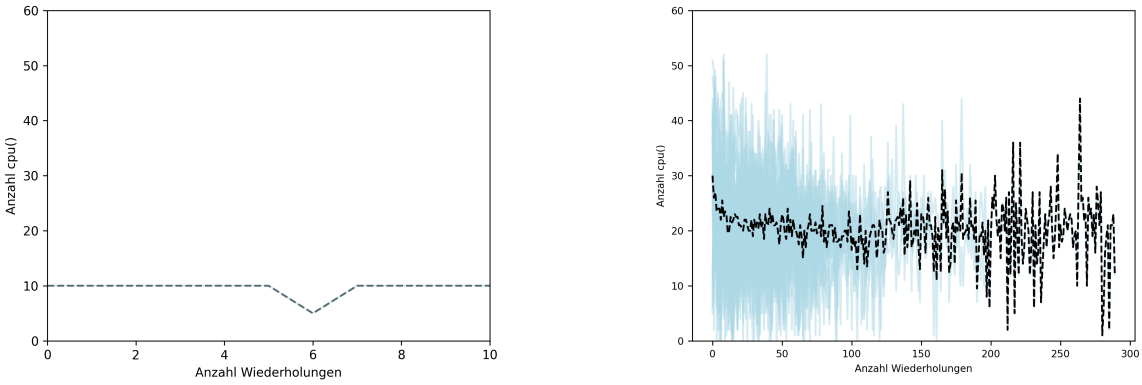
9.4 Ergebnisse der Testläufe

Die Testläufe zwischen PerfFuzz und dem erweiterten EvoGFuzz wurden auf einem Server der Humboldt-Universität zu Berlin durchgeführt. Der Server verfügt über die

folgenden Spezifikationen:

- Modell: Asus ESC4000
- CPU: Xeon 6354
- Anzahl der CPUs: 2
- Anzahl der Kerne: 36
- Anzahl der Threads: 72
- Taktfrequenz: 3,6 GHz
- RAM: 1 TB

9.4.1 Zeitverbrauch



PerfFuzz Erweiterter EvoGFuzz

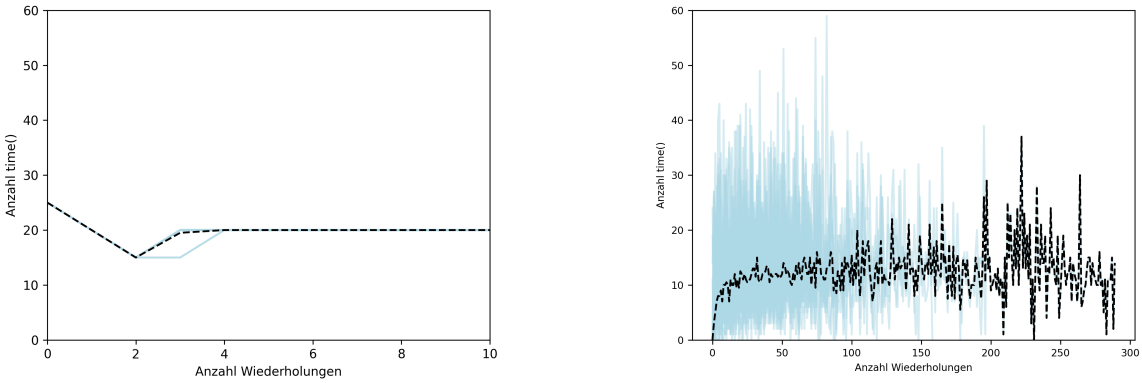
Abbildung 15: Anzahl der cpu() Vorkommnisse in PerfFuzz und dem erweiterten EvoG-Fuzz

Auf Abbildung 15 ist es gut zu sehen, dass PerfFuzz beim Arbeiten den Wert 'cpu(<zahlen>)' einmal wegmutiert hat, und danach wieder festgestellt hat, dass er dadurch weniger verbraucht hat, jedoch hat er nicht weiter diese Richtung verfolgt, die Inputs zu mutieren.

EvoGFuzz hat hier sichtbar besser abgeschnitten. Die Mittelwertlinie ist stets über der Mittelwertlinie von PerfFuzz.

PerfFuzz generiert Inputs, die diverse Probleme bei der SUT auslösen können, die einen hohen Leistungswert ergeben, wie zum Beispiel die Anzahl der bei malloc-Anweisungen zugewiesenen Bytes, die Anzahl der Cache-Misses oder Page Faults bei Speicherlade-/Speicheranweisungen, die Anzahl der I/O-Operationen über Systemkomponenten hinweg usw [9].

Um sicherzustellen, dass ich alle potenziellen Leistungsprobleme gründlich untersuche, habe ich alle relevanten Funktionen ausgewertet, die einen hohen Leistungsverbrauch aufweisen können. Dabei habe ich sowohl den CPU-Zeitverbrauch als auch den Wanduhrzeit- und Speicherverbrauch berücksichtigt.



PerfFuzz Erweiterter EvoGFuzz

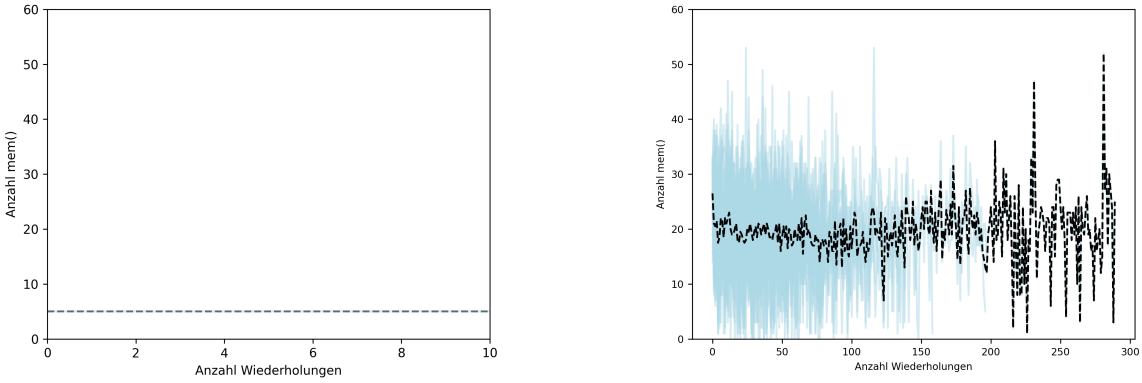
Abbildung 16: Anzahl der time() Vorkommnisse in PerfFuzz und dem erweiterten EvoGFuzz

Jedoch scheint PerfFuzz den Wanduhrzeitverbrauch nicht so intensiv betrachtet zu haben wie ich es getan habe. Dies könnte darauf hinweisen, dass PerfFuzz möglicherweise stärker auf andere Leistungsaspekte fokussiert ist oder andere Schwerpunkte bei der Durchführung von Tests setzt.

EvoGFuzz hat in einigen Fällen eine höhere Anzahl an Inputs von 'time(<zahlen>)' erreicht, insbesondere zu Beginn der Testläufe.

Allerdings zeigt die Mittelwertlinie, dass diese Werte jedoch nicht immer der Fall waren.

9.4.2 Speicherverbrauch



PerfFuzz Erweiterter EvoGFuzz

Abbildung 17: Anzahl der cpu() Vorkommnisse in PerfFuzz und dem erweiterten EvoG-Fuzz

PerfFuzz hat den Speicherverbrauch (große Allokation von einem Array) nicht als Leistungsverbrauch erkannt, und hat nichts mit dem Input gemacht.

Beim Speicherverbrauch sieht man tatsächlich den größten Unterschied zwischen Perf-Fuzz und EvoGFuzz. Die Mittelwertlinie von EvoGFuzz befindet sich stetig immer über dem Wert von PerfFuzz.

Das bedeutet, dass tatsächlich der größte Verbrauch an Leistung während der Speicher-allokationen passiert und EvoGFuzz diese Inputs, die Speicherallukationen auslösen, erwartungsgemäß gefunden hat.

10 Evaluierung

Da ich unabhängige Stichproben betrachte und keine Annahmen über die Verteilung der Ergebnisse treffen kann, verwende ich den nicht-parametrischen Mann-Whitney-U-Test [4, 20]. Dieser Test ermöglicht es zu überprüfen, ob der Mittelwert von der Anzahl der Vorkommnisse der beiden Fuzzern für jedes Problem wesentlich unterschiedlich ist. Durch die Anwendung dieses Tests kann ich statistische Informationen erhalten, die mich bei der Bewertung der Unterschiede zwischen den Fuzzern unterstützen.

10.1 Ergebnisse der Mann-Whitney-U Tests zwischen dem Baseline und dem erweiterten EvoGFuzz

- H_0: Die Anzahl der gefundenen Inputs, die einen bestimmten Fall auslösen, ist **gleich geblieben**.

- H_1: Die Anzahl der gefundenen Inputs, die einen bestimmten Fall auslösen, ist beim überarbeiteten EvoGFuzz **höher als** beim Baseline.

Testfall	p-Wert	Kann die Null-Hypothese ausgeschlossen werden?
Wanduhrzeit	0,00000038	Ja
CPU-Zeit	0,00000038	Ja
Bestimmte Exception	0,00189831	Ja
Absturz	0,69745252	Nein
Speicherverbrauch	0,00000054	Ja
Codeabdeckung	0,00000041	Ja

Die Ergebnisse der MWU-Tests zeigten wesentliche Unterschiede bei der CPU-Zeit, der Wanduhrzeit, dem Speicherverbrauch, der Suche nach bestimmten Exceptions und der Codeabdeckung. Dies bedeutet, dass die erweiterte Version von EvoGFuzz in diesen Aspekten bessere statistische Ergebnisse erzielt als der Baseline.

Tests für den 'generellen Absturz' ergaben für beide Versionen ähnliche Ergebnisse. Dies zeigt, dass der Baseline diese Art von Fehler bereits erkennen konnte und keine wesentlichen Unterschiede zwischen den Versionen festgestellt wurden.

10.2 Ergebnisse der Mann-Whitney-U Tests zwischen PerfFuzz und dem erweiterten EvoGFuzz

- H_0: Die Anzahl der gefundenen Inputs, die einen bestimmten Fall auslösen, ist **gleich geblieben**.
- H_1: Die Anzahl der gefundenen Inputs, die einen bestimmten Fall auslösen, ist beim überarbeiteten EvoGFuzz **höher als** bei PerfFuzz.

Testfall	p-Wert	Kann die Null-Hypothese ausgeschlossen werden?
Wanduhrzeit	0,99999939	Nein
CPU-Zeit	0,00000008	Ja
Bestimmte Exception	-	-
Absturz	-	-
Speicherverbrauch	0,00000003	Ja
Codeabdeckung	-	-

Die Ergebnisse der MWU-Tests zeigten wesentliche Verbesserung bei der CPU-Zeit und dem Speicherverbrauch. Dies bedeutet, dass die erweiterte Version von EvoGFuzz in diesen Aspekten bessere statistische Ergebnisse erzielt als PerfFuzz.

Für die Fälle, wo ein Absturz getestet wurde, wurden die Ergebnisse ignoriert, beziehungsweise nicht getestet, da PerfFuzz nach Leistungsproblemen statt nach Abstürzen sucht.

Bei der CPU-Zeit und dem Speicherverbrauch konnte ein vielversprechendes Ergebnis

erzielt werden. In über 99% der Fälle kann nachgewiesen werden, dass die erweiterte Version von EvoGFuzz eine bessere Leistung erbringt.

Für die Wanduhrzeit konnte kein gutes Ergebnis erreicht werden. Die Nullhypothese kann in diesem Fall nicht ausgeschlossen werden.

10.3 Diskussion der Ergebnisse

- **RQ1:** Ist der erweiterte EvoGFuzz effektiver bei der Suche nach pathologischen Inputs, die nicht-funktionales Programmverhalten verursachen als der Baseline EvoGFuzz?
- **RQ2:** Ist der erweiterte EvoGFuzz effektiver bei der Suche nach pathologischen Inputs, die Performance hindern, als PerfFuzz?

Um die oben genannten Forschungsfragen beantworten zu können, muss ich zunächst die bereits erreichten Ergebnisse betrachten.

In den meisten Testfällen scheint die verbesserte Version von EvoGFuzz statistisch wesentlich bessere Ergebnisse zu erzielen als die Basisversion. Die Anzahl der Inputs, die zu einem bestimmten Ergebnis führen, war deutlich höher als bei der Basisversion, es sei denn, es trat eine Exception oder ein Absturz auf.

Diese Behauptung wird durch die Diagramme und MWU-Tests gestützt, die zeigen, dass die erweiterte Version eine bessere Leistung aufweist. Es ist deutlich zu sehen, dass die erweiterte Version in fast allen Bereichen besser abschneidet.

Die erste Frage lautet also: Ist der erweiterte EvoGFuzz effektiver bei der Suche nach pathologischen Inputs, die nicht-funktionales Programmverhalten verursachen als der Baseline EvoGFuzz? Die Antwort auf diese Frage ist zweifellos positiv.

Um **RQ2** beantworten zu können, müssen zuerst die Probleme von PerfFuzz klargestellt werden.

Das Hauptproblem von PerfFuzz liegt darin, dass er keine Grammatiken benutzt, um die Inputs zu generieren, sondern es werden an den initialen Inputs zufallsartige Mutationen durchgeführt, und dabei wird die Syntax der SUT ignoriert. Diese Schwäche des Fuzzers kann dazu führen, dass die Inputs, die man vorher bestimmt hat, so weit wegmutiert werden, dass man die SUT nicht einmal starten kann.

Man sieht diese Schwäche von PerfFuzz auch klar in den Testergebnissen. Wenn EvoGFuzz neue Inputs generiert, tut er das mit Bedacht auf die Syntax, damit man einen Korpus von 100 Inputs hat, die die SUT definitiv durchlaufen. Wenn PerfFuzz die Inputs jedoch mutiert hat, sind nie welche **dazu** gekommen, es wurden immer welche

wegmutiert.

Das liegt aber tatsächlich an dem Unterschied zwischen dem Aufbau der beiden Fuzzer.

Was jedoch die Ergebnisse wesentlich verbessert hat, ist die Erweiterung von EvoGFuzz. Der Baseline EvoGFuzz hat so gut wie gar keine Überschneidung mit PerfFuzz. PerfFuzz sucht nach Leistungsproblemen, während EvoGFuzz Abstürze aufspürt.

Nach meiner Erweiterung sucht EvoGFuzz nicht nur nach Abstürzen, sondern kann auch Leistungsprobleme erkennen. Diese Behauptung wird durch die Diagramme und MWU-Tests gestützt, die zeigen, dass die erweiterte Version von EvoGFuzz eine bessere Leistung aufweist als PerfFuzz.

Anschließend kommen wir zur zweiten Frage: Ist der erweiterte EvoGFuzz effektiver bei der Suche nach pathologischen Inputs, die Performance hindern, als PerfFuzz? Und die Antwort auf diese Frage ist ebenso zweifellos positiv.

11 Einschränkungen

Die Testläufe zeigten vielversprechende Ergebnisse, es ist jedoch zu beachten, dass diese in einer bestimmten Testumgebung durchgeführt wurden, die eine ideale Möglichkeit geboten hat, die Eingabe mit dem entsprechenden Fehler abzugleichen. Es ist zu beachten, dass dieser Ansatz zwar auf einem synthetischen Beispiel ausgewertet wurde, aber die Ergebnisse können bei der Anwendung auf echte Programme variieren.

Es gibt einige Herausforderungen im Zusammenhang mit der Codeabdeckung in EvoGFuzz. Ein Problem besteht darin, dass die SUT explizit mit Funktionsnamen definiert werden muss, damit die Unterpakete von EvoGFuzz bei der Bewertung nicht berücksichtigt werden. Dies ist wichtig, um sicherzustellen, dass die Codeabdeckung korrekt gemessen wird und nicht durch interne Pakete von EvoGFuzz beeinträchtigt wird. Ein weiteres potenzielles Problem tritt auf, wenn die SUT selbst verschiedene Pakete verwendet. In solchen Fällen können diese zusätzlichen Pakete das Endergebnis der Codeabdeckung verfälschen.

Die tatsächliche Leistung und Effektivität dieser Methode in einem echten Programm hängt von verschiedenen Faktoren ab, beispielsweise der Komplexität des Codes, der Art der Fehler, der Struktur des Programms usw. Daher wird empfohlen, die Methode in einem breiteren Spektrum echter Verfahren auszuwerten, um eine zuverlässigere Schätzung ihrer Leistung zu erhalten.

12 Schlussfolgerung

Die Auswertung zeigt, dass mein Ansatz in der Lage war, EvoGFuzz soweit zu erweitern, dass es derzeit möglich ist, diverse Leistungsprobleme aufzudecken.

Insgesamt stellt die erweiterte Version von EvoGFuzz - mit signifikanten Verbesserungen bei den Testergebnissen und der Leistung - einen Fortschritt gegenüber der Basisversion dar.

Bei einem detaillierten Vergleich zwischen EvoGFuzz und PerfFuzz zeigt sich ein klarer Vorteil von EvoGFuzz bei der Suche nach pathologischen Inputs, die zu nichtfunktionalem Programmverhalten führen. EvoGFuzz erzielte signifikant bessere Ergebnisse und identifizierte eine wesentlich größere Anzahl solcher Inputs im Vergleich zu PerfFuzz.

Dies verdeutlicht die überlegene Effektivität von EvoGFuzz in Bezug auf die Erkennung und das Auffinden von potenziell problematischen Inputs, die zu unerwartetem oder nicht beabsichtigtem Verhalten führen können.

13 Zukünftige Arbeit

Zukünftige Arbeiten sollten meiner Meinung nach auch andere Aspekte des Software-Testens erforschen. Beispielsweise könnte man eine automatische Test-Software mit EvoGFuzz verbinden. Für diesen Fall bietet Detox [21] eine gute Alternative.

Bei Detox geht es darum, dass eine App für sowohl iOS als auch Android automatisch durchgetestet werden kann. Man muss dafür zuerst jeder Taste oder jedem Inputfeld eine Test-ID geben. Danach kann man mit den Tests anfangen. Detox hat eine einfache Syntax und lässt sich gut mit EvoGFuzz verbinden (Beispiel in Javascript):

- await expect(element(by.id('MyButton'))).toBeVisible();
- await element(by.id('MyButton')).tap();

Um EvoGFuzz mit Detox verbinden zu können, muss man zuerst die komplette Struktur der zu testenden Applikation beschreiben. Das heißt, was für Tasten oder bedienbare Elemente sich auf dem Screen befinden, und wo man hinkommt, wenn man eine davon drückt. Sobald man die komplette Struktur der Applikation als Grammatik beschrieben hat, könnte man direkt mit dem Testen anfangen. Ein kleines Beispiel für die Grammatik einer Applikation wäre:

Algorithm 2 Detox Grammatik

```
'<start>': ['<arith>']
'<arith>': ['<function>','<function><function>' ]
'<function>':
        ['await expect(element(by.id('<pictures>'))).toBeVisible();,

        'await expect(element(by.id('<button>'))).toBeVisible();
        await element(by.id('<button>')).tap(); ,

        '<arith>']
'<button>':
        ['nextPage',
        'prevPage',
        'exit'],
'<pictures>':
        ['headline',
        'footer',
        'CEO',
        'logo']
```

Mit dieser Grammatik wäre der Fuzzer in der Lage, alle Bilder und alle Tasten in dieser Applikation zu testen. Nachdem der Fuzzer einen Testkorpus generiert hat, muss dieser Korpus gespeichert und mit Detox gestartet werden.

Falls zum Beispiel irgendeins von den Bildern bei den Testläufen fehlen sollte, scheitern diese Testfälle, und man bekommt von Detox eine Fehlermeldung. Sobald man eine Fehlermeldung bekommen hat, könnte der Fuzzer den kompletten Weg zur Reproduktion speichern und dadurch beim Softwaretesten eine große Hilfe bieten.

14 Bibliografie

Literatur

[1] American fuzzy lop - documentation, Jul 2017.

[2] H. Cavusoglu, B. Mishra, and S. Raghunathan. The effect of internet security breach announcements on market value: Capital market reactions for breached firms and internet security developers. *International Journal of Electronic Commerce*, 9(1):70–104, 2004.

[3] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[4] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske. Evolutionary grammar-based fuzzing. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*, pages 105–120. Springer, 2020.

[5] P. S. Foundation. subprocess — subprocess management.

[6] P. S. Foundation. Time - time access and conversions.

[7] P. S. Foundation. tracemalloc — trace memory allocations.

[8] M. E. Khan and F. Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*, 3(6), 2012.

[9] C. Lemieux, R. Padhye, K. Sen, and D. Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

[10] R. McNally, K. Yiu, D. Grove, and D. Gerhardy. Fuzzing: the state of the art. 2012.

[11] G. Miklovics. Github repository. `https://github.com/Miklovig/evogfuzzplusplus`, 2023.

[12] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[13] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340, 2019.

[14] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller. Inputs from hell. *IEEE Transactions on Software Engineering*, 48(4):1138–1153, 2020.

[15] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[16] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[17] V. Vikram, R. Padhye, and K. Sen. Growing a test corpus with bonsai fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 723–735. IEEE, 2021.

[18] V. Vishal. Python measure the execution time of a program, Feb 2022.

[19] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 765–777, 2020.

[20] F. Wilcoxon. *Individual comparisons by ranking methods*. Springer, 1992.

[21] Wix.com. Detox. `https://github.com/wix/Detox`, 2023.

[22] M. Zalewski. American fuzzy lop 2.52b, Jul 2017.

[23] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Code coverage. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. Retrieved 2023-01-07 13:54:15+01:00.

[24] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Mutation-based fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. Retrieved 2023-01-07 13:54:15+01:00.

[25] B. Zorn, P. Hilfinger, et al. A memory allocation profiler for c and lisp programs. In *Proceedings of the Summer USENIX Conference*, pages 223–237. USENIX Association Berkeley, CA, USA, 1988.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 4. Juli 2023