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

Multifile Fuzzing von Office Open XML Workbooks

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Lukas Jünemann
geboren am: 30.07.1995
geboren in: Berlin

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Timo Kehrer

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

Inhaltsverzeichnis

1 Einführung

Kernaspekt dieser Arbeit ist das Testen von Anwendungen, welche zur Verarbeitung Office Open XML Dokumenten dienen. Solche Anwendungen finden in Privathaushalten und Unternehmen große Verwendung, in Form von Tabellenkalkulationen oder Textverarbeitung. Zur Realisierung dieser Tests wird Fuzzing verwendet. Der Abschnitt „2 Grundlagen“ beschäftigt sich näher mit verschiedenen Facetten von Software-Tests, Fuzzing und Office Open XML Dokumenten. Im Rahmen der Ausarbeitung entstand ein Fuzzing-Werkzeug, dessen Architektur-Entscheidungen unter „3 Architektur“ dokumentiert sind. Mithilfe der, unter „4 Realisierung“ beschriebenen Software, wurden zwei Anwendungen zur Verarbeitung von Office Open XML Workbooks in verschiedene Fuzzing-Kampagnen getestet. Die Beschreibung und Auswertung dieser Kampagnen, verbunden mit drei Forschungsfragen, ist unter „5 Auswertung“ zu finden.

1.1 Motivation

Anwendungssammlungen wie „Microsoft Office ™“, „Apache OpenOffice ™“ oder auch Online-Lösungen wie „Google Docs ™“ gehören zu einer der am weitesten verbreiteten Anwendungskategorien, den „Office-Suites“. Microsoft Office hatte schon vor der Corona-Pandemie 180 Millionen monatliche Nutzer, mit seit Jahren steigender Tendenz [Red19]. Diese Anwendungen sind Teil der schulischen Ausbildung vieler Menschen, gelten als elementare Aspekte der digitalen Selbstermächtigung und sind auch in der Arbeitswelt unverzichtbar. Die drei Kernanwendungen sind: Textverarbeitungen, Tabellenkalkulationen und Präsentationsprogramme. Tabellenkalkulationen dienen zum strukturierten Ablegen von Berechnungsdaten und Formeln. In vielen Fällen sind diese Daten und Formeln Teil des geöffneten Dokuments, können jedoch, abhängig von Dateiformat und Anwendung, auch dynamisch von anderen Quellen nachgeladen werden.

Ein standardisiertes Dateiformat ist der Office Open XML / ECMA-376 / ISO/IEC 29500:2008 Standard. Dokumente dieses Standards tragen unter anderem die Dateiendungen `.docx`, `.xlsx` oder `.pptx`. Sie sind seit 2008 das Standard-Format der am weitesten verbreiteten Office-Suites [Mic08]. Aufgrund der weiten Verbreitung von Office-Anwendungen und -Dateiformaten, stellen diese ein vielversprechendes Ziel für Software-Tests und -Validierung von Außen dar. Ziel dieser Arbeit ist es, die Resilienz gegen fehlerhafte oder bösartige Eingaben in Tabellenkalkulationsanwendungen näher zu untersuchen.

Ansätze zum Fuzzen einzelner Dateien existieren schon seit den frühen 2000ern [SGA07, Part 2 Targets and Automation, Kapitel 2 What is Fuzzing?, History of Fuzzing]. Das Durchführen von Fuzzing-Tests mit mehreren Dateien ist ein wenig

untersuchter Ansatz, der im Rahmen dieser Arbeit umfangreich beleuchtet werden soll.

1.2 Aufgabenstellung

Die zu erstellende Software soll bestehende Möglichkeiten für Black-Box Tests von Tabellenkalkulationen erweitern. Ein wichtiges Kriterium wird dabei die Plattform- und damit auch Anwendungsunabhängigkeit sein, um Back-to-Back Tests realisieren zu können. Es soll möglich sein, die Test-Routinen, den Datei-Generator und auch ihre Interfaces ohne umfangreiche Implementierungskenntnisse zu nutzen.

Zu Erstellen ist ein Fuzzing-Generator, dessen erzeugte Dateien und Dateigraphen gleichermaßen gegen verschiedene Anwendungen getestet werden können. Interessante Testergebnisse – markiert durch Fehlermeldungen, Timeouts und Stacktraces – sollen strukturiert abgelegt und ausgewertet werden können. Performance-Metriken der einzelnen instrumentierten Anwendungen werden während der Tests erfasst.

Die Ergebnisse verschiedener Office-Suite Tabellenkalkulationen sind in dieser Arbeit zu dokumentieren und, soweit möglich, zu vergleichen. Sollten während der Bearbeitung Bugs und anderes Fehlverhalten von instrumentierten Anwendungen festgestellt werden, wird dies an die korrespondierenden Entwickler gemeldet.

Die Versuchsauswertung soll sich auf drei Forschungsfragen konzentrieren:

RQ1: Kann Multifile Fuzzing mehr Exception-Types erzeugen als das Fuzzen einzelner Dateien?

RQ2: Können die beiden SUT mit diesem Setup vergleichbare Back-to-Back Ergebnisse erzielen?

RQ3: Welche Rückschlüsse können aus unterschiedlich leistungsstarken Systemen mit der selben SUT gezogen werden?

2 Grundlagen

Das folgende Kapitel dient zur Vermittlung grundlegenden Konzepte für die Bearbeitung der Aufgabenstellung. Die Abschnitte bauen teilweise aufeinander auf und sollen Erklärungen liefern, die zum Verständnis der Kapitel „3 Architektur“ und „4 Realisierung“ von Bedeutung sind.

2.1 Software-Testing

Softwareentwicklung ist häufig ein explorativer, kollektiver Prozess, welcher von Menschen im kulturellen Kontext einer Organisation stattfindet [Mit+16, Misguided evidence leading to bias]. Dabei kann eine Vielzahl von Fehlern auftreten, beginnend bei fehlerhaften Spezifikationen, bis hin zu mangelndem Fachwissen oder der Nutzung einer unangemessenen Architektur. Verschiedene Maßnahmen der Software-Verifikation sind in der Lage einem Teil dieser Fehler zu begegnen oder diesen sogar vorzubeugen. Durch entsprechende Abstraktion und umfangreiche dynamische Tests ist es beispielsweise möglich, selbst nach umfangreichen Änderungen, das erwartete Verhalten einer Software zu überprüfen.

Neben statischen Code-Analysen oder formaler Verifikation, werden in der modernen Softwareentwicklung dynamische Tests in Form von Unit-, Integration- und System-Tests verwendet, um bereits im Entwicklungsprozesses das korrekte Verhalten, besonders in Randfällen, sicherzustellen [But09, 9.3 Automatic Builds – Static Analysis]. Zu diesem Zweck werden verschiedene Testfälle entworfen, welche die Schnittstelle einer einzelnen Klasse (Unit-Test), eines Moduls (Integration-Test) oder ganzer Anwendungen und Bibliotheken (System-Test) auf erwartetes Verhalten testen [Kan99, 3 Test types and their place in the softrware development process]. Schlagen nach Änderungen an Teilen einer Software einzelne Tests fehl, deutet dies darauf hin, dass das erwartete Verhalten nicht erfüllt wird.

Im Folgenden zeigt ein Beispiel eine sehr einfache Utility-Klasse, welche Additions- und Divisionsfunktionalität bereitstellt. Ein zu dieser Klasse passender Unit-Test würde beispielsweise die Addition sehr großer oder negativer Zahlen testen, was einen Randfall des Parameterraums darstellt. Die Division durch null ist ein Randfall, der unterschiedlich gehandhabt werden kann. So kann ein Unit-Test dazu beitragen, das erwartete Verhalten zu dokumentieren. Sollte sich das Verhalten durch spätere Modifikation des Funktionsinhalts ändern, wird der Test fehlschlagen. Dies kann von Bedeutung sein, da sich Nutzer einer Funktion eventuell auf das Verhalten der geworfenen Exception verlassen, um einen solchen Divisor speziell zu behandeln.

```java
public final class MathUtil {

    private MathUtil() {}

    public static int add(int a, int b) {
        return a + b;
    }

    public static int divide(int dividend, int divisor) {
        return dividend / divisor;
    }
}
```

Codeauszug 1: `MathUtil` – Eine beispielhafte Utility-Klasse für numerische Operationen

```java
public class MathUtilTest {

    @Test
    public void testAdd() {
        assertEquals(-1, MathUtil.add(Integer.MIN_VALUE,
                                      Integer.MAX_VALUE));
    }

    @Test
    public void testAddOverflow() {
        assertEquals(-2, MathUtil.add(Integer.MAX_VALUE,
                                      Integer.MAX_VALUE));
    }

    @Test(expected = ArithmeticException.class)
    public void testDivideByZero() {
        MathUtil.divide(1, 0);
    }
}
```

Codeauszug 2: `MathUtilTest` – Ein Unit-Test mit Randfällen für verschiedene Funktionen der `MathUtil` Klasse

Kennt ein Software-Entwickler den zu testenden Code sehr gut oder verfügt über umfangreiches Fachwissen, ist es häufig trivial passende Tests zu entwerfen. Der Umfang und die Qualität der Tests sind dabei durch äußere Parameter, wie die Erfahrung des Entwicklers oder wirtschaftliche Güter, wie Zeit oder personelle Ressourcen, beschränkt. Bedingt durch diese Einschränkungen werden oft Metriken, wie

Line-, Branch- oder Class-Coverage als Maße für eine hinreichende Test-Abdeckung genutzt. Um diese Metriken zu erfassen, wird während der Testausführung mittels instrumentierender Software ermittelt, welche Teile des Quelltextes durchlaufen wurden. Dabei ist unerheblich, ob diese Teile der Software für die Testausführung von Relevanz oder überhaupt Ziel des Tests waren. Dadurch sind einfache Coverage-Metriken mit einer gewissen Unschärfe versehen.

Sogenanntes „Black-Box Testing“ beschreibt die Software-Verifikation mittels dynamischer Tests, ohne Kenntnisse einer konkreten Implementierung jenseits der Schnittstelle [SGA07, Part 1 Background, Kapitel 1 Vunarability Discovery Methods, Black Box Testing]. Es kann dazu genutzt werden, um unvoreingenommen und lediglich anhand gewisser Test-Spezifikationen, die korrekte Funktion einer Software sicherzustellen. Diese Art der Test-Entwicklung wird oft ergänzend zu anderen dynamischen Tests genutzt. Da die Parameterräume vieler Schnittstellen bewusst beschränkt sind, ist es so nicht möglich, jeden Teil einer Software zu testen. Jedoch besteht über Instrumentierung die Möglichkeit zur Messung, ob durch einen Test neue Programmteile, in Form von Bytecode, erreicht wurden. Außerdem ist es möglich, andere Anforderungen, abseits eines korrekten Test-Ergebnisses, an einen Test zu stellen, wie beispielsweise eine Beschränkung der Ausführungsdauer oder des allozierten Speichers. Während der Testdurchführung sind lediglich die Parameter und die Beobachtungen bekannt, nicht jedoch die innere Funktionalität der Anwendung. Da kommerzielle Software häufig als Kompilat ohne Quelltext vorliegt, ist dies ein möglicher Testansatz zur Untersuchung kommerzieller, proprietärer Software.

Diese Kriterien erlauben es auch unterschiedliche „Software under Test (SUT)“, also untersuchte Anwendungen, miteinander zu vergleichen. Dies kann aus verschiedenen Gründe notwendig sein, etwa weil gleicher Funktionalitäten in unterschiedlichen Anwendungen verglichen werden sollen oder um konkrete Verbesserungen nach umfangreichen Änderung quantifizierbar zu machen. Das Verwenden identischer Testparameter zum Testen unterschiedlicher Software wird als „Back-to-Back Testing“ bezeichnet [Vou88, 1. Introduction]. Es ist durch umfangreiche Tests möglich, Rückschlüsse auf Fehlverhalten einer Implementierung zu ziehen. Diese Rückschlüsse können dabei helfen, gezielter Randfälle mit eventuell unerwünschtem Verhalten zu identifizieren [Vou88, 5. Summary]. Das Wissen über solche Randfälle kann während der Integration der SUT, beispielsweise im Zuge von Automatisierungen, von Bedeutung sein.

Dynamische Software-Tests mit statischen Daten sind lediglich ein Teil der Software-Validierung und können die Korrektheit einer SUT für ausgewählte Parameter überprüfen. Da die Parameterräume von Schnittstellen häufig zu umfangreich sind, um alle Parameterkombinationen zu überprüfen, werden Test oft auf Randfälle begrenzt. Die Qualität und Anzahl sinnvoller Tests mit statischen Daten ist dabei von äußeren Faktoren abhängig, was sich insbesondere bei umfangreicheren

Implementierungen zeigt. Neben statischen Daten ist es möglich, Tests mit dynamischen oder zufälligen Daten durchzuführen, womit sich der folgende Abschnitt beschäftigt.

2.2 Fuzzing

Fuzzing kann ein Ansatz sein, um den Einschränkungen dynamischer Software-Tests mit statischen Daten zu begegnen. Anstelle statischer Daten erzeugt ein Fuzzer zufällige Parameter, um eine Zielanwendung zu testen [SGA07, Part 1 Background, Kapitel 1 Vunarability Discovery Methods, Automated Testing or Fuzzing]. So ist es möglich spezielle Randfälle eines Parameterraums oder unerwartete Parameterkombinationen auszunutzen, um Fehlerfälle gezielt zu provozieren. Diese dynamischen Tests werden wiederholt durchgeführt, wobei neue Parameter auch gezielte Mutationen vorheriger Parameter sein können. Umfangreiches Fuzzing hat sich in als Brute-Force Ansatz in der Sicherheitsforschung bewährt, jedoch kann nicht jede SUT auf gleiche Art und Weise getestet werden [Kle+18, Abstract, 1 Introduction]. Für verschiedene Anwendungszwecke haben sich verschiedene Ansätze des Fuzzing herausgebildet, von denen ausgewählte im Folgenden vorgestellt werden.

2.2.1 Mutation-based Fuzzing

Abweichend vom naiven Ansatz völlig zufälliger Parameter existiert die Methode des mutation-based Fuzzing. Hierfür wird eine Menge valider Input-Parameter benutzt, welche anschließend auf verschiedene Arten gezielt oder zufällig mutiert werden [SGA07, Part 2 Targets and Automation, Kapitel 11 File Format Fuzzing, Brute Force or Mutation-Based Fuzzing]. Dies kann den Implementierungsaufwand für Fuzzing-Generatoren maßgeblich reduzieren. Instrumented Fuzzer können die erfassten Metriken einer instrumentierten Anwendung nutzen, um Parameter für Tests gezielt zu mutieren und beispielsweise neue Pfade im Programmablauf einer SUT zu erreichen. Einer der bekanntesten instrumented Fuzzer, „american fuzzy lop (AFL)“, nutzt das Feedback der instrumentierten Software, um Inputs mit verschiedenen Strategien wie bitweisen oder arithmetischen Operationen zu verändern [Zal21]. Durch die Instrumentierung wird die Branch-Coverage gemessen. Mutationen gelten demnach als erfolgreich, wenn neue Code-Bereiche innerhalb der instrumentierten Anwendung erreicht werden. Als Binary-Fuzzer erzeugt AFL dabei selbst Byteströme welche dann in eine Zielsemantik überführt werden können.

In dem Studienprojekt, welches dieser Arbeit zugrunde liegt, wurde AFL beispielsweise als Zufallsquelle genutzt, um mithilfe von grammar-based Fuzzing, also aufbauend auf bestehenden Metamodellen, zufällige SVG-Dateien zu erzeugen [Jün22, 4.1 Versuchsaufbau]. Die gemessene Branch-Coverage wurde genutzt, um

den Bytestrom zu mutieren, der dem Dokument-Generator als Quelle des Zufalls zur Generierung dient.

Neben der Mutation der Zufallsquelle für Fuzzing-Generatoren kann mutation-based Fuzzing auch direkt mit Parametern genutzt werden, deren Inhalt eine semantische und syntaktische Bedeutung haben. Dabei gibt es jedoch einige Herausforderungen zu beachten. Verweisen innerhalb von Grammatiken beispielsweise Objekte aufeinander und ein solcher Verweis wird beliebig mutiert, können semantisch fehlerhafte Parameter entstehen und die Programmausführung der SUT verhindern. Prüfsummen können einem Ansatz mit reinem mutation-based Fuzzing ebenfalls im Weg stehen [SGA07, Part 2 Targets and Automation, Kapitel 11 File Format Fuzzing, Brute Force or Mutation-Based Fuzzing].

2.2.2 Semantisches Fuzzing

Es existieren zahlreiche Dateiformate, welche nicht in menschenlesbarer Form vorliegen, sondern abstrakt als zufälliger Bytestrom betrachten werden können, wie Medienformate für Bild, Ton und Video oder Bytecode von dynamischen Bibliotheken und Executables. Jedes dieser Dateiformate folgt zwar einer Spezifikation, wodurch die Bytefolgen nicht wirklich zufällig sind, jedoch ist es für primitive Fuzzing-Tests ausreichend von quasi-zufälligen Bytefolgen auszugehen. Eine SUT interpretiert diesen Bytestrom der jeweiligen Spezifikation folgend. Dabei auftretendes Fehlverhalten, in Form von z.B. Speicherüberläufen oder Endlosschleifen, kann unter anderem auf kritische Sicherheitslücken hindeuten.

Viele Anwendungen erwarten jedoch komplexe Inputs, die einer bestimmten Struktur oder Grammatik folgen [Pad+19, 1 Introduction]. Teile dieser Inputs haben einen semantischen Wert und werden von einer Anwendung nach einer syntaktischen Überprüfung in höhere Datenstrukturen überführt. Diese Datenstrukturen werden anschließend in inhaltlichen Zusammenhang gebracht und mit der inneren Logik der Anwendung verarbeitet. Beispiele für diese strukturierten Daten sind beispielsweise XML-Dateien in Form von SVGs oder Maven „Project Object Model (POM)“. Dieses Prinzip lässt sich in Form von grammar-based Fuzzing verallgemeinert auf alle strukturierten Daten übertragen wie Quelltexte, Serialisierungen wie INI- oder JSON-Dateien oder Assembler- und Bytecode. Zum Erzeugen dieser strukturierten Daten werden Generatoren genutzt.

Ein Fuzzing-Generator muss in der Lage sein, mithilfe einer externen Zufallsquelle, beispielsweise einem „Pseudorandom Number Generator (PRNG)“, Inputs zu erzeugen die den engen syntaktischen und semantischen Anforderungen der SUT folgen [Pad+19, 1 Introduction]. Der Fuzzing-Generator kann dabei so implementiert werden, dass er gezielt bestimmte Aspekte eines Zielformats berücksichtigt, beispielsweise SVG-Animationen im Kontext eines SVG-Generators [Jün22, 3.4

Konkrete Implementierung]. Die iterative Natur des Fuzzing erlaubt auch probabilistische Ansätze, um mit gewisser Wahrscheinlichkeit gegen Grammatiken zu verstoßen.

Neben Metamodellen in Form von Grammatiken oder Schemata, wie „XML Schema Definition (XSD)“ oder ANTLR-Grammatiken, können auch Referenz-Implementierungen für das Erzeugen von semantisch sensiblen Testdaten genutzt werden. Im Rahmen dieser Arbeit wird das Fuzzing von Office Open XML Workbooks gezeigt. Dazu wird eine Referenz-Implementierung des „Office Open XML“-Standard, „Apache POI“, genutzt, um semantisch korrekte Workbooks zu erzeugen.

2.2.3 File Fuzzing

Im Allgemeinen existieren verschiedene Wege, um Parameter an eine SUT zu reichen. Neben der direkten Nutzung einer API beim White-box Testing existieren oft „Standard Development Kits (SDKs)“, um Anwendungen untereinander integrieren zu können. APIs und SDKs erlauben es programmatische Test-Routinen zu erzeugen, die mit Datenstrukturen arbeiten, welche der SUT ohne Umwege übergeben werden können. Anwendungen bieten abseits dieser Möglichkeiten die regulären Schnittstellen zur Interprozesskommunikation in Form von Pipes, Kommandozeilen-Parametern und Dateien. File-Fuzzing beschäftigt sich mit dem Fuzzing von Anwendung über Dateiinhalte und -Formate.

Dabei bezieht sich das Fuzzing beim Generieren der Inputs, in der Regel nicht auf die Metadaten einer Datei, wie Name oder Attribute, sondern auf Dateiinhalte. Dies vereint verschiedene Herausforderungen aus dem mutation-based und semantischen Fuzzing. So können Fuzzing-Test-Dateien aus validen, bereits existierenden, Dateien abgeleitet, oder mithilfe von Grammatiken und Spezifikationen, generiert werden. Wie bereits unter „2.2.2 Semantisches Fuzzing“ beschrieben können Generatoren gezielt auf einzelne Teile einer Spezifikation abzielen. Datei-Generatoren sind über Parameter konfigurierbar, um bestimmte Dateieigenschaften zu erzielen [SGA07, Part 2 Targets and Automation, Kapitel 11 File Format Fuzzing, Intelligent Brute Force or Generation-based Fuzzing]. Diese Wandelbarkeit erleichtert das Identifizieren interessanter Parameterräume zur Dateigenerierung und für anschließende Fuzzing-Kampagnen.

Beim Fuzzing einzelner Dateien ist es für viele Testfälle ausreichend, die Testdaten als Bytestrom in den Standard-Input oder andere im Arbeitsspeicher befindliche Datenstruktur abzubilden. Dadurch kann die Nutzung des Dateisystems umgangen werden, was in vielen Anwendungsfällen Performance-Vorteile hat. Da im Zuge der Ausarbeitung Multifile Fuzzing untersucht wird, ist der Einbezug des Dateisystems unumgänglich, jedoch kann dieses Dateisystem eine RAM-Disk sein. Multifile Fuzzing beschreibt den Vorgang des Fuzzing mehrerer aufeinander verweisender

Dateien. Viele Anwendungen nutzen für Verweise auf Dokumente in anderen Dateien Notationen, die einen relativen oder absoluten Dateipfad beinhalten. Neben der Nutzung von URLs existieren viele anwendungsspezifische Notationen. Diese bestehen in der Regel zum einem aus der Abbildung des Dateipfads, zum anderen aus der Adressierung einer Struktur innerhalb der referenzierten Datei. Beispiele dafür sind XLink-Referenzen in XML-basierten Dateiformaten oder externe Referenzen in Microsoft Excel Office Open XML Dokumenten [Moz23] [Mic21a]. Insbesondere beim Back-to-Back Testing verschiedener Anwendungen kann das Erzeugen dieser Referenzen eine Herausforderung sein, da diese anwendungs- und plattformspezifisch sein können [Mic21a] [Gir20].

2.3 Office Open XML

Die für diese Arbeit namensgebenden Office Open XML Dokumente folgen der Spezifikation „ECMA-376“, welche später im Standard „ISO/IEC 29500:2008“ standardisiert wurde. Grundlage des Standardisierungsprozesses war eine Präsentation Microsofts beim „Technical Committee 45 (TC45)“ des „European Computer Manufacturers Association (ECMA)“ am 08. Dezember 2005. Die weite Adaption von XML zur Persistenz strukturierter und unstrukturierter Daten, ergänzt um den großen Bedarf von Nutzern für interoperable Office-Dokumente, war eine der Motivationen für diesen Prozess [Pao+05].

Da Microsoft zu diesem Zeitpunkt eine der verbreitetsten Office-Suites vertrieb, wollten sie, aus dieser Anwendungssammlung heraus, die notwendigen Features des offenen Standards abstecken [Han02]. Dieser sollte alle Fähigkeiten der historischen, proprietären Office-Dateiformate unterstützen und so als rückwärts-kompatibles Standardformat dienen. Neben Microsoft war auch Novell, die mit „WordPerfect“ jahrzehntelange Erfahrung mit Textverarbeitungsdokumenten haben, Teil des TC45 [Int06] [Pao+05].

Die aktuellste Version von 2021 ist in vier Abschnitte eingeteilt:

1. Fundamentals And Markup Language Reference
2. Open Packaging Conventions
3. Markup Compatibility and Extensibility
4. Transitional Migration Features

Besonders der erste Abschnitt ist für diese Ausarbeitung von Bedeutung. Die im Namen dieser Arbeit beschriebenen Workbooks sind demnach eine Sammlung mehrerer Worksheets [Int16, 12.1 Glossary of SpreadsheetML-Specific Terms]. Diese Sheets sind zweidimensionale Gitter aus Zellen, welche die eigentlichen, numerischen oder textuellen, Daten beinhalten. Diese textuellen Daten können in einer speziellen, strukturierter Notation eine Formel, also eine Berechnungsvorgabe

für den Zellinhalt, darstellen. Neben ihrem Inhalt besitzen Zellen auch andere Eigenschaften wie Textformatierung, -ausrichtung, -farbe oder einen Rahmen zur visuellen Abgrenzung von anderen Zellen.

Ein Office Open XML Dokument selbst ist ein herkömmliches ZIP-Archiv, dessen einzelne Bestandteile in strukturierter Art geordnet sind [Int16, 8.2 Packages and Parts]. Dabei wird unterschieden zwischen „WordprocessingML“, „SpreadsheetML“ und „PresentationML“ Dokumenten. Ein Dokument besteht aus sogenannten Parts, welche als gepackte XML-Dokumente innerhalb des Archivs vorliegen.

Die Bestandteile der, für diese Arbeit zentralen, SpreadsheetML-Dokumente sind einzelne Worksheet-Descriptor [Int16, 8.5 SpreadsheetML]. Jedes Sheet beinhaltet die Informationen zu seinen Zellen. Andere mögliche Bestandteile eines Workbooks sind gefilterte Tabellen, Bilder oder Kommentare. Zugänglich wird das Dokument durch einen zentralen Workbook-Part, der nicht alle Informationen beinhaltet, sondern auf die andere Parts verweist.

Der Spezifikation nach ist eine Zelle ein XML-Element mit dem Namen `c` innerhalb eines Worksheet-Descriptor [Int16, 18.3.1.4 c (Cell)]. Neben dem Referenz-Attribut `r`, dass die Position einer Zelle innerhalb eines Worksheets angibt, ist vor allem das „Cell Data Type“-Attribut `t` von Bedeutung.

Eine Zelle kann folgende Datentypen annehmen: [Int16, 18.18.11 ST_CellType (Cell Type)]

- `b` – Ein boolescher Wahrheitswert
- `d` – Zur Darstellung eines Datums im ISO 8601 Format
- `e` – Der Datentyp für Fehler
- `inlineStr` – Dieser Datentyp zeigt an, dass die Zelle einen Text mitsamt Formatierung im Wert-Bereich des XML-Elements beinhaltet. Durch diesen Datentyp können Wertverweise durch andere Zellen-Attribute überschrieben werden.
- `n` – Eine Zahl
- `s` – Eine Referenz auf einen geteilten String innerhalb des Workbooks
- `str` – Eine Formel

3 Architektur

Das folgende Kapitel soll ein Überblick über grundsätzliche Annahmen und Herausforderungen der Entwicklung der, begleitend zu dieser Arbeit entstandenen, Software geben. Es handelt sich dabei um Überlegungen während der Entwicklung, deren konkrete Implementierung im Kapitel „4 Realisierung“ dokumentiert wird. In diesem Kapitel wird ein Experiment in eine festgelegte Anzahl von Versuchen eingeteilt. Unterschiedliche Versuche desselben Experiments beziehen sich demnach auf unterschiedliche Dokumente die, innerhalb desselben Experiments, vom selben Generator mit denselben Einstellungen generiert wurden.

3.1 Fuzzing-Generator

Im Studienprojekt wurden die Möglichkeiten zur Abstraktion von Dokument-Referenzen bereits beleuchtet [Jün22, 2.1.2 Referenzen in Dateien]. Der praktikabelste Ansatz war die Nutzung eines sogenannten „Linkpools“. Ein Linkpool ist eine Menge von Referenzen aus denen gezielt oder zufällig gewählt werden kann, um sie während der Dokumentgenerierung zu nutzen. Dieser Linkpool kann entweder statisch oder dynamisch gefüllt werden. Dies bedeutet, dass entweder vor der Generierung eines Dokuments eine feste Menge an referenzierbaren Objekten und Dokumenten existiert, oder nach und nach generierte Dokumente Teil des Linkpools werden können. Letzteres hat die Eigenschaft, dass die durchschnittliche Tiefe von zufälligen referenzieren Dokumenten immer weiter zunimmt, wohingegen ersteres eine feste oder maximale Dokument-Tiefe garantiert. Auf die Bedeutung der Dokument-Tiefe wird unter „4.2.1 Fuzzing-Generator“ näher eingegangen.

Da im Studienprojekt JQF die Grundlage der entwickelten Software bildete, wurde ein `junit-quickcheck` Generator erstellt [Hol20]. Diese Art Generatoren sind zum Erzeugen zufälliger Parameter geeignet und haben nicht zwingend einen internen Zustand. Die Nutzung eines Linkpool-basierten Generators, bringt neben einem Zustand, auch einen Lebenszyklus mit sich. Dieser Lebenszyklus beginnt in einem Vorbereitungszustand mit leerem Linkpool und geht in den eigentlichen Generierungszustand mit initialem Linkpool über. So kann eine initiale Menge an Referenzen entweder vor der Generierung der ersten Dokumente erzeugt werden oder aus bestehenden Dokumenten extrahiert und indexiert werden. Um den Generierungsprozess zwischen Dateien unterbrechen und später fortsetzen zu können, muss es möglich sein, die Referenzen zwischen Dokumenten zu serialisieren. So kann zu einem späteren Zeitpunkt ein neuer Generator neue Dokumente, unter Nutzung des wiederhergestellten Linkpools, fortsetzen.

Die Anforderung an einen Generator für Office Open XML Workbooks erzeugen problemspezifische Architekturherausforderungen. Es existieren sieben Datentypen

mit unterschiedlichen Randfällen (siehe „2.3 Office Open XML“). Der Generator muss in der Lage sein einen zufälligen Datentypen für eine Zelle zu wählen und für jeden dieser Datentypen zufällige Werte zu erzeugen. Ein Ansatz des grammar-based Fuzzing von Formeln sollte vermieden werden, da dies außerhalb der Aufgabenstellung dieser Arbeit liegt. Zukünftig könnte der Generator um diesen Aspekt erweitert werden. Eine weitere Herausforderung ist das Erstellen von Formeln, um Referenzen zu anderen Workbooks konkret abzubilden. Die Spezifikation von Zellenreferenzen legt die Nutzung absoluter Dateipfade nahe [Int16, 18.17.2.3 Cell References]. Da die Dokumente jedoch später auf unterschiedlichen Clients unterschiedlicher Plattformen geöffnet werden, muss eine Möglichkeit zur Nutzung relativer Dateipfade im selben Arbeitsverzeichnis geschaffen werden. Referenzen aus dem Linkpool müssen vom Generator entweder dateisystem- und anwendungsunabhängig notiert werden oder später durch die versuchsausführenden Clients umgeformt werden.

3.2 Plattformunabhängige Instrumentierung

Zum Black-Box Testing ganzer Anwendung muss eine Möglichkeit zur Instrumentierung bestehen. Diese Instrumentierung muss verschiedene Anforderungen erfüllen, um während der Versuchsdurchführung sinnvolle und auswertbare Metriken zur erzeugen. So ist es beispielsweise möglich Anwendung lediglich über die Kommandozeile zu instrumentieren, jedoch sind die erfassbaren Metriken dann auf die Standardausgaben und Exit-Codes beschränkt. Doch nicht jede Anwendung nutzt unterschiedliche Exit-Codes, um verschiedene Fehler zu symbolisieren. Auch Standardausgaben können im Fall schwerwiegender Fehler einfach abbrechen. Zur konkreten Instrumentierung von Anwendungen kann es daher notwendig sein auf ein, durch den Hersteller bereitgestelltes, „Standard Development Kit (SDK)“, oder eine andere Art informationsreicher Interprozesskommunikation, zurückzugreifen.

Bei der plattformunabhängigen Instrumentierung werden zwei Grundannahmen miteinander verbunden. Die erste Annahme ist, dass sich die Durchführung von Versuchen auf verschiedenen Plattformen mit verschiedenen Anwendungen mit denselben algorithmischen Schritten beschreiben lässt. Die zweite Annahme ist, dass es eine Menge von Metriken gibt, die während oder nach der Durchführung eines Versuches, plattformunabhängig erfasst werden kann und Schlüsse über den inneren Zustand der SUT zulässt.

Mit der SUT werden im Wesentlichen zwei Vorgänge ausgeführt, die einer algorithmischen Abbildung bedürfen. Der eine ist die eigentliche Versuchsdurchführung mit einem Versuchsdokument und etwaigen weiteren Metadaten zur Datei, die zur Bestimmung der Ergebnismetriken notwendig sein können. Da jedes Experiment ein konfiguriertes Timeout hat, also eine maximale Dauer für jeden einzelnen Versuch, muss dieser Vorgang unterbrechbar sein. Diese Ausführung sollte wiederholbar

sein, falls der Fehlschlag eines Versuchs auf äußere Umstände, wie fehlerhafte Interprozesskommunikation zurückzuführen ist. Dies ist der andere Vorgang, welcher abgebildet werden muss. Vor der eigentlichen Durchführung muss die SUT in der Regel vorbereitet werden. So kann eine Anwendung bereits gestartet werden, um die Testzeiten des ersten Dokumentes nicht zu verfälschen und eine Kommunikation aufzubauen. Nach einem Crash durch ein vorheriges Dokument muss es Möglichkeiten geben, im Zuge der Vorbereitung auch alte Programmzustände der SUT zu bereinigen, um zukünftige Ergebnisse nicht zu verfälschen.

Die im Verlauf des Versuchs erfassten Metriken sind im Black-Box Testing durch die Schnittstelle zur SUT limitiert. Abseits von den Metriken, welche durch die anwendungsspezifische Instrumentierung selbst erfasst werden, kann es sinnvoll sein, allgemeine System-Metriken zu erfassen. So kann eine plötzliche hohe CPU- oder Speicher-Auslastung auf Endlosschleifen hindeuten. Diese Art von Metriken kann auf jedem System erfasst werden und ist dadurch anwendungs- und plattformunabhängig.

3.3 Kommunikationsmodell und Zustandsverwaltung

Das Studienprojekt nutzte JQF, um Objekte für parametrisierte Unit-Tests zu erzeugen [Jün22, 4.1 Versuchsaufbau]. Diese enge Kopplung zwischen Generator und SUT in derselben JVM, hat den Vorteil, quasi beliebig komplexe Test-Inputs erzeugen zu können, die als höhere Datenstrukturen einen eigenen inneren Zustand besitzen können. Der Unit-Test kann diese Inputs beliebig umformen, um sie für die SUT konsumierbar aufzubereiten.

Der plattformunabhängige Entwurf der im Rahmen dieser Arbeit zu erstellenden Software erfordert implizit ein verteiltes System. Wie unter „1.2 Aufgabenstellung“ beschrieben, sollen dieselben Dokumente und Dokumentgraphen in Form von Back-to-Back Testing zur Untersuchung verschiedener Anwendungen auf verschiedenen Plattformen genutzt werden. Dies legt ein verteiltes System nahe, in dem ein Dokument-Generator im Zentrum mehrerer Clients steht, welche die eigentliche Testdurchführung vornehmen. Diese Form der Systemarchitektur erzeugt eine Reihe weiterer Herausforderungen. Da in diesem Fall der Generator und die SUT nicht nur logisch, sondern ebenfalls räumlich getrennt sind, ist die mögliche Komplexität der Test-Objekte maßgeblich vom gewählten Kommunikationsmodell abhängig. Mehrere Clients können gleichzeitig unterschiedliche oder auch dieselben Dokumente testen, weshalb es nicht mehr ausreicht Test-Objekte nacheinander zu generieren. Clients werden auf verschiedenen Plattformen, mit unterschiedlicher SUT unterschiedliche Ausführungszeiten für einzelne Dokumente erzielen. Dadurch wird es notwendig, den Fortschritt eines Clients als Zustand aufzubewahren. Dies kann auf dem Client oder dem Dokument-Server geschehen, was unterschiedliche Architektur-Folgen hat.

3.3.1 Zustandsaufbewahrung

Wenn Clients ihren Experiment-Fortschritt selbst aufbewahren, müssen sie zur Auswahl des nächsten Versuches ein Modell des gesamten Experiments aufbewahren, um zu entscheiden, welche Versuche noch nicht durchgeführt sind. In einer alternativen Variante müssen sie die Menge aller bereits durchgeführten Versuche eines Experiments an den Dokument-Server übermitteln, damit dieser den nächsten Versuch bestimmt. Ersteres hat den Vorteil, dass die Ergebnisse der Versuche separat auf jedem Client aufbewahrt werden würde. Dieser umfangreiche Zustand wäre so auf mehrere einzelne Systemteilnehmer verteilt, was ihn unter Umständen besser handhabbar machen kann. Nachteile dieser Implementierung sind zum einen die redundante Aufbewahrung der Experimente mit den einzelnen Versuchs-Abbildungen. Diese können bei umfangreichen Experimenten tausende von Versuchen mit tausende von Dokumenten, beziehungsweise deren IDs, beinhalten und müssten auf jedem Client vorliegen. Zum anderen ist eine daraus folgende Konsequenz, dass Experimente und deren Versuche und Dokumente immer im Ganzen vorliegen müssen, damit ein Client mit dem Experiment beginnen kann. Es ist denkbar, ein Experiment bei dieser Art der Architektur in Batches bzw. Unterexperimente mit Dokument-Teilmengen einzuteilen, was jedoch zu weiterem Overhead führt, da die Clients nun zusätzlich die Batch-Fortschritte und die Menge aller Batches verwalten müssten.

Wird der Fortschritt auf dem Dokument-Server aufbewahrt, wird die Redundanz der vollständigen Experiment-Beschreibung vermieden. Meldet ein Client ein Ergebnis für einen Versuch, gilt dieser als abgeschlossen. Da ein Client nun nacheinander separate Versuche durchführt, ist es nicht mehr notwendig vor Beginn eines Experiments alle Versuche vorzubereiten, da Clients jeweils nur Zugriff auf den aktuell zugeteilten Versuch haben. Der Dokument-Server kann parallel zur Durchführung auf den Clients weitere Versuchsdokumente generieren. In letzter Konsequenz müssen Clients das Modell eines Experiments nicht mehr kennen, da ihnen vom Dokument-Server nur einzelne Versuche präsentiert werden. Diese Art der Zustandshaltung reduziert die auf einem Client zu haltenden Informationen auf einzelne Versuche und verlagert den Overhead gänzlich zum Dokument-Server.

3.3.2 Kommunikationsmodell

Für Client-Server Anwendungen hat sich als Interaktionsmodell „Request-Response“ etabliert. Der Vorstellung folgend, dass ein Server einen oder mehrere Dienste anbietet, kann ein Client diese gezielt anfragen. Ein Client, welcher einen gewissen Dienst nutzen möchte, sendet über einen Konnektor eine Anfrage an den Server, welche der Server ablehnen oder annehmen kann und anschließend eine entsprechende Antwort sendet [Fie00, 3.4.1 Client-Server]. Clients sind dabei aktive Komponenten,

welche einem Programmablauf folgen und Server sind reaktive Komponenten die nach entsprechender Anfrage eine gewünschte Aktion ausführen.

Für verschiedene Anwendungszwecke existieren unterschiedliche Protokolle, welche diesem Prinzip folgen. Das „Hypertext Transfer Protocol (HTTP)“ ist eines der verbreitetsten Protokolle dieser Art. Es gibt sehr wenige Einschränkungen zur Gestaltung des Inhalts einer Anfrage. Dies erlaubt es andere Protokolle in HTTP einzukapseln. Um konsistente Konnektor-Schnittstellen zu bieten, haben sich verschiedene Architekturarten für verteile Hypermedia-Systeme entwickelt, das bekannteste ist „Representational State Transfer (REST)“ [Fie00, Kapitel 5 Representational State Transfer]. REST liefert ein einheitliches und zustandsloses Interface zur Client-Server-Kommunikation über HTTP [Fie00, 5.1 Deriving REST]. Zustandslosigkeit bezieht sich dabei auf die Beziehung zwischen verschiedenen Requests. Es soll vermieden werden verschiedene Requests miteinander in Beziehung zu setzen. Dies erzwingt gleichzeitig, dass jede Anfrage alle Informationen, welche zur Beantwortung notwendig sind, bereits enthalten muss. Der Server kann intern einen globalen Zustand zur Diensterbringung besitzen. Ein einfaches Beispiel kann ein Zeitserver sein, welcher es erlaubt eine Uhrzeit über den einen uniformen REST-Request anzufragen. Die Antwort wird sich, je nach Server-Zustand, in diesem Falle der System-Uhrzeit, ändern.

Das HTTP-Protokoll ermöglicht verschiedene Arten der Serialisierung von Test-Objekten, welche in verschiedenen Notationen, wie z.B. JSON oder XML, im Rahmen der Server-Client-Kommunikation als Zustandsabbildungen über ein REST basiertes Protokoll ausgetauscht werden können. Auch ohne enge Kopplung zwischen Generator und SUT, wie im Studienprojekt, können so, trotz logischer und räumlicher Trennung, komplexe Objekte ausgetauscht werden [Jün22, 4.1 Versuchsaufbau].

4 Realisierung

Dieses Kapitel beschäftigt sich mit der zu dieser Arbeit implementierten Software. In diesem Rahmen wurde eine Software bestehend aus drei Modulen entwickelt, dem Server, dem Client und dem Server-Frontend. Im folgenden Kapitel soll jeder Part mit seinen besonderen Eigenschaften und Herausforderungen beleuchtet werden. Entscheidende Details der Implementierung sollen dabei praktisch beschrieben werden, Code-Ausschnitte dienen zur exemplarischen Veranschaulichung.

4.1 Grundlegende Technologien

Die Implementierung wurde hauptsächlich in Java 17 als multi-modulares Gradle-Projekt umgesetzt. Server und Client, sowie Server und Frontend, kommunizieren über eine REST-API miteinander.

Serverseitig ist die REST-API mit `spring-boot-starter-web` aus dem Spring-Framework umgesetzt. Weitere Spring-Komponenten sind `spring-boot-starter-data-jpa` zur Abstraktion der Datenbank-Implementierung, `spring-boot-starter-mail` zum Versand von Benachrichtigungen und `spring-boot-starter-websocket` zum Bereitstellen von Event-Daten.

Neben der Client-Implementierung existiert ein weiteres Modul welches die API bedient, das Frontend. Dieses Frontend ist in Typescript mit dem Angular-Framework realisiert und wird von Gradle, als JAR verpackt, in das Backend-Kompilat eingebettet. So wird der Spring interne Webserver genutzt, um neben der dynamischen REST-API auch die statischen Ressourcen des Frontends auszuliefern.

Das Projekt weist folgende Modul-Struktur auf:

- `mofuzz-mf-core` – Beinhaltet grundlegende Interfaces und Klassen zur Abstraktion von Generatoren, Dateien, Links und Linkpools
- `mofuzz-mf-distributed` – Sammel-Modul für die wesentlichen Komponenten des verteilten Systems
  - `mofuzz-document-client` – Der plattformunabhängige Client zur Instrumentierung der Anwendung und Ausführung der Versuche
  - `mofuzz-document-server` – Der zentrale Server zum Verwalten von Experimenten, Dateien und Ergebnissen
  - `mofuzz-document-server-frontend` – Das Frontend des Servers, erlaubt das Verwalten von Experimenten und Clients über ein Web-Interface

- `mofuzz-mf-impl` – Implementationen verschiedener Generatoren zum Fuzzen verschiedener Anwendungen, z.B. zum Erstellen der Workbooks die im Kern dieser Arbeit stehen

4.2 Dokument-Server

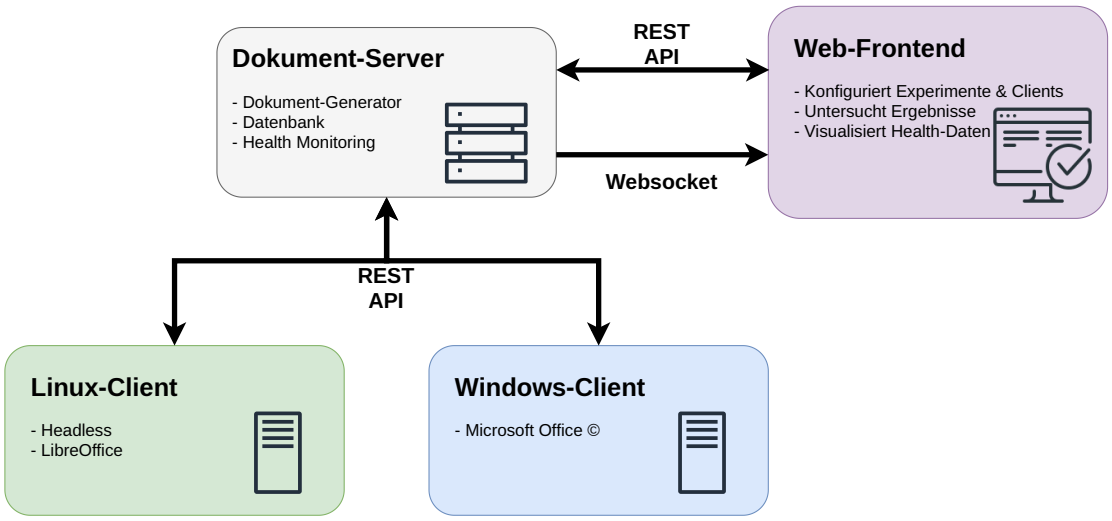


Abbildung 1: Übersicht der Schnittstellen zur Interaktion der einzelnen Komponenten

Der Dokument-Server, im Folgenden als Server bezeichnet, steht im Zentrum der Architektur. Als zentrales Bindeglied der verschiedenen Komponenten, wie Clients, Frontend, Datenbank, Generator und Datei-Persistenz, mussten viele Anforderungen miteinander vereint werden. Beispielsweise sollten die Clients selbst möglichst zustandslos sein, dies verlagert die Zustandsverwaltung auf den Server. REST-APIs mit Client-Server-Interaktion sind dem Grundgedanken nach jedoch zustandslos. Das bedeutet, dass in einer Anfrage bereits alle zur Beantwortung benötigten Informationen beinhaltet sind [Fie00, 5.1.3 Stateless]. In der implementierten Software muss ein Client lediglich einen validen, serverseitig konfigurierten Client-Identifier kennen und bei jedem Request übermitteln. Eine der Grundannahmen ist es, dass jede Komponente dieses Systems spontan ausfallen kann. Gerade beim Fuzzing großer Datenmengen ist es nicht unüblich, dass unzureichender Festplatten- oder Arbeitsspeicher zur Verfügung stehen und das Betriebssystem Prozesse unangekündigt beendet. Der Wiederherstellungsvorgang soll dann verlässlich und in absehbarer Zeit verlaufen, um das Systemziel, die Durchführung der Experimente mit der SUT, möglichst effizient zu verfolgen. So muss der Systemzustand des Servers, seiner Teilkomponenten und der Clients in einer Datenbank persistiert werden und aus dieser wiederhergestellt werden können. Es muss möglich sein die Dokumentgene-

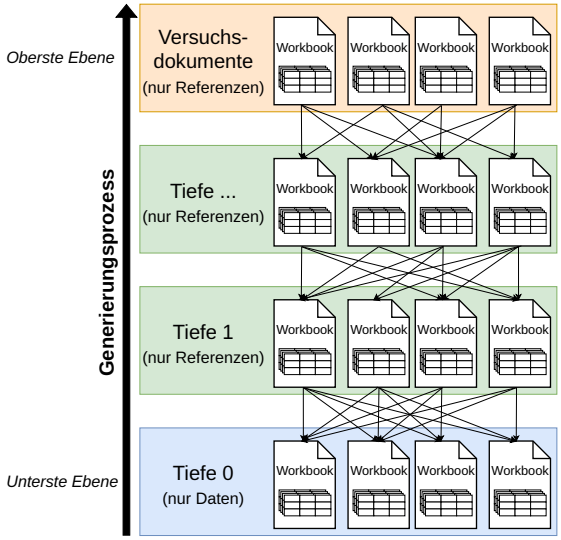


Abbildung 2: Visualisierung der Unterteilung des Generierungsprozesses in Ebenen

rierung spontan zu unterbrechen und nach einem Neustart wieder aufzugreifen, um Clients unterbrechungsarm mit Dateien zur Versuchsdurchführung zu versorgen.

4.2.1 Fuzzing-Generator

In der konkreten Implementierung wurde ein Generator mit einem statischen Linkpool von Dokumenten niedrigerer Tiefe, als die der Zieldokumente, verwendet. Tiefe bezieht sich hier auf die Anzahl an Referenzen, denen gefolgt werden muss, um innerhalb des gerichteten Dokumentgraphen vom Wurzeldokument zu den unverlinkten Basisdokumenten zu gelangen. Dazu wurden die referenzierbaren Dokumente des Linkpools Ebene für Ebene generiert: Beginnend bei der untersten Ebene, in der die Basisdokumente lediglich Daten und Formeln, jedoch keine Verweise, beinhalten, hinauf bis zu Dokumenten, deren Tiefe um eins niedriger war als die der Zieldokumente (siehe Abb. 2).

Diese Art der Generatoren ist mit der abstrakten Klasse `PoolBasedGenerator` abgebildet. Diese Klasse erbt von der Klasse `Generator` der `junit-quickcheck` Bibliothek und besitzt vier generische Typ-Parameter [Hol20]:

- `D` – Der anwendungsspezifische Dokument-Typ, der generiert werden soll, im Fall dieser Arbeit `LinkedFile`, eine Abstraktion eines verlinkenden Workbooks
- `S` – Dem Serialisierungstyp von Referenzen, der in den meisten Fällen ein String ist. Da dem Nutzer jedoch maximale Freiheitsgrade gewährt werden soll, wurde dieser Typ ebenfalls als generischer Parameter gestaltet

- `L` – Der Link-Typ muss das Interface `SerializableLink` implementieren. Diese Abstraktion erlaubt es einer Referenz so viele Metadaten zu beinhalten wie für den Anwendungsfall notwendig. Die Serialisierbarkeit wird genutzt, um Linkpools außerhalb des Generators persistieren zu können.
- `C` – Der Konfigurations-Typ des Generators muss von der Klasse `PoolBasedGeneratorConfig` erben. Diese Klasse nimmt den Link-Typ selbst als generischen Typ-Parameter und verwaltet den Linkpool des Generators. Dieser Mechanismus wurde bewusst gewählt, um eine logische Trennung der Verwaltung des Linkpools und der Generierung der Dokumente nahezulegen.

Das durch den Nutzer zu implementierende Interface wurde auf wenige Funktionen heruntergebrochen, um den Implementierungsaufwand auf das wesentliche zu beschränken und gleichzeitig eine intuitive Schnittstelle bereit zustellen. Neben dem Vorbereiten des eigentlichen Linkpools und der Generierung von Dokumenten, gibt es Funktionen zum Wiederherstellen des Linkpools aus serialisierten Links. Letzteres ist eine der wenigen Stellen, an denen das Modul `mofuzz-mf-core` die Nutzung einer Klasse vorschreibt. Die Nutzung der `File` Klasse ist an dieser Stelle vorgeschrieben. Dieser Kompromiss wurde gewählt, da das Interface der Klasse zum einen sehr umfangreich ist und auch im sehr großen Java-Ökosystem wenig Konkurrenz außerhalb der Standard-Bibliotheken begegnet. Zum anderen ist bei der Nutzung von Multifile Fuzzing die Nutzung mehrerer Dateien in einem Arbeitsverzeichnis konzeptuell sehr naheliegend und soll Nutzer, zur Strukturierung ihres Generierungsprozesses, implizit in diese Richtung leiten.

Die Vererbung der `junit-quickcheck` `Generator`-Klasse ist für diese Arbeit nicht von Bedeutung, erlaubt im Allgemeinen jedoch die Nutzung der Generatoren zur Erzeugung von parametrisierten Tests und damit das Fuzzing mittels JQF. Durch die unter „4.3.2 Plattformunabhängige Instrumentierung“ beschriebene Art der Instrumentierung kam die einfache Nutzung von JUnit-Tests nicht infrage.

Der Generator ist über seine Konfiguration mit den folgenden Parametern anpassbar:

- Modellbreite – Anzahl der Spalten einzelner Sheets
- Modellhöhe – Anzahl der Zeilen einzelner Sheets
- Zieltiefe – Anzahl der Dokumentebenen die unter den finalen Versuchsdokumenten existieren sollen
- Sheetanzahl – Anzahl der Sheets die in jedem einzelnen Dokument, im Linkpool und für Versuche, generiert werden sollen

Um die Größe des statischen Linkpools zu begrenzen, ist die Menge der Dokumente jeder Ebene begrenzt. Jede Ebene enthält die gleiche Anzahl Dokumente, welche der folgenden Formel folgen:

$$Dokumentanzahl = round(\sqrt{Modellbreite \cdot Modellhöhe \cdot Sheetanzahl \cdot Zieltiefe})$$

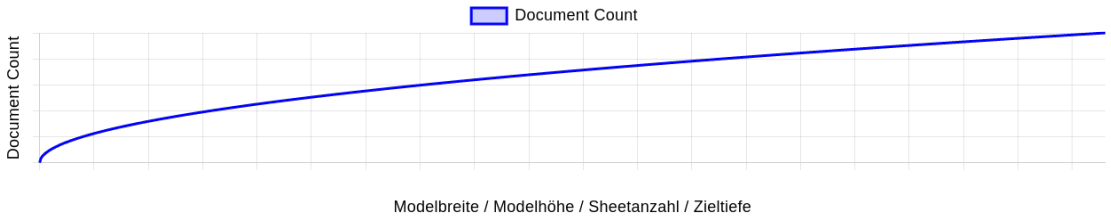


Abbildung 3: Abhängigkeit, der Gesamtanzahl an Dokumenten einer Ebene, von der Modellbreite, Modellhöhe, Sheetanzahl und Zieltiefe, jeweils unter der Annahme die anderen Parameter bleiben konstant

Für jedes Experiment wird ein eigener Generator mit einer, aus dem Experiment abgeleiteten, Konfiguration erstellt. Die Experimente wurden mit einem `PreparationState` versehen, der das Verhalten beim Anfragen durch Clients beeinflusst. Wenn ein Client Dokumente eines Experimentes in verschiedenen Zuständen anfragt, wird wie folgt reagiert:

- `UNPREPARED` – Das Experiment ist unvorbereitet und wird zum ersten Mal angefragt. Demnach muss die Generierung des statischen Linkpools und anschließend der einzelnen Versuchsdokumente starten. Das Experiment wird in den `PREPARING` Zustand überführt.
- `PREPARING` – Das angefragte Experiment befindet sich gerade in Vorbereitung. Je nach Dokument-Server und Experiment-Parametern kann dieser Zustand Stunden oder Tage anhalten. Den Clients wird in dieser Zeit eine leere Antwort zurückgegeben, damit die Anfrage nach einer selbstgewählten Zeitspanne erneut durchgeführt wird.
- `PREPARED` – Dieser Zustand signalisiert, dass der initiale Linkpool generiert wurde. Im Moment der Anwendung dieses Zustands wird auch die reguläre Dateigenerierung für das Experiment gestartet. Es werden dabei nie alle Dateien eines Experiments hintereinander generiert, sondern einzelne Batches definierter Größe. Bei der Anfrage des nächsten Versuchs für einen Client wird mittels eines Schwellwerts geprüft, ob noch ausreichend weitere generierte Dokumente existieren, falls nicht, wird das nächste Batch generiert.

Der Schwellwert für noch verbleibende Versuche beträgt in der aktuellen Implementierung 50 Versuche, bevor ein neues Batch von maximal 100 Versuchen zusätzlich generiert wird. Die 100 ist dabei die Obergrenze, da die Dokumentzahl eines Experiments beliebig ist und so das letzte Batch kleiner ausfallen kann.

Fragt ein Client einen neuen Versuch an, identifiziert der Server das zugewiesene Experiment und die Menge der bereits generierten Versuchsdokumente. Anschließend werden, aus der Datenbanktabelle für Versuchsergebnisse die Dokument-Identifier gesammelt für die dieser Client bei diesem Experiment bereits Ergebnisse geliefert hat. Aus der Menge generierter Dokumente, für welche noch kein Ergebnis des Clients vorliegt, wird das nächste Versuchsdokument ausgewählt.

Sollte ein Client Versuche eines Experiments schneller verarbeiten als der Server neue generieren kann, wird dies als Sonderfall behandelt. Dieses Ausnahmeszenario sollte vermieden werden, da sich der Client jetzt in einer synchronen Wartezeit befindet, bis ein neues Dokument generiert wird. In diesem Fall wird eine E-Mail Benachrichtigung an den User versandt und synchron ein neues Versuchsdokument erstellt und sofort zurückgeliefert. Das Eintreten dieses Zustandes ist ein Hinweis für eine Überlastung des Servers, beispielsweise durch eine zu geringe Schreibrate auf die Festplatte oder eine hohe CPU-Auslastung durch zu viele gleichzeitige Generierungsprozesse.

4.2.2 Verwaltung von Versuchsergebnissen

Ein Versuchsergebnis wird durch das Tripel aus Experiment-, Datei- und Client-Identifier identifiziert. Dass ein Client einen Versuch abgeschlossen hat, wird daran identifiziert, dass der Client ein Ergebnis für diesen Versuch geliefert hat. Diese Art unterstreicht die Zustandslosigkeit des Clients, da er weder alle Versuche eines Experiments kennen muss, noch alle durch ihn bereits abgeschlossenen Versuche. Der Server benötigt lediglich den Client-Identifier, um den Zustand eines Clients zu identifizieren. Ein Experiment gilt für den Server dann für einen Client als abgeschlossen, wenn für jeden Versuch eines Experiments, repräsentiert durch ein Versuchsdokument, ein Ergebnis des Clients existiert. Die Datenerhebung für Testergebnisse findet ausschließlich auf dem Client statt, die Aufbewahrung jedoch ausschließlich auf dem Server. Sollte es nicht möglich sein ein Ergebnis für einen Versuch auf dem Server abzulegen, etwa durch Netzwerk- oder Datenbank-Fehler, wird der Client diesen Versuch erneut ausführen, da es keinen Hinweis darauf gibt, dass der Versuch bereits durchgeführt wurde. Es besteht dadurch das Risiko unnötiger Versuchsausführungen, jedoch wird eine Zustandshaltung auf dem Client vermieden. Da Clients jederzeit ausfallen und neu gestartet werden können, muss der globale Zustand immer durch den Server gehandhabt werden. Dies erleichtert den Recovery-Prozess nach einem spontanen Ausfall.

Die durch den Client erhobenen Daten für ein Versuchsergebnis beinhalten:

- `experiment` – Der Identifier des Experiments zu dem dieses Ergebnis gehört
- `fileDescriptor` – Der Identifier des Versuchsdokuments für dieses Ergebnis

- `originClient` – Der Identifier des Clients auf dem der Versuch ausgeführt wurde
- `previousFile` – Der Identifier der vorherigen Versuchsdatei, um später die Reihenfolge von Versuchen nachvollziehen zu können
- `exception` – Die Exception-Message, sollte ein Crash auftreten
- `crash` – Boolean-Wert, der angibt, ob ein Crash während des Versuchs auftrat
- `hang` – Boolean-Wert um zu signalisieren, dass ein Timeout aufgetreten ist
- `errorCount` – Anzahl von erkannten Fehlern und Warnungen der Anwendung im Falle einer regulären Versuchsausführung
- `duration` – Die Dauer des Versuchs mit Millisekunden-Auflösung
- `timestamp` – Der Zeitstempel des Versuchsbeginns auf dem Client (kann durch Zeitzonen von der Server-Zeit abweichen)

4.2.3 Health Monitoring

Durch die Clients werden verschiedene Health-Metriken erhoben. Diese werden serverseitig um Watchdogs ergänzt, um außergewöhnliche Systemzustände der Clients außerhalb der SUT zu erkennen.

Zu den gesammelten Metriken gehören:

- Relative CPU-Auslastung
- Relative RAM-Auslastung
- Relativ belegter Festplattenspeicher des Laufwerks auf dem sich das Arbeitsverzeichnis befindet

Diese Metriken werden von jedem Client, sowie dem Server, sekündlich erhoben. Während der Server sich des `@Scheduled` Mechanismus des Spring-Frameworks bedient, um die Daten mit einer fixen Rate zu erheben, nutzen die Clients einen separat implementierten Mechanismus:

Codeauszug 3 zeigt die gewählte „best-effort“ Implementierung. Dies bedeutet der Client versucht dem Anspruch, einmal pro Sekunde die Metriken zu versenden, so gut wie möglich nachzukommen. Sollte dies aus verschiedenen Gründen nicht möglich sein, etwa weil die Erhebung der Daten oder der Versand zum Server zu lange dauert, kann es sein, dass serverseitig die Frequenz eingehender Health-Daten niedriger ist. Im Fall einer Überlastung des Clients, sollte das System nicht zusätzlich mit weiteren Tasks belastet werden, weshalb sich für diese Implementierung entschieden wurde, welche in einem einzelnen Thread arbeitet.

```java
@AllArgsConstructor
public class HealthWorker extends ReportingWorker {

    private static final int HEALTH_REPORT_INTERVAL_MS = 1000;

    private final BackendConnector connector;

    @Override
    protected void work() throws InterruptedException {
        while (!isStopped()) {
            var start = System.currentTimeMillis();
            connector.reportHealth(HealthReport.fromSystemState());
            var sleepTime = HEALTH_REPORT_INTERVAL_MS - ↩
                (System.currentTimeMillis() - start);
            if (sleepTime > 0) {
                Thread.sleep(sleepTime);
            }
        }
    }
}
```

Codeauszug 3: `HealthWorker` – Fixed-rate Scheduling über Zeitmessungen mit best-effort Ansatz

Serverseitig werden diese Metriken in einem „Moving-Average“ gesammelt. Dieser ist zeit-basiert, sodass jeder Wert beim Einfügen mit einem Zeitstempel versehen wird. Vor der Berechnung des Durchschnitts werden veraltetes Messwerte entfernt. Der Umfang dieses Durchschnitts ist frei wählbar, in der konkreten Implementierung werden 60 Sekunden genutzt. Ergänzt wird diese Konfiguration durch einen „Confidence“-Schwellwert, eine Ganzzahl welche angibt wie viele Werte mindestens vorhanden sein müssen, damit das Ergebnis als verlässlich gilt. Dieser Parameter ist nicht proportional zum Zeit-Umfang der Werte, sondern kann beliebig gesetzt werden. Die konkrete Implementierung geht davon aus, dass der Wert verlässlich ist, wenn für die letzten 60 Sekunden mindestens 30 Werte vorhanden sind. Es können über „System-Properties“, in der Kommandozeile oder der Konfigurationsdatei des Servers, Beschränkungen für diese Health-Metriken gesetzt werden. Die Standardwerte sind dabei 90 % Auslastung für CPU und RAM, sowie 80 % für die Festplatte. Sollten diese Werte vom Server oder einem Client überschritten werden, wird der Nutzer via E-Mail benachrichtigt. Die Anzahl der Benachrichtigungen ist beschränkt, um bei längeren Lastspitzen nicht zu viele Nachrichten zu verschicken. Für einzelne Clients kann die Benachrichtigung nach Bedarf im Frontend deaktiviert werden.

Ergänzt wird die Überwachung gemeldeter Daten durch Watchdogs, welche das Ausbleiben von Meldungen überwachen. Sollte ein Client für 5 Minuten keine

Health-Metriken oder für 30 Minuten kein Ergebnis melden, wird der Nutzer ebenfalls benachrichtigt. Dies ist ein Hinweis auf eine Absturz der Client-Anwendung oder eine langanhaltende, blockierende Versuchsausführung. Das Ausbleiben von Ergebnissen kann zudem ein Hinweis sein, dass es weitreichende Probleme bei der Verarbeitung von Versuchsdokumenten gibt. So können zum Beispiel sehr große Versuchsdokumente zu einer anhaltenden Systemüberlastung führen.

4.2.4 Event-Daten

Neben der umfangreichen API für das Frontend und die Clients war ein Ziel der Implementierung die Veröffentlichung von Event-Daten zu ermöglichen. Im Bereich der verteilten Systeme erlaubt push-style Event-Verteilung geringere Latenzen zur Event-Verarbeitung bei Konsumenten [Etz11, 2.1.3 Push-style event interactions]. Im Falle der Implementierung dieser Arbeit erlaubt es ein frühes Reagieren auf kritische Systemzustände. Diese Events werden im Backend über einen Websocket in abonnierbaren Namensräumen, sogenannten Topics, veröffentlicht. Die veröffentlichten Events sind:

- Versuchsergebnisse, welche auf einem separaten Topic für jeden Client veröffentlicht werden
- Health-Metriken veröffentlicht der Server ebenfalls auf einem separaten Topic pro Client. Dies erlaubt eine Visualisierung im Frontend, wie auch eine programmatische Reaktion externer Systeme außerhalb derjenigen, welche direkt am Fuzzing beteiligt sind
- Server-Output wird zu Debugging-Zwecken auf einem Topic veröffentlicht, um auf anderen Systemen zu lesbar zu sein, ohne direkten Zugriff auf den Host der Serveranwendung zu haben. Durch die Nutzung eines `StreamGobbler`, einer Klasse welche unbegrenzt Daten eines endlosen `InputStream` verarbeiten kann, wird der Standard-Output der Serveranwendung mithilfe eines `BufferedReader` zeilenweise gelesen und anschließend zeilenweise versandt

4.3 Dokument-Client

Der Dokument-Client ist die Software, welche für jeden Versuch eines Experiments eine Liste von Dokumenten des Servers empfängt und die konkrete Anwendung instrumentiert, um das eigentliche Black-Box Testing durchzuführen. Problemefelder sind dabei das Dateimanagement der einzelnen Versuche eines Experiments, sowie das plattformunabhängige Sammeln und Health-Metriken und die eigentliche Instrumentierung.

4.3.1 File Management

Im Rahmen des Studienprojekts war eine der wesentlichen Herausforderungen das Dateimanagement. Es ist programmatisch trivial, hunderttausende Dateien zu generieren, jedoch erzeugt dies praktische Probleme mit Betriebs- und Dateisystemen. So ist häufig die Anzahl an I-Nodes in einem Dateisystem beschränkt. Ein I-Node ist dabei die Repräsentation einer Datei und ihrer Metadaten [Tan09, 1.6.3 Systemaufrufe zur Verzeichnisverwaltung]. Um diese Probleme zu umgehen, wurden einzelne Versuchsverzeichnisse nach dem Versuch in ein Archiv gepackt. Damit konnten tausende von I-Nodes zu einem einzelnen zusammengefasst und bei Bedarf zur späteren Untersuchung wieder dekomprimiert werden. Dies hatte ebenfalls den Vorteil den Bedarf an Festplattenspeicher zu reduzieren. Ein entscheidender Nachteil war, dass zwischen zwei Versuchen jedes Mal synchron das Arbeitsverzeichnis archiviert werden musste, um Verzögerungen während der Versuche durch hohe Festplattenauslastung zu verhindern [Jün22, 4.1 Versuchsaufbau].

Wie bereits in „2.2.3 File Fuzzing“ beschrieben, muss zum Multifile Fuzzing immer der Weg über ein Dateisystem gewählt werden. Daher ist vorausschauendes Dateimanagement ein wesentlicher Faktor zur Performanz und Handhabbarkeit einzelner Versuche oder ganzer Experimente. Um erwähnte Probleme von Vornherein zu adressieren, wurde in dieser Arbeit ein fundamental anderer Ansatz gewählt. Dieser Ansatz unterstützt ebenfalls die verteilte Architektur des Gesamtsystems und ist auf die weitgehende Zustandslosigkeit der Clients ausgelegt. Ein wesentlicher Mechanismus ist die Entkopplung der Server- und Client-Anwendung von der Art der Dateiübertragung über minimalistische Interfaces. Ein anderer Mechanismus ist das Caching von Dateien auf Seite der Clients, um die Versuchsvorbereitung zu beschleunigen und gleichzeitig die Belastung des Dateisystems mit, nicht am aktuell ausgeführten Versuch beteiligten, Dateien zu beschränken.

Abstraktion der Datei-Persistenz

Die Entkopplung von der Dateiübertragung wird über je ein Interface auf Server- und Clientseite realisiert. Serverseitig existiert das Interface `FilePersistence`, welches auf das Persistieren und Löschen von Dateien beschränkt ist. Werden Versuchsdateien erzeugt, können diese implementierungsunabhängig mit einem festen Key abgelegt werden. Sollte während der Generierung ein Fehler auftreten oder ein Experiment-Fortschritt zurückgesetzt werden müssen, können Dateien unter Angabe dieses Keys gezielt gelöscht werden. Clientseitig dient das Interface `FileAccessor` lediglich zum Anfragen von Dateien. Unter Nutzung eines Keys kann der Inhalt einer Datei gezielt angefragt und unter einem Pfad im Dateisystem abgelegt werden. In der Umsetzung wurde diese Entkopplung via „Amazon Web Services (AWS)“ realisiert. AWS bietet unter dem Dienst „S3“ die Möglichkeit Daten in sogenannten „Buckets“ zu organisieren. Die Daten können in diesen

Buckets unter einem Identifier abgelegt, angefragt und gelöscht werden. So ist als Referenz-Implementierung die Klasse `AwsFilePersistence` entstanden, welche unter Angabe eines Bucket-Identifier Versuchsdateien in einem Bucket persistieren kann. Diese Implementierung wird über den, im Spring-Framework verbreiteten, „Dependency Injection“ Mechanismus lediglich über das Klassen-Interface `FilePersistence` anderen Komponenten der Serveranwendung zur Verfügung gestellt. Dies erlaubt später einen einfachen Austausch der Implementierung, in der entsprechend annotierten Factory-Methode, ohne weitere Anpassungen an anderen Stellen der Serveranwendung vornehmen zu müssen. Die Implementierung des Interfaces wird per Dependency Injection anderen Komponenten des Dokument-Servers zur Verfügung gestellt [Win23]. Das clientseitige Gegenstück ist, der Nomenklatur folgend, die Klasse `AwsFileAccessor`. Objekte dieser Klasse können, ebenfalls unter Angabe eines Bucket-Identifier, mithilfe des Keys auf einzelne abgelegte Dateien in einem Bucket zugreifen.

Das Management von Dateieinhalten wurde bewusst zu einem externen Dienst-Anbieter ausgelagert, um verschiedene Herausforderungen, welche nicht im Fokus der Aufgabenstellung liegen, zu lösen:

- Speicherplatz – Der Summe der Dateien aller gesammelten Experimente kann potenziell hunderte Gigabyte in Anspruch nehmen. Hochverfügbare, hochperformante Speichervolumina dieser Größenordnung sind technisch nicht trivial zu realisieren.
- Zugriffsraten – Durch die Ausführung mehrerer unterschiedlicher Experimente auf verteilten Clients werden sehr viele Dateien gleichzeitig angefragt. Wichtige Faktoren sind sowohl Lese-Raten von Speichermedien, als auch Netzwerk-Kapazitäten in Richtung der einzelnen Clients.
- Redundanz und Replikation – Zur späteren Untersuchung müssen Dateien zukunftssicher und im Falle von Datenverlusten, redundant aufbewahrt werden. Diese Herausforderung ist durch die Nutzung externer Anbieter implizit gelöst.

Durch die Abstraktion ist es möglich Dateien mit wenig Implementierungsaufwand auch anders zu persistieren, beispielsweise auf einem Network-Share, über einen separaten REST-Controller mit eigener API oder mit jedem anderen Dateiübertragungssystem. Wesentliche Voraussetzung ist das Unterstützen der drei Grundoperationen (Persistieren, Löschen, Abfragen) und die Verfügbarkeit sowohl von Server-, als auch Clientseite.

Datei-Cache

Clientseitig existiert eine Klasse `FileCache`, deren Objekte unter Nutzung eines `FileAccessor`, Dateiinhalte in einem festgelegten Verzeichnis zur schnelleren Be-

reitstellung vorhalten können. Dies hat deutliche Performance-Vorteile bei der Vorbereitung von Experimenten. Anstatt jede benötigte Datei erneut über das Netzwerk übertragen zu müssen, wird eine Eigenschaft des unter „4.2.1 Fuzzing-Generator“ erwähnten statischen Linkpools ausgenutzt. Durch diesen erzeugt der Generator Versuchsdateien, welche immer wieder dieselben Dateien im Linkpool referenzieren. Der Client kann so eine begrenzte Menge häufig verwendeter Dateien lokal vorhalten und lediglich in das Arbeitsverzeichnis der SUT kopieren, anstatt sie über das Netzwerk anfragen zu müssen. Die getroffene Grundannahme ist, dass im Regelfall das Kopieren von Dateien im selben Dateisystem oder zwischen zwei Dateisystemen des Clients schneller ist, als der externe Zugriff über den `FileAccessor`. Zur Realisierung der Caching-Funktionalität wurde ein `LoadingCache` von Googles „Guava“ Bibliothek genutzt. Dieses abstrakte Interface benötigt zwei generische Typ-Parameter für Schlüssel- und Werte-Typ der zu verwaltenden Daten. Die dahinterstehende Implementierung ist ein LRU-Cache mit einer, über die Client-Anwendung konfigurierbaren, maximalen Größe [Pro21]. Der Standardwert für die Cache-Größe in der Client-Implementierung sind zehntausend Dateien. Über einen `RemovalListener` kann darauf reagiert werden, wenn Elemente aus dem Guava-Cache verdrängt werden. In diesem Fall wird die zugehörige Datei gelöscht.

4.3.2 Plattformunabhängige Instrumentierung

So wie der Generator im Kern der Serveranwendung, steht die Instrumentierung der SUT im Zentrum der Client-Anwendung. Im Studienprojekt wurden über JQF Java-Bibliotheken durch Unit-Tests instrumentiert. Dadurch, dass die Versuche in derselben JVM stattfanden konnten viele Metriken durch die Instrumentierung sehr einfach erfasst werden. Code-Coverage, Crashes und deren Stacktraces und Hangs konnten durch JUnit und JQF gehandhabt werden. Versuchsdateien wurden in Form von Funktionsparametern direkt in einen JUnit Test-Case hineingegeben [Jün22, 4.1 Versuchsaufbau].

Das Black-Box Testing der Office-Suites, als Ziel dieser Arbeit, erfordert eigenen Implementierungsaufwand zum Erfassen dieser Metriken. Um diesen Aufwand zu minimieren, war es ein Ziel einen Großteil des Versuchsablaufs für beide untersuchten Office-Suites zu vereinheitlichen und erst im entscheidenden Schritt der Ausführung auf die anwendungsspezifische Implementierung zurückzugreifen. Zu diesem Schritt wurde ein Interface `Application` erstellt, dass anwendungsunabhängige Funktionen bereitstellt, die für die einzelnen SUT implementiert werden müssen:

- `prepare` – Bereite die Ausführung von Tests vor, lade die entsprechende Office-Suite und stelle die Interprozesskommunikation her
- `isExecutionPrepared` – Zur Versicherung, dass die Interprozesskommunikation weiterhin besteht. Diese Methode wird direkt vor jedem Versuch

aufgerufen und ist besonders wichtig, wenn es beim vorherigen Versuch zu einem Crash kam und die Anwendung re-initialisiert werden musste.

- `execute` – Die Kern-Funktionalität einer Anwendung, die Dokumente laden kann. Kann mittels einer `InterruptionHook` zur Unterbrechung aufgerufen werden, jedoch entscheidet die jede Implementierung selbst, ob einem Aufruf zur Unterbrechung, beispielsweise bei einem Timeout, gefolgt werden soll. Diese Funktion gibt die Anzahl an Warnungen der Zielanwendung als Ganzzahl zurück.

- `shouldRetry` – Hierüber kann eine Implementierte `Application` entscheiden einen Versuch zu wiederholen, weil der aufgetretene Fehler beispielsweise nicht mit der Anwendung, sondern der Interprozesskommunikation zu tun hat.

- `cleanup` – Nach einem Crash muss häufig die Anwendung vollständig geschlossen und neu geöffnet werden, um eventuelle Seiteneffekte auf spätere Versuche auszuschließen. Zu diesem Zweck wird nach einem Crash diese Methode mit einem anschließenden `prepare` genutzt, um den nächsten Versuch vorzubereiten.

Der abstrakte Versuchsablauf findet im Client in einer Endlosschleife statt und lässt sich wie folgt abstrahieren:

1. Frage den nächsten Versuch vom Dokument-Server ab
2. Bereite den Workspace der Anwendung vor
    - Lösche alte Versuchsdateien
    - Lade neue Versuchsdateien über den `FileCache`
3. Führe den Versuch mit der konfigurierten SUT durch
    - Starte den Versuch mit einem gesetzten Timeout
    - Sollte ein Crash auftreten zeichne ihn auf und beende den Versuch
    - Sollte ein Timeout auftreten vermerke dies und weise die Zielanwendung über den `InterruptionHook` an, den Versuch regulär zu unterbrechen
    - Iteriere über alle Sheets des Workbooks und zähle die Zahl an Fehlermeldungen in einzelnen Zellen
4. Melde das Ergebnis an den Server
5. Sollte ein Crash oder eine andere Ausnahme aufgetreten sein re-initialisiere die SUT

Auch wenn das Interface zum Zweck des Black-Box Testing erzeugt wurde, lässt es sich darüber hinaus nutzen. Für diese Arbeit wurde es für zwei externe Anwendungen mit feature-reichen Programm-Schnittstellen implementiert. Diese beiden Anwendungen wurden gezielt gewählt, um die Plattformunabhängigkeit der Implementierung herauszustellen, da die Eine (Microsoft Excel ™) lediglich für „Microsoft Windows ™“ verfügbar ist, die Andere (The Document Foundation LibreOffice ™) ist plattformunabhängig. Im Folgenden wird kurz auf die Art der Instrumentierung der beiden Anwendungen eingegangen.

Microsoft Excel

```
1  public void openExcelBook(String filename) throws COMException {
2      this.invokeNoReply("Open", getWorkbooks().getIDispatch(),
3          new VARIANT[] {
4              new VARIANT(filename),
5              new VARIANT(3), // Update Links - 3 aktualisiert alle ↩
                   Links
6              VARIANT_MISSING, //ReadOnly
7              VARIANT_MISSING, // Format
8              VARIANT_MISSING, // Password
9              VARIANT_MISSING, // WriteResPassword
10             VARIANT_MISSING, // Ignore ReadOnly Recommended
11             VARIANT_MISSING, // Origin
12             VARIANT_MISSING, // Delimiter
13             VARIANT_MISSING, // Editable
14             VARIANT_MISSING, // Notify
15             VARIANT_MISSING, // Converter
16             VARIANT_MISSING, // AddToMru
17             VARIANT_MISSING, // Local
18             new VARIANT(0) // CorruptLoad - 0 Öffnet die Datei regulär
19     });
20 }
```

Codeauszug 4: COM-Schnittstelle: Links beim Programmstart aktualisieren und scheinbar korrupte Dateien nicht reparieren [Mic22] [Mic21b]

Als Tabellenkalkulation der verbreitetsten Office-Suite war es ein Ziel dieser Ausarbeitung Microsoft Excel zu instrumentieren. Excel bringt ein eigenes SDK mit, dass Interprozesskommunikation nutzt, um Excel zu bedienen. Dies wird von verschiedenen Anwendungen zu Dokumentkonvertierung genutzt. Dieses SDK hat C# Bindings, welche sich nur über Umwege in Java nutzen lassen. Als Schnittstelle wurde „Java Native Access (JNA)“, eine Bibliothek zum direkten Zugriff auf geteilte,

dynamische Bibliotheken in Verbindung mit der Excel Interprozesskommunikation-API genutzt [Pro23a] [Mic18]. JNA ist in der Lage verschiedene Schnittstellen unter Windows zu nutzen, wie etwa „Object Linking and Embedding (OLE)“ und „Component Object Model (COM)“. Das Projekt beinhaltet bereits ein Beispiel, dass Excel instrumentieren kann, jedoch ohne wesentliche Funktionen wie dem Setzen wichtiger Flags oder das Auslesen von Fehlern aus Zellen. Zur besseren Nutzbarkeit innerhalb der Umsetzung, wurden die entsprechenden Werkzeuge ergänzt. Beim Öffnen von Dateien muss zusätzlich eine Flag gesetzt werden, um einen Dialog zu verhindern, der selbst auftritt, wenn die grafische Benutzeroberfläche programmatisch deaktiviert wurde. Die Flag, welche erlaubt scheinbar korrupte Dateien ohne Nutzerinteraktion zu laden, wird in Visual Basic und C# über benannte Parameter gesetzt, über die COM-Schnittstelle stehen jedoch lediglich positionelle Argumente zur Verfügung, weshalb auch für alle anderen Flags der Wert `VARIANT_MISSING` gesetzt werden muss. (siehe Codeauszug 4)

The Document Foundation LibreOffice

```
1  public static XComponent openDoc(String fnm, XComponentLoader ↩
       loader) {
2      return openDoc(fnm, loader, Props.makeProps(
3          // Headless
4          "Hidden", true,
5          // ALWAYS_EXECUTE_NO_WARN=4 führt Makros aus, ohne ↩
               Bestätigung des Nutzers
6          "MacroExecutionMode", (short)4,
7          // QUIET_UPDATE=1 aktualisiert die Werte aller Makros beim ↩
               Laden von Dokumenten ohne bei Fehlern dem Nutzer eine ↩
               Warnung zu zeigen
8          "UpdateDocMode", (short)1
9      ));
10 }
```

Codeauszug 5: JLOP: Makros zu externen Dateien beim Programmstart laden und die Werte im Hintergrund zu aktualiseren [Proa] [Prob]

LibreOffice ist eine plattformunabhängige Office-Suite, die selbst zu großen Teilen in Java implementiert ist. Als Schnittstelle wurde das Uno-Interface von Apache OpenOffice genutzt, da LibreOffice 2010 von diesem Projekt als Fork entstanden ist [Pro23b]. Für diese API existiert eine Wrapper-Bibliothek „Java LibreOffice Programming (JLOP)“, welche Prinzipien des Uno-Protokolls und der JNA-Bibliothek vereint. Dieser Wrapper [Dav22b], verfasst von Dr. Andrew Davison an der „Prince of Songkla University“ [Dav22a] umfasst Werkzeuge zum Öffnen und Schließen von Dateien, sowie dem Auslesen von Fenster-Inhalten der Office-Anwendung. Um sie

für die Implementierung der Arbeit nutzbar zu machen, wurde ein Gradle-Modul erstellt, welches eine JAR-Datei erzeugt. Zusätzliche Flags wurden beim Öffnen von Dokumenten ergänzt (siehe Codeauszug 5).

4.4 Server Frontend

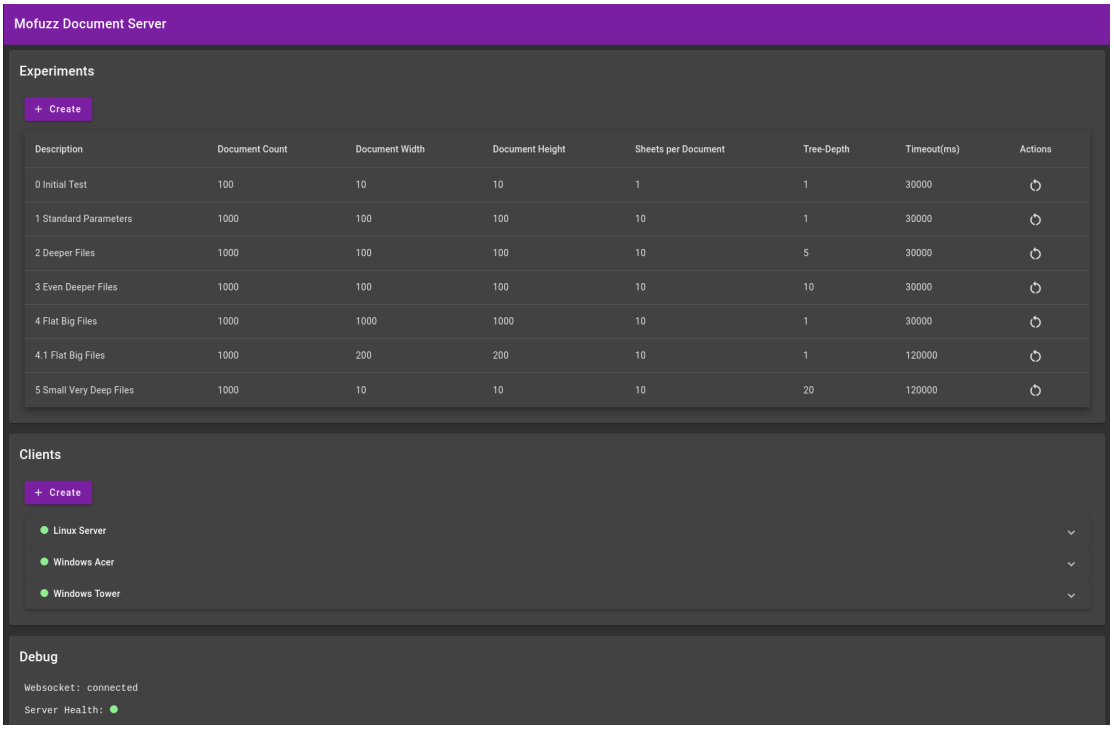


Abbildung 4: Web-Frontend implementiert mit Angular und Material-Design

Das Frontend erfüllt zwei Anforderungen: Das einfache Anlegen von Clients und Experimenten mit allen benötigten Parametern, sowie das Bereitstellen von Debug Informationen während die Clients ihre konfigurierten Experimente ausführen.

Das Erstellen von Experimenten und das Registrieren der Clients sind durch Dialoge geführt, die an den entsprechenden Stellen sinnvolle Standardwerte setzten, ein Beispiel dafür sind Experiment-Timeouts von 30 Sekunden.

Als Debug-Informationen werden die Event-Daten des Servers und der Clients (siehe „4.2.4 Event-Daten“) in der Form von Indikatoren mit Tooltips präsentiert. Zusätzlich ist es in der Detailansicht für einzelne Clients möglich den Fortschritt des aktuellen Experiments, sowie den Zeitstempel des letzten Testergebnisses, einzusehen.

Die Ergebnisse der Experimente können in der Detailansicht näher untersucht werden. Das Frontend ist hier unterteilt in einen allgemeinen Bereich für die Ergebnisse aller Clients und einen client-spezifischen Bereich. Letzterer visualisiert zugleich die historischen Aufzeichnungen der Health-Metriken während der Versuchsdurchführung.

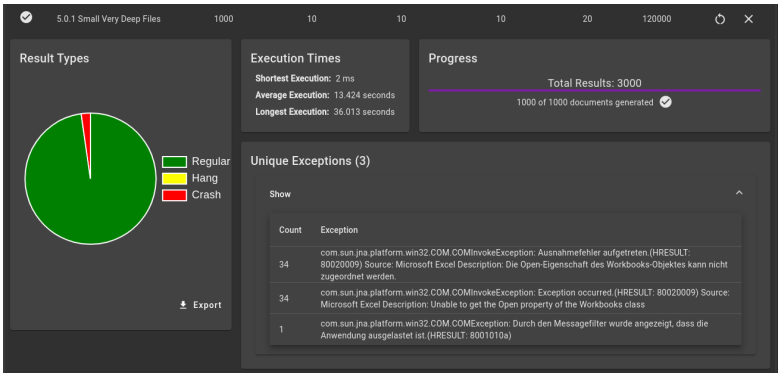


Abbildung 5: Ansicht der allgemeinen Ergebnisse eines Experiments

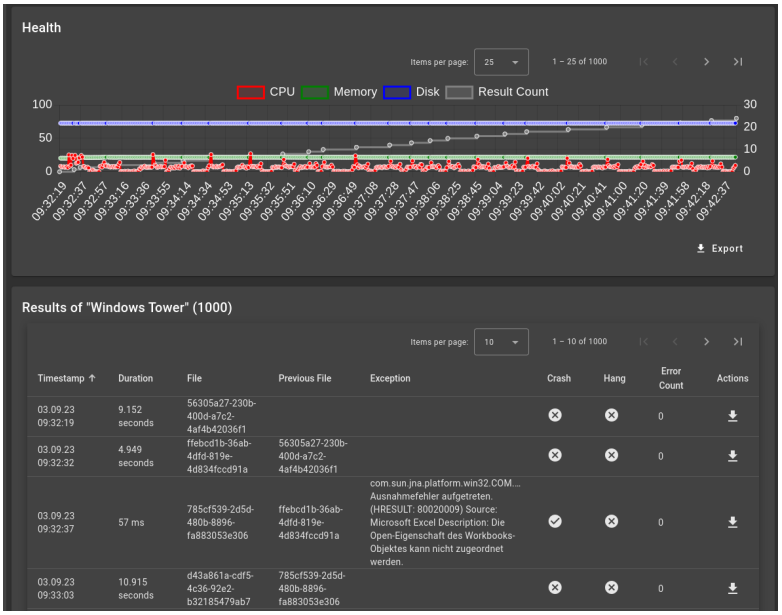


Abbildung 6: Ansicht der Ergebnisse eines Clients für ein ausgewähltes Experiment

Am Fuß der Ansicht ist der Standard-Output des Servers einsehbar, um den aktuellen Zustand des Servers besser zu veranschaulichen. So kann zum Beispiel das Generieren von Test-Dokumenten beobachtet werden, was über reguläre Funktionen der REST-API nicht möglich ist.

Technisch ist das Frontend ein Gradle-Modul, welches das Gradle-Node-Plugin nutzt. Das npm-Buildscript wird innerhalb des Gradle-Lifecycle ausgeführt und anschließend wird das Kompilat mit dem Gradle-Java-Plugin in einer JAR verpackt. Als Angular Projekt nutzt es zusätzlich nur `SockJS` zur Integration des Websocket.

5 Auswertung

Die erstelle Software ist in der Lage Black-Box und Back-to-Back Testing verschiedener Office-Suites durchzuführen. Um Excel und LibreOffice zu vergleichen und währenddessen die Fähigkeiten des entwickelten Werkzeugs zu untersuchen, wurden verschiedene Fuzzing-Kampagnen durchgeführt. Das Erstellen dieser Kampagnen war ein explorativer Prozess, der im folgenden Kapitel von verschiedenen Seiten beleuchtet wird.

5.1 Versuchsaufbau

Der Versuchsaufbau lässt sich in verschiedene Betrachtungsebenen unterteilen. Dies ist sinnvoll, da sich einzelne Ebenen mit sehr unterschiedlichen Herausforderungen befassen. Grob wird zwischen dem äußeren Versuchsaufbau des verteilten Systems und dem inneren Versuchsaufbau des Generators unterscheiden.

5.1.1 Infrastruktur und Orchestrierung

Zum grundsätzlichen Versuchsaufbau war es notwendig verlässliche, hochverfügbare Infrastruktur zu schaffen. Um die Plattformunabhängigkeit der realisierten Software zu zeigen, wurde Microsoft Excel unter Windows und LibreOffice unter Linux, getestet. Zu diesem Zweck wurde ein virtualisierter Server in einem Rechenzentrum angemietet, welcher für den Versuchsaufbau als Linux-Client des Dokument-Servers dient. Die technischen Daten der einzelnen Clients können der folgenden Tabelle entnommen werden:

Bezeichnung	**Linux-Client**	**Laptop-Client**	**Tower-Client**
System	Virtualisiert Ubuntu 22.04.3 LTS	Laptop Windows 10 22H2	Stand-PC Windows 10 22H2
Leistungs-parameter	6 vCPUs AMD Epyc Series 16 GB RAM 100 GB NVme	2 CPU-Threads Intel Pentium 3556U 16 GB RAM 256 GB SSD	16 CPU-Threads AMD Ryzen 7 1700X 32 GB RAM 1 TB NVme
Java-Version	OpenJDK 17.0.8	OpenJDK 17.0.2	OpenJDK 17.0.2
SUT	LibreOffice 7.3.7.2 30 Build 2	Microsoft Excel 2309 Build 16827.20130	Microsoft Excel 2309 Build 16827.20130

Der Dokument-Server wurde auf einem separaten virtuellen Linux-Server (Ubuntu 20.04.6 LTS, 4 Intel Core Haswell vCPUs, 8GB RAM, 100GB SSD) in Form eines Docker-Containers ausgerollt. Über `docker-compose` wurden mehrere Container für das Backend in einem virtuellen Netzwerk orchestriert:

- Eine „MariaDB“ Datenbank zur Persistenz der verschiedenen Zustände, Metadaten, Health-Metriken, etc.
- Ein Web-Interface zur direkten Interaktion mit der Datenbank. Es dient vor allem zu Debugging-Zwecken, da die Datenbank selbst nicht öffentlich verfügbar ist (Adminer).
- Der `mofuzz-document-server` Container mit dem eigentlichen Backend
- Ein „Caddy“ Webserver als Proxy vor den eigentlichen Diensten
  - Der Dokument-Server ist unter `mofuzz.alotof.tech` verfügbar
  - `db.alotof.tech` wird zur Bereitstellung des Web-Interfaces der Datenbank genutzt

5.1.2 Parametrisierung der Experimente

Wie bereits unter „4.2.1 Fuzzing-Generator“ beschrieben, ist der realisierte Generator für Modellbreite, Modellhöhe, Zieltiefe und Sheetanzahl konfigurierbar. Diese Parameter werden, etwa zur Rekonstruktion nach einem Neustart, in einem Experiment-Objekt in der Datenbank gespeichert. Neben den Generator-Parametern haben Experimente weitere konfigurierbare Metadaten, wie eine Beschreibung, die Versuchsanzahl und das Versuchstimeout. Mit dieser Parameterauswahl wurden verschiedene Experimente durchgeführt:

Beschreibung	**Dokument-anzahl**	**Modell-höhe**	**Modell-breite**	**Sheet-anzahl**	**Ziel-tiefe**	**Timeout (in ms)**
0 Initial Test	100	10	10	1	1	30000
1 Standard Parameters	1000	100	100	10	1	30000
2 Deeper Files	1000	100	100	10	5	30000
3 Even Deeper Files	1000	100	100	10	10	30000
4 Flat Big Files	1000	1000	1000	10	1	30000
4.1 Flat Big Files	10000	200	200	10	1	120000
5 Small Very Deep Files	10000	10	10	10	20	120000
6 Unlinked Files	10000	10	10	10	0	120000

Nicht jedes Experiment wurde vollständig durchgeführt und nicht jedes vollständige Experiment beinhaltet diskussionsfähige Daten. So erzeugte Experiment „2 Deeper Files“ plattformunabhängig keine Exceptions, im Gegensatz zu Experiment „4 Flat Big Files“, dessen Dokumente nach mehreren Tagen Generierungszeit, lediglich Timeouts erzeugten, da die Clients nicht in der Lage waren, verzweigte Dokumente

mit jeweils 10 Millionen Zellen in vertretbarer Zeit zu laden. Die folgende Diskussion der Ergebnisse wird sich auf eine Auswahl der Experimente mit interessanten Ergebnissen beschränken.

5.1.3 Abstrakter Versuchsablauf

Neben dem technischen Aufbau der Versuchsumgebung, ist es wichtig, den Versuchsablauf zu verstehen. Wie unter „4.2.1 Fuzzing-Generator“ beschrieben, werden die Dateigraphen einzelner Versuche in Ebenen generiert. Zur Vorbereitung eines Versuchs wird ein sogenannter Linkpool erzeugt. Begonnen wird auf der untersten Ebene, deren Dokumente keine Verweise beinhalten, sondern Daten der verschiedenen, unter „2.3 Office Open XML“ beschriebenen, Datentypen. Jeder Datentyp ist dabei gleich wahrscheinlich und der zufällige Wert der Zelle wird, je nach Datentyp, anders erzeugt. Die Inhalte jeder Zelle, jedes Sheets, in jedem Dokument sind zufällig erzeugt. Als Zufallsquelle wird die `Random` Klasse der Java-Standardbibliothek, ohne festen Seed verwendet. Die Berechnung für die Anzahl der zu generierenden Dokumente pro Ebene ist in der Generator-Klasse dieser Arbeit festgelegt. In der, für den Versuchsaufbau genutzten Implementierung, skaliert diese Anzahl nicht mit der Anzahl der Versuche eines Experiments, sondern mit anderen Parametern des Generators, wie unter „4.2.1 Fuzzing-Generator“ beschrieben. Demnach haben beispielsweise zehn Experimente mit 1000 Versuchen eine größere Varianz an nicht verlinkenden Dokumenten der untersten Ebene, als ein Experiment mit zehntausend Versuchen.

$$Dokumentanzahl = round(\sqrt{Modellbreite \cdot Modellhöhe \cdot Sheetanzahl \cdot Zieltiefe})$$

Alle Dokumentenebenen, über die unterste Daten-Ebene hinaus, beinhalten keine Daten mehr, sondern ausschließlich Verweise auf Zellen anderer Dokumente aus dem Linkpool. Diese Verweise werden zufällig aus dem Linkpool gewählt. Ist eine Ebene vollständig generiert, werden die Zellen ihrer Dokumente dem Linkpool hinzugefügt. Die für jedes Experiment konfigurierte Zieltiefe legt fest, wie viele Ebenen zur Versuchsvorbereitung zu generieren sind. Es werden alle Dokumentebenen, ausschließlich der eigentlichen Versuchsebene, vorbereitet. Die oberste Ebene der Versuchsdokumente beinhaltet dann die Wurzeldokumente der gerichteten Dateigraphen.

Ist die Versuchsvorbereitung abgeschlossen, wird ein erstes Batch an Versuchsdokumenten generiert. Auch diese beinhalten keine eigenen Daten, sondern ausschließlich Verweise auf Zellen in anderen Dokumenten des Linkpools. Warten die Clients auf die Vorbereitung eines neuen Experimentes, fragen sie alle 30 Sekunden erneut beim Server Versuchsdokumente an. Parallel zur Generierung des ersten Batch beginnen die Clients mit der Versuchsdurchführung, sobald das erste Versuchsdokument zur

Verfügung steht. Dem Client wird dabei der Identifier des Versuchsdokuments, die Identifier aller Dokumente des darunterliegenden Dokumentengraph, das Versuchstimeout, sowie die Modellhöhe und -breite mitgeteilt.

Nun beginnt die clientseitige Versuchsvorbereitung. Der Client bereitet das Arbeitsverzeichnis vor, indem er alle vorhandenen Dateien löscht. Anschließend werden über den lokalen Dateicache (siehe „4.3.1 File Management“), die für den Versuch notwendigen, Dokumente entweder angefragt oder direkt aus dem Cache in das Arbeitsverzeichnis kopiert. Anschließend beginnt der eigentliche Versuch. Im Versuch wird mit der SUT das Versuchsdokument geöffnet. Kann dieses Dokument geöffnet werden, wird über jede Zelle, jedes Sheets des Dokuments iteriert. Beinhaltet die Zelle einen anwendungsspezifischen Fehlercode, wird ein Zähler erhöht. Ist die Iteration abgeschlossen, wird der Versuch beendet und die Anzahl der erkannten Fehler zurückgegeben. Kommt es bei der Versuchsdurchführung zu Exceptions, bricht der Versuch ab und die Exception-Message bildet, zusammen mit einem Crash-Flag, das Ergebnis des Versuchs. Dauert eine Versuchsausführung länger als das für das Experiment konfigurierte Timeout, wird sie von außen abgebrochen und das Versuchsergebnis erhält ein Timeout-Flag. Nach der Versuchsdurchführung durch den Client wird das Ergebnis an den Dokument-Server übermittelt und der nächste Versuch angefragt.

Der Dokument-Server beachtet bei der Auswahl des nächsten Versuches die Menge der Versuche, für die ein Client bereits Ergebnisse geliefert hat. Außerdem wird überprüft, wie groß die Menge der verfügbaren Versuche ohne Ergebnis ist. Ist die Größe dieser Menge unter einem Schwellwert, in der konkreten Anwendung 50, und sind noch nicht alle Versuche eines Experiments generiert, wird serverseitig parallel mit der Generierung des nächsten Batch begonnen. Das Ergebnis eines Versuches ist über das Frontend des Dokument-Servers zugänglich. Für interessante Ergebnisse können durch den Nutzer Dokumentgraphen einzelner Versuche händisch, in Form eines Archivs, heruntergeladen und untersucht werden.

5.2 Ergebnisse der Experimente

Die Durchführung der Experimente war ein explorativer Prozess. Dies ist auch darin begründet, dass die ersten Experimente vor allem als manuelle System-Tests der gesamten Software fungierten. Diese Ergebnisse beinhalten Crashes die in späteren Versuchen nicht mehr auftreten konnten. Neben der Stabilität war auch die Parameterauswahl explorativ. Beispielsweise hat der Versuch „4 Flat Big Files“ einen Nachfolge-Versuch „4.1 Flat Big Files“, dessen Dateien nur noch 4 % so groß sind, da die vormals generierten Dokumente in Anzahl und Dateigröße jedes zumutbare Maß überschritten. Dies unterstreicht zwar die Zuverlässigkeit der Generator-Implementierung, erzeugt jedoch keine auswertbaren Ergebnisse. Im Folgenden werden die zehn Experimente „6 Unlinked Files“ als

„Baseline-Experimente“ und die zehn Experimente „5 Small Very Deep Files“ als „Multifile-Experimente“, bezeichnet.

Die Versuche „5 Small Very Deep Files“ und „6 Unlinked Files“ wurden zehn Mal mit je 1000 Dokumenten durchgeführt. Dies wurde getan, da das Potential eines Fuzzers im wesentlichen von der Seeds abhängt [Kle+18, 5 Seed Selection]. Die Seeds sind in diesem Fall die Dokumente ohne Referenzen auf der untersten Ebene. Wie unter „5.1.3 Abstrakter Versuchsablauf“ beschrieben, ist die Varianz dieser Dokumente in der konkreten Implementierung größer, wenn eine Fuzzing-Kampagne in mehrere Experimente aufteilt ist.

Bei den, in den folgenden Abschnitten gezeigten, Liniendiagrammen ist auf der Abszissenachse nicht die Zeit abgebildet. Grund dafür sind die sehr unterschiedlichen Ausführungszeiten der Experimente auf den unterschiedlichen Clients. Besonders die, um Größenordnungen kleineren, Ausführungszeiten des Linux-Clients würden ein Abbilden auf einer gemeinsamen Zeitachse sehr unübersichtlich machen (siehe Abb. 7). Aus diesem Grund wird in der Abszissenachse der Fortschritt der Versuche der Experimente abgebildet. Demnach bezieht sich beispielsweise eine 100 auf den einhundertsten Versuch in der Reihe aus Versuchen.

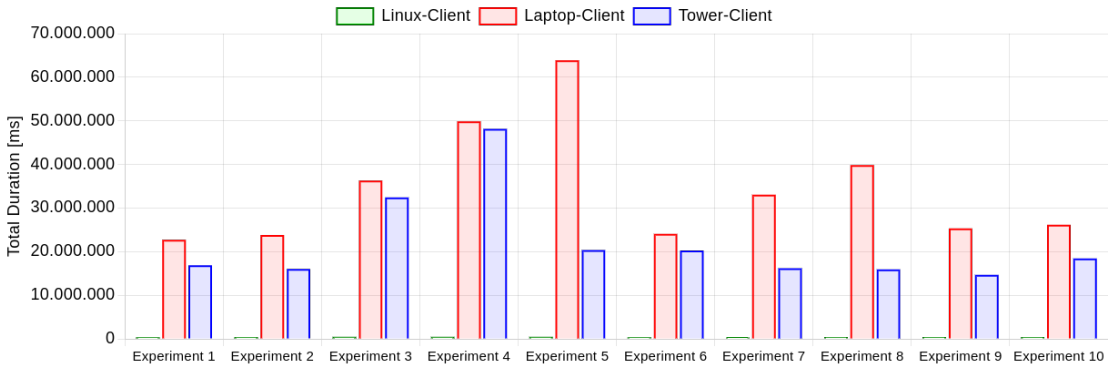


Abbildung 7: Die Ausführungszeiten ganzer Experimente in Millisekunden – Sie unterscheiden sich zu stark, um sie sinnvoll auf einer gemeinsamen Zeitachse darzustellen

Die aufgezeichneten Ergebnisse der Clients sollen genutzt werden, um die folgenden drei Forschungsfragen zu beantworten:

RQ1: Kann Multifile Fuzzing mehr Exception-Types erzeugen als das Fuzzen einzelner Dateien?

RQ2: Können die beiden SUT mit diesem Setup vergleichbare Back-to-Back Ergebnisse erzielen?

RQ3: Welche Rückschlüsse können aus unterschiedlich leistungsstarken Systemen mit der selben SUT gezogen werden?

Diese drei Forschungsfragen wurden bewusst gewählt, um unterschiedliche Aspekte der Realisierung zu beleuchten. So hat RQ1 besonderen Bezug zum Fuzzing, RQ2 zum Back-to-Back Testing und RQ3 zum Black-Box Testing der beiden SUT.

5.2.1 Effektivität des Multifile Fuzzing

Dieser Abschnitt beantwortet RQ1 „Kann Multifile Fuzzing mehr Exception-Types erzeugen als das Fuzzen einzelner Dateien?“. Die Frage bezieht sich explizit auf Exception-Types, da es für die Bewertung der Leistungsfähigkeit eines Fuzzers nicht ausreicht die absolute Anzahl der Crashes zu vergleichen. Grund dafür ist, dass verschiedene Inputs den selben Bug hervorrufen können. Das hervorrufen des selben Bugs mit verschiedenen Inputs ist kein Leistungsmerkmal eines Fuzzers [Kle+18, 7 Performance Measures]. Aus diesem Grund müssen die aufgezeichneten Crashes de-dupliziert werden. Ein Lösungsansatz kann Stack-Hashing sein, also Call-Stacks oder Stacktraces über einen Hash abzubilden und nur einzigartige Hashes zur Evaluation heranzuziehen. Die konkret implementierte Instrumentierung für Excel und LibreOffice dieser Arbeit ist jedoch nicht in der Lage, diese Informationen zu erfassen. Der Stacktrace einer Exception bezieht sich hierbei immer auf den Punkt, an dem die Ausführung der instrumentierten Anwendung an die Interprozesskommunikation übergeben wird.

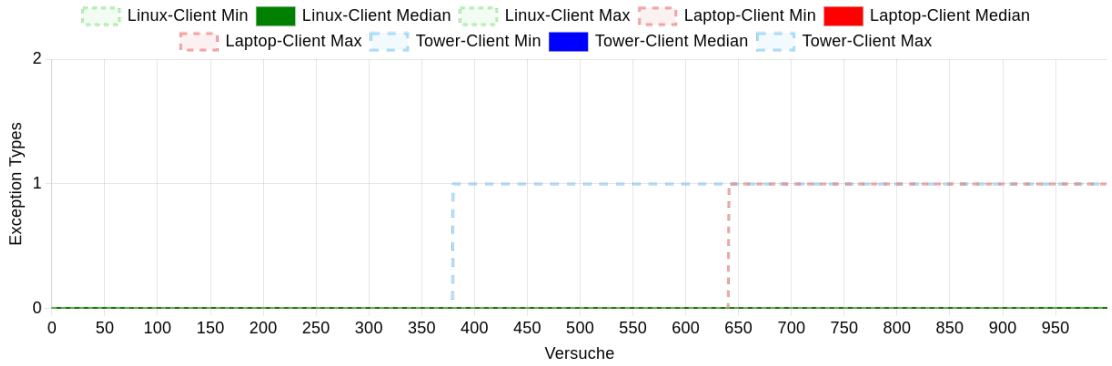


Abbildung 8: Anzahl unterschiedlicher Exception-Types über die Versuche der Baseline-Experimente

Das Diagramm der Baseline-Experimente (Abb. 8) zeigt, dass die einfachen Workbooks, ohne Referenzen, im Mittel (Median) keine Exception-Types erzeugt haben. Einzelne Experimente haben auf dem Laptop- und Tower-Client zwar Exception-Types erzeugt, jedoch nicht genug um den Median zu beeinflussen. Das Diagramm der Multifile-Experimente (Abb. 9) zeigt, dass im Mittel (Median) Laptop- und Tower-Client bereits nach wenigen Versuchen einen aufgetretenen Exception-Type verzeichnen. Der Maximalwert des Laptop-Clients zeigt als einzige Datenspur zwei Exception-Types. Diese sind:

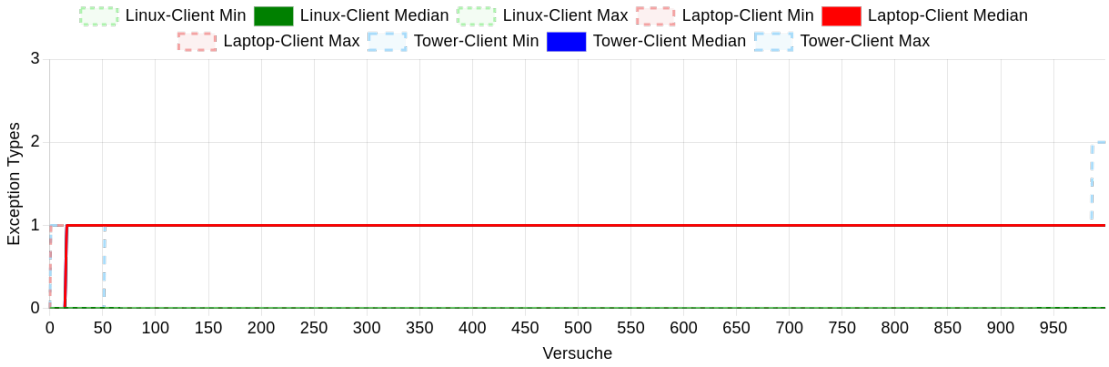


Abbildung 9: Anzahl unterschiedlicher Exception-Types über die Versuche der Multifile-Experimente

- `com.sun.jna.platform.win32.COM.COMInvokeException` – Der generische Fehler welcher, durch die Interprozesskommunikation, Crashes von Excel symbolisiert.
- `java.util.concurrent.TimeoutException` – Ein Timeout, dass aus zu klärenden Gründen nicht als Timeout, sondern Crash erkannt wurde. Dieser Umstand ist vermutlich auf einen Implementierungsfehler in der Instrumentierung zurückzuführen.

Für den Linux-Client konnten in keinem der Experimente Exception-Types aufgezeichnet werden. Der Mann-Whitney-U-Test zeigt jedoch, dass in den Multifile-Experimenten für den Laptop- ($p \approx 0,0025$) und Tower-Client ($p \approx 0,0006$) signifikant mehr Exception-Types gemessen wurden.

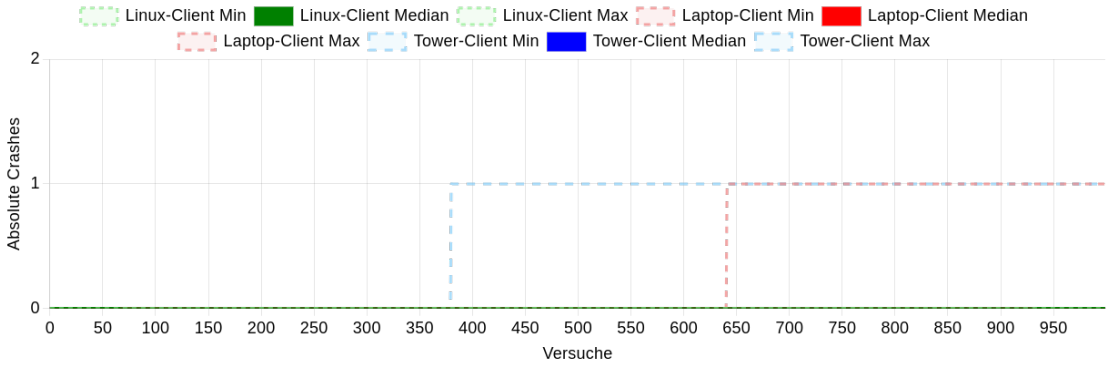


Abbildung 10: Anzahl an Crashes über die Versuche der Baseline-Experimente

Auch wenn die absolute Anzahl der Crashes keine Performance-Metrik eines Fuzzers ist, kann sie zeigen, mit welcher Zuverlässigkeit der Fuzzer in der Lage ist

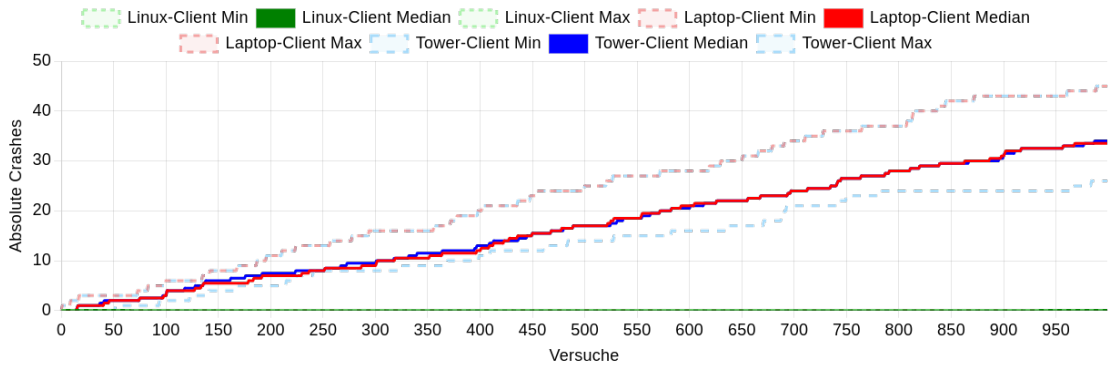


Abbildung 11: Anzahl an Crashes über die Versuche der Multifile-Experimente

Crashes zu erzeugen. All diese Crashes können sich potentiell auf den selben Bug beziehen, in diesem Fall ist der Fuzzer jedoch zumindest in der Lage diesen zuverlässig zu reproduzieren. Die Diagramme (Abb. 10 und Abb. 11) zeigen, dass die Baseline-Experimente im Median keine Crashes erzeugen. Im Maximum wurden bei diesen Experimenten einzelne Crashes verzeichnet. Der Laptop- und Tower-Client verzeichnen über die 1000 Versuche der Multifile-Experimente im Median 33,5, respektive 34, Crashes. Die Diskrepanz erklärt sich aus einem Experiment, bei dem der Laptop-Client keine Crashes verzeichnete, wodurch auch der Minimalwert für diesen Client bei 0 bleibt. Der Tower-Client misst im gleichen Experiment 33 Crashes des Typs `com.sun.jna.platform.win32.COM.COMInvokeException` mit der Fehlermeldung „*Ausnahmefehler aufgetreten.(HRESULT: 80020009) Source: Microsoft Excel Description: Die Open-Eigenschaft des Workbooks-Objektes kann nicht zugeordnet werden.*“. Da dieser Exception-Type lediglich auf dem Tower-Client aufgetreten ist, liegen Ursachen außerhalb des Versuchsaufbaus, wie etwa Dateisystemfehler, nahe. Eine nähere Untersuchung der betroffenen Dateien zeigte keine Auffälligkeiten. Alle betroffenen Dateien ließen sich auf dem Tower-Client manuell öffnen. Der Mann-Whitney-U-Test zeigt, dass sowohl der Laptop-Client ($p \approx 0.0009$), als auch der Tower-Client ($p \approx 0.0002$) während der Multifile-Experimente signifikant mehr Crashes aufzeichneten.

Abschließend lässt sich RQ1 „Kann Multifile Fuzzing mehr Exception-Types erzeugen als das Fuzzen einzelner Dateien?“ mit den erhobenen Daten für Microsoft Excel bestätigen. Multifile-Experimente haben für beide Clients mit Excel als SUT zuverlässiger den Exception-Type provozieren können. Auch konnten mit den Multifile-Experimenten mehr Crashes als mit den Baseline-Experimente provoziert werden. Es muss näher untersucht werden, ob sich diese Crashes auf unterschiedliche Bugs beziehen.

Für LibreOffice lässt sich RQ1 nicht bestätigen, da weder die Baseline- noch die Multifile-Experimente in der Lage waren, Crashes der Software zu provozieren. Dieses Ergebnis kann verschiedene Ursachen haben, beispielsweise einen fehler-

toleranten Umgang beim Öffnen verlinkter Dateien. Außerhalb von LibreOffice kann auch eine fehlerhafte Instrumentierung die Erkennung der Crashes verhindert haben.

5.2.2 Vergleichbarkeit von Tabellenkalkulationsanwendungen

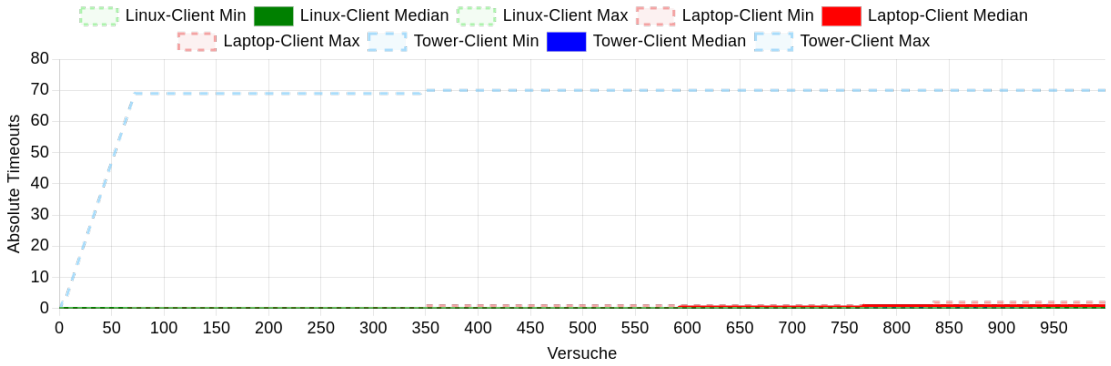


Abbildung 12: Anzahl an Timeouts über die Versuche der Baseline-Experimente

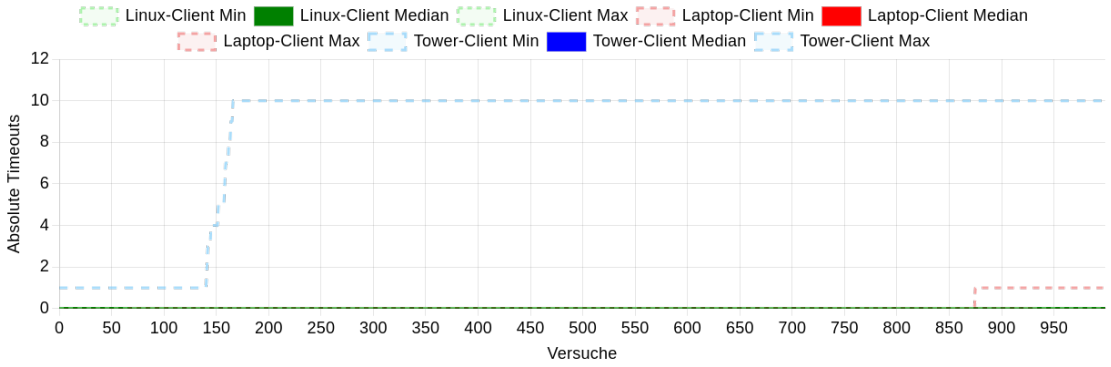


Abbildung 13: Anzahl an Timeouts über die Versuche der Multifile-Experimente

Ein wesentlicher Aspekt der entstandenen Implementierung ist das Testen verschiedener SUT mit den selben Dokumenten, auch Back-to-Back Testing genannt. Der folgende Abschnitt soll RQ2 „Können die beiden SUT mit diesem Setup vergleichbare Back-to-Back Ergebnisse erzielen?“ von verschiedenen Seiten beleuchten. Der Abschnitt „5.2.1 Effektivität des Multifile Fuzzing“ ist bereits auf Crash Metriken eingegangen und hat gezeigt, dass für LibreOffice kein einziger Crash aufgezeichnet werden konnte. Eine andere erfasste Metrik der Versuchsdurchführung sind Ausführungszeiten, beziehungsweise Timeouts. Für jedes Experiment wird eine maximale

Ausführungszeit für einzelne Versuche festgelegt. Wird diese überschritten wird das Ergebnis als Timeout gewertet. Timeouts können durch Dateien ausgelöst werden, bei denen eine SUT beispielsweise nicht mehr terminiert, ohne einen Crash auszulösen.

Die Visualisierung (Abb. 12 und Abb. 13) zeigt, dass in den meisten Experimenten lediglich vereinzelte Timeouts aufgetreten sind. Für beide Experiment-Reihen war das Timeout auf zwei Minuten konfiguriert. Der Median am Ende der Baseline-Experimente beträgt 0 für den Linux-Client, respektive 1 für den Laptop- und Tower-Client mit Excel als SUT. Das Maximum für den Laptop-Client ist zwei Timeouts im selben Experiment, während der Tower-Client 70 Timeouts verzeichnete. Innerhalb der Multifile-Experimente beträgt der Median für alle Clients 0. Der Laptop-Client verzeichnete maximal einen, der Tower-Client 10 Timeouts. Beim Linux-Client trat kein einziges Timeout auf, was besonders nachvollziehbar wird, wenn die Dauer einzelne Experimente betrachtet wird (Abb. 14 und Abb. 15).

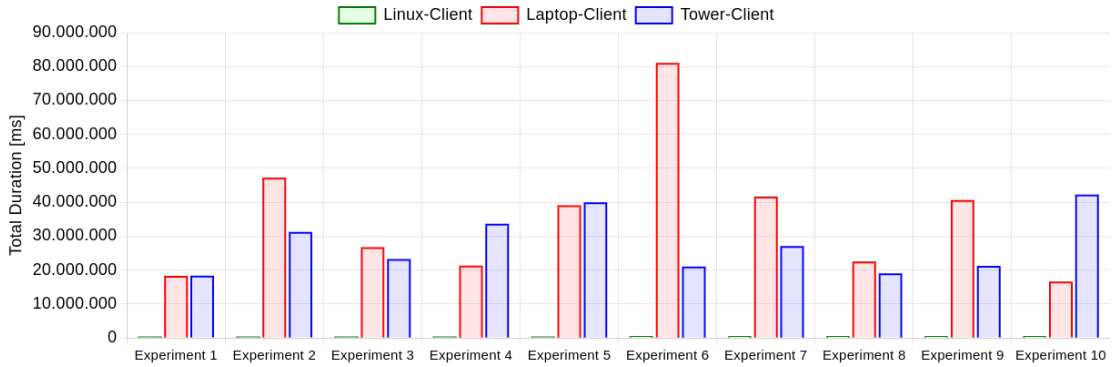


Abbildung 14: Ausführungszeit der Baseline-Experimente in Millisekunden

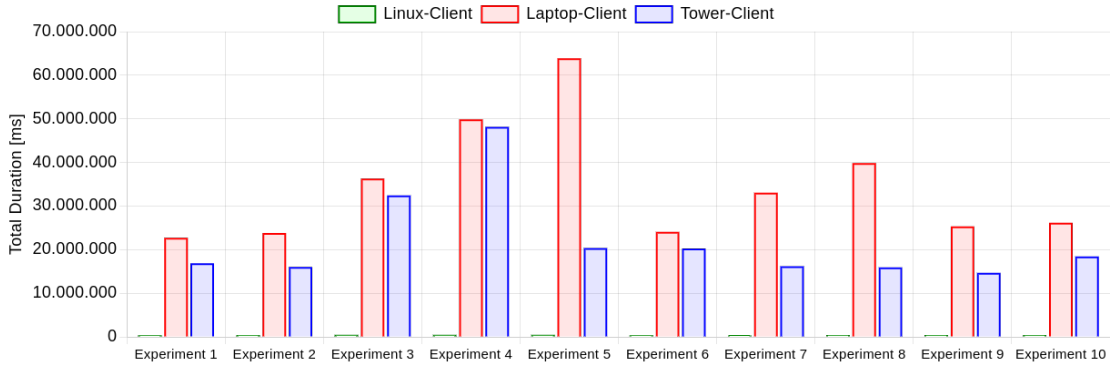


Abbildung 15: Ausführungszeit der Multifile-Experimente in Millisekunden

Den größten Unterschied verzeichnet ein Baseline-Experiment, bei dem das Ausführen aller Versuche auf dem Laptop-Client mehr als 200 mal länger dauerte als auf

dem Linux-Client. Diese absoluten Ausführungszeiten lassen sich jedoch nur bedingt mit der Menge der Timeouts in Verbindung bringen, da einzelne Timeouts keinen maßgeblichen Einfluss auf die Gesamtdauer aller Versuchsdokumente eines Experiments haben müssen. Aufgrund der unterschiedlichen Instrumentierung relativiert sich die Ausführungszeit als Vergleichskriterium, da während des Versuchs nicht ausschließlich die Zeit zum Öffnen der Datei gemessen wurde. Die Ausführungszeit beinhaltet neben dem Öffnen der Dateien durch die SUT zusätzlich die Evaluation der Zelleninhalte jedes Sheets, um Zellen mit einem Fehlerzustand zu identifizieren. Da in beiden Experiment-Reihen die Versuchsdokumente aus 1000 Zellen bestanden, machen sich Performance-Unterschiede beim Auslesen der Zellinhalte über die Instrumentierung maßgeblich in der Gesamtzeit eines Versuches bemerkbar. Um zu identifizieren, was die Ursache für die stark unterschiedlichen Ausführungszeiten ist, müsste die Zeitmessung der anwendungs-unabhängigen Instrumentierung in zwei Phasen „Öffnen“ und „Auslesen“ unterteilt werden. Dies könnte auch weitere Schlussfolgerungen auf die innere Funktionsweise der SUT erlauben, beispielsweise mögliche Korrelationen von Timings mit Dateigrößen beim Öffnen einer Datei oder dem Auslesen von Werten je nach Dateigraph-Tiefe.

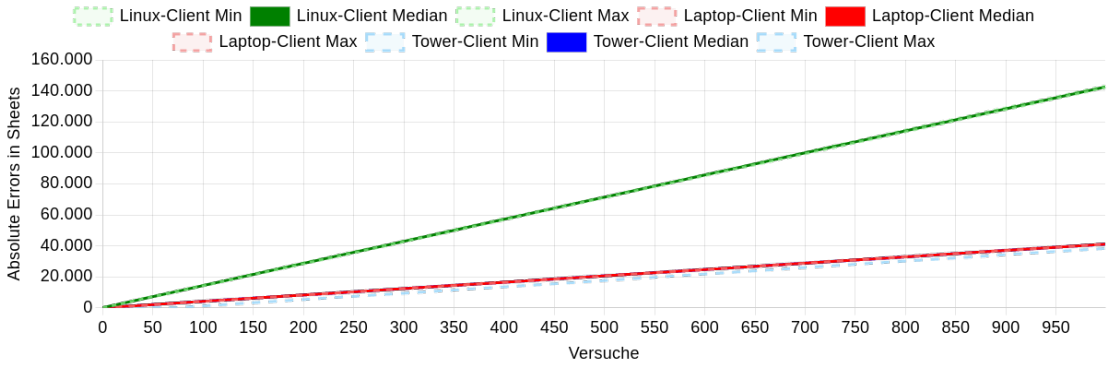


Abbildung 16: Anzahl erkannter Zellenfehler über die Versuche der Baseline-Experimente

Durch die Instrumentierung wurden die einzelnen Zellen jeder Versuchsdatei auf Fehlerzustände untersucht. Jede der anwendungsspezifischen Instrumentierungen erlaubt die Identifikation des Fehlertyps. Während der Versuchsdurchführung wurde lediglich das Vorhandensein eines Fehlerzustandes erkannt, jedoch wurde nicht anhand der Art des Zellfehlers unterschieden. Die Instrumentierung für Excel war nicht in der Lage den Fehlerzustand in den Zellen der Multifile-Experimente zu erkennen. Dies kann daran liegen, dass die Art die Instrumentierung für Excel dazu nicht in der Lage ist oder, dass Zellen mit Referenz-Formeln spezielle Fehlertypen haben, die von der Instrumentierung nicht erkannt wurden. Demnach zeigen die Diagramme (Abb. 16 und Abb. 17) lediglich Daten der Baseline-Experimente. Sie zeigen, dass der Linux-Client im Median mehr als dreimal mehr Fehler erkennt. Beide Anwendungen implementieren den gleichen Standard und teilen sich demnach

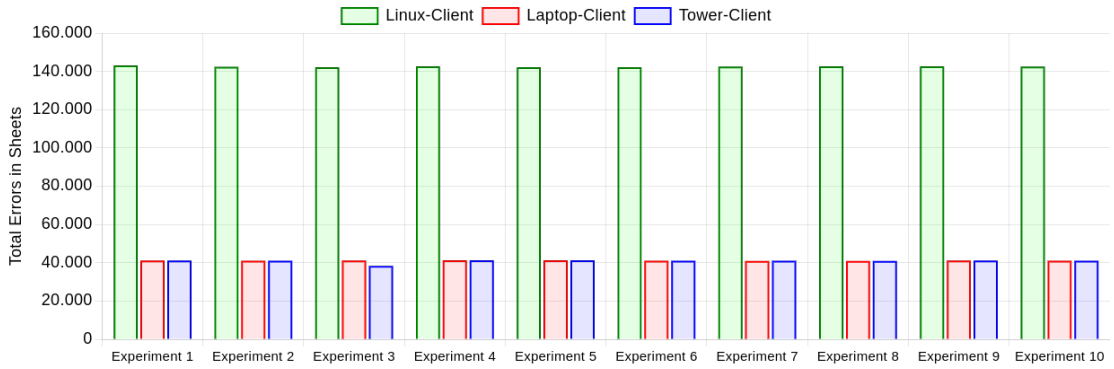


Abbildung 17: Anzahl erkannter Zellenfehler der Baseline-Experimente

eine Menge definierter Zellfehler, jedoch gibt es daneben eine anwendungsspezifische Menge an Fehlern [Int16, 18.17.3 Error values] [Gir23] [Mic23]. Dies kann den deutlichen Unterschied zwischen den beiden SUT erklären. Die leichten Unterschiede zwischen dem Laptop- und Tower-Client, welche beide Excel instrumentierten, lässt sich mit den Fehlerzahlen der unterschiedlichen Crashes und Timeouts erklären. Tritt ein Crash oder Timeout auf, wird die Anzahl der Zellen mit Fehlerzuständen nicht bestimmt und pauschal mit 0 gewertet. Werden die Daten um diese Versuche bereinigt, bleibt ein einziger Versuch in Baseline-Experiment 9, in dem der Laptop- und Tower-Client 46, respektive 45, Fehlerzustände messen. Bei einer manuellen Untersuchung wurden auf beiden Clients 45 Fehlerzustände in der betroffenen Datei ermittelt, dafür wurde die Excel eigene Formel `=SUMPRODUCT(--ISERROR(A1:J10))` genutzt. Die anwendungsspezifische Instrumentierung öffnet Dateien mit deaktivierter Reparatur-Option (siehe „4.3.2 Plattformunabhängige Instrumentierung“). Dies ist beim manuellen Öffnen nicht möglich. Es besteht die Möglichkeit, dass die Reparatur-Option Grund für die Diskrepanz zwischen Instrumentierung und manueller Prüfung ist. Dadurch kann jedoch nicht der Unterschied zwischen den beiden Clients erklärt werden, da diese die selbe Instrumentierung nutzen.

Der Abschnitt „5.2.1 Effektivität des Multifile Fuzzing“ stellt bereits fest, dass mit der realisierten Software, weder Exception-Types noch die absolute Anzahl Crashes zwischen den beiden Tabellenkalkulationen vergleichbar sind. Der Hauptgrund dafür ist, dass für LibreOffice nicht ein einziger Crash aufgezeichnet werden konnte. Dieser Abschnitt zeigt warum ebenfalls Timeouts, Ausführungszeiten und erkannte Fehlerzustände einzelner Zellen problematische Vergleichsmetriken sind. Timeouts sind bei der Versuchsplanung festgelegte Werte die vor allem der Effizienz der Versuchsdurchführung dient. Ausführungszeiten sind in der konkreten Implementierung nicht näher aufgeschlüsselt, was die Möglichkeit offen lässt, dass Unterschiede in der Art der Instrumentierung begründet sind und nicht in der SUT selbst. Insgesamt lässt sich Schlussfolgern, dass keine der erfassten Metriken in der Lage ist, eine eindimensionale Vergleichbarkeit der beiden SUT zu ermöglichen. RQ2

„Können die beiden SUT mit diesem Setup vergleichbare Back-to-Back Ergebnisse erzielen?“ lässt sich dadurch nicht bestätigen, da ohne vergleichbare Metriken auch keine vergleichbaren Ergebnisse erzielt werden können. Dieses Ziel kann erreicht werden, wenn beispielsweise Ausführungszeiten in Phasen erfasst werden würden oder Fehlerzustände nach Fehlerart gruppiert werden würden, statt alle Fehler einer Versuchsdatei als einzelne Zahl zu erfassen.

5.2.3 Rückschlüsse durch erfasste Metriken

Der vorherigen Abschnitt betrachtete die Vergleichbarkeit von Testergebnissen zwischen verschiedenen Plattformen mit unterschiedlichen SUT. In diesem Abschnitt sollen die Testergebnisse des Laptop- und Tower-Clients verglichen werden, da diese die selbe SUT auf der selben Plattform nutzen. RQ3 „Welche Rückschlüsse können aus unterschiedlich leistungsstarken Systemen mit der selben SUT gezogen werden?“ soll mit Hinblick auf die unterschiedliche Hardwarekonfiguration der beiden Systeme bewertet werden.

Wie unter „4.2.3 Health Monitoring“ beschrieben, wurden verschiedene Health-Metriken durch die Clients erfasst. Dies geschah, um auch ohne manuelle Beobachtung eine unterbrechungsfreie Versuchsdurchführung sicherzustellen. Mit Watchdogs wird sichergestellt, dass Health-Metriken und Test-Ergebnisse regelmäßig an den Dokument-Server gemeldet werden. Sollte mehr als 30 Minuten kein Ergebnis gemeldet werden, wird eine Meldung an den Nutzer per E-Mail ausgelöst. Dies war von besonderer Bedeutung, da die Excel Instrumentierung nicht in der Lage war den Excel Prozess von außen zu beenden. Sollte Excel für ein Versuchsdokument nicht terminieren, kann dies die weitere Versuchsdurchführung unbegrenzt verzögern. Die längste Versuchsdurchführung wurde für den Tower-Client im Rahmen der Experiment-Reihe „4.1 Flat Big Files“ gemessen und beträgt über elf Stunden für einen einzigen Versuch des Experiments. Dieser Versuch terminierte ohne äußeres Eingreifen. Neben den Watchdogs, beinhaltete der Dokument-Server auch Health-Monitoring über konfigurierbare Schwellenwerte. Überschreitet der Moving-Average einer Metrik über einen gewissen Zeitraum ihren Schwellwert, wird eine Warnung an den Nutzer gesendet. Für die relative CPU-Auslastung beträgt dieser Schwellwert 90%. Da der Laptop-Client lediglich eine CPU mit zwei Threads besitzt, betrug die CPU Auslastung dauerhaft über 90%. Zu diesem Zweck wurden die Health-Warnungen für einzelne Clients deaktivierbar gemacht. Vor diesem Hintergrund ergab sich die Annahme, dass der Laptop-Client wesentlich mehr Timeout-Ergebnisse verzeichnen würde, als der Tower-Client. Die Daten der Versuchsergebnisse widerlegen diese Annahme.

Die Diagramme (Abb. 18 und Abb. 19) zeigen die Absolute Anzahl an Timeouts einzelner Clients über die Baseline- und Multifile-Experimente. In den Daten verzeichnet der Linux Client, welcher für die Betrachtung von RQ3 zu vernachlässigen ist, kein einziges Timeout. Sie zeigen, dass der Laptop-Client lediglich in zwei

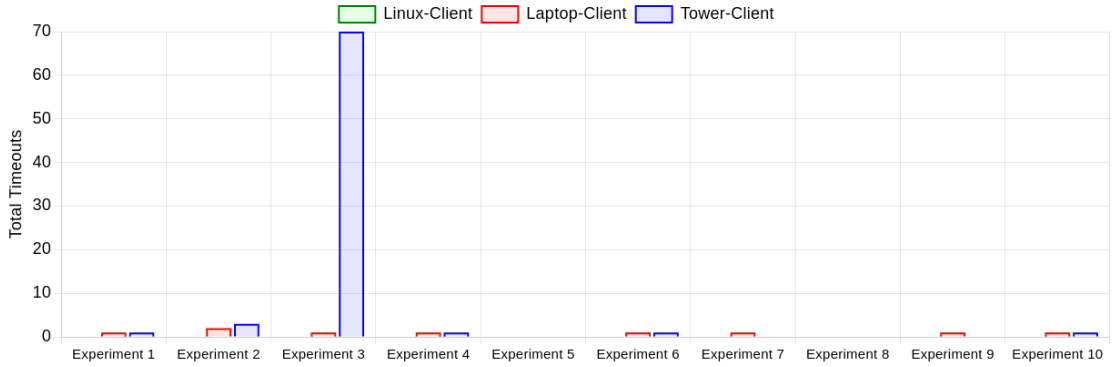


Abbildung 18: Anzahl an Timeouts der Baseline-Experimente

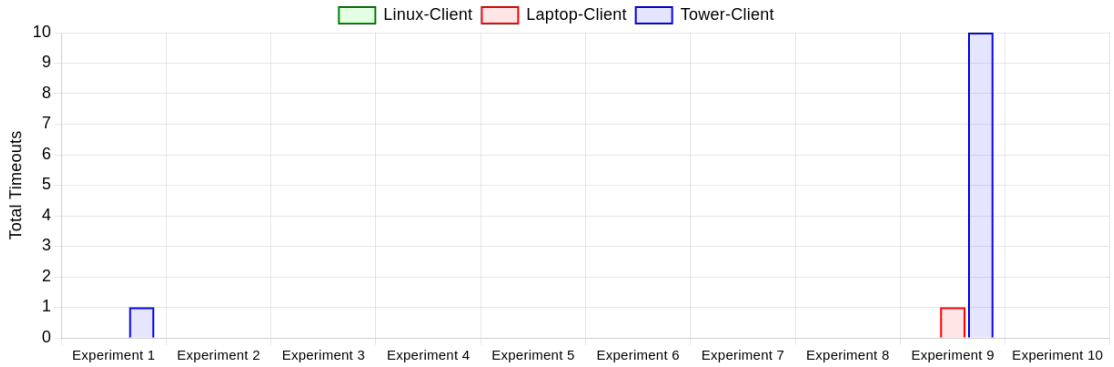


Abbildung 19: Anzahl an Timeouts der Multifile-Experimente

Experimenten, Baseline-Experiment 7 und 9, mehr Timeouts verzeichnet als der Tower-Client. In diesen Experimenten ergab sich je ein Timeout, wohingegen der Tower-Client die Versuche ohne Timeouts absolvierte. Mit 70 Timeouts der in Baseline-Experiment 3 und zehn Timeouts in Multifile-Experiment 9 überragen die akkumulierten Ergebnisse des Tower-Clients die anderen Experimente bei Weitem. Dies ist vor dem deutlichen Leistungsvorteil der Hardwarekonfiguration des Tower-Client von Bedeutung. Ähnlich zu „5.2.1 Effektivität des Multifile Fuzzing“ liegt auch hier der Einfluss äußerer Faktoren, abseits der SUT oder Instrumentierung, nahe. Ein wesentlicher Unterschied zwischen den beiden Systemen ist die Art der verbauten Festplatte. Beim Laptop-Client ist eine 2,5 Zoll SATA SSD mit Gehäuse verbaut. Der Tower-Client nutzt dagegen eine NVme ohne eigenen Heatsink. Diese Bauart von Festplatten neigt unter hoher Last zum „Thermal-Throttling“ [Şev23, Abstract]. Dabei wird die Leistungsfähigkeit einer Festplatte unter Last reduziert, um eine Beschädigung der Bauteile durch starke Wärmeentwicklung zu verhindern. Um diese Annahme zu verifizieren, wäre es notwendig, die Festplattenauslastung und -temperatur als zusätzliche Health-Metriken während der

Versuchsdurchführung zu erfassen.

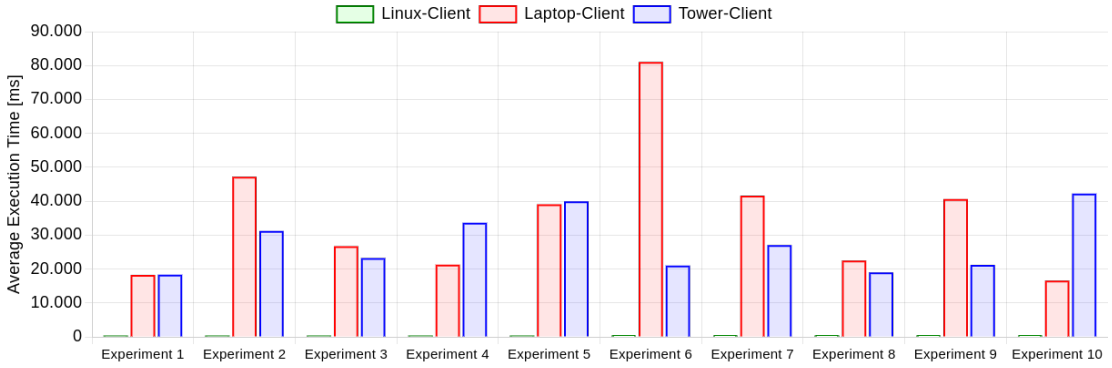


Abbildung 20: Durchschnittliche Ausführungszeiten einzelner Versuche der Baseline-Experimente in Millisekunden

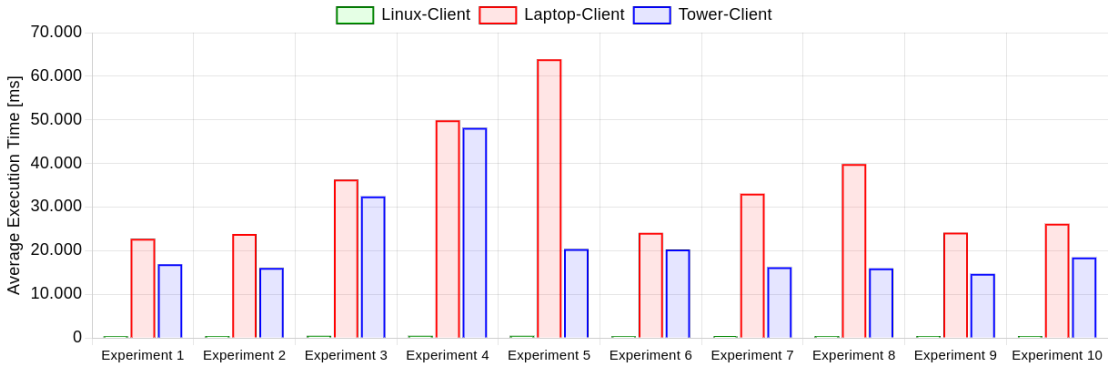


Abbildung 21: Durchschnittliche Ausführungszeiten einzelner Versuche der Multifile-Experimente in Millisekunden

Neben den Timeouts liegt für die Bewertung der Performance auch die durchschnittliche Versuchsdauer nahe. „5.2.2 Vergleichbarkeit von Tabellenkalkulationsanwendungen“ relativiert die Ausführungsdauer als Vergleichskriterium, vor allem auf Grundlage der unterschiedlichen Instrumentierung der beiden SUT. Da der Laptop- und Tower-Client die selbe SUT mit der selben Instrumentierung verwenden, ist es naheliegend, in diesem Kontext die Versuchsdauer als Kriterium heranzuziehen. Aus den erfassten Ergebnissen (Abb. 20 und Abb. 21) ergibt sich eine durchschnittliche Versuchsdauer des Laptop-Client von 35,5 Sekunden (Median: 32,9 Sekunden) für die Baseline-Experimente und 34,4 Sekunden (Median: 29,6 Sekunden) für die Multifile-Experimente. Der Tower-Client erzielte für die Baseline eine durchschnittliche Versuchsdauer von 27,7 Sekunden (Median: 25,1 Sekunden) und 21,9 Sekunden (Median: 17,7 Sekunden) für die Multifile-Experimente. Dies zeigt, dass die durchschnittliche Versuchsdauer für beide Clients bei Multifile-Experimenten niedriger ist, als die der Baseline.

Die Multifile-Experimente werden durch den Laptop-Client in ca. 3% (Median: 11%), beziehungsweise durch den Tower Client in ca. 26% (Median: 42%) weniger Zeit durchgeführt als die Baseline-Experimente. Werden die durchschnittlichen Ausführungszeiten von Laptop- und Tower-Client untereinander vergleichen, ergibt der Mann-Whitney-U-Test für die Baseline-Experimente keine Signifikanz ($p \approx 0.4962$). Bei Multifile-Experimenten ergibt sich ein signifikanter Unterschied der Ausführungszeiten zwischen den beiden Clients ($p \approx 0.0051$). In den Einstellungen von Excel existiert eine Option, um Multi-Threading für die Evaluation von Berechnungen zu aktivieren. Auf beiden Clients war diese Option aktiviert. Da Referenzen zu anderen Dokumenten als Formeln notiert werden, betrifft die Option wahrscheinlich ebenfalls das Laden von Inhalten aus anderen Dateien. Dies würde den deutlichen Vorteil des Tower-Clients, zum einen gegenüber den Baseline-Experimenten und zum anderen gegenüber dem Laptop-Client erklären, da dieser 16 CPU-Threads besitzt. Der Laptop Client besitzt hingegen nur zwei CPU-Threads (siehe „5.1.1 Infrastruktur und Orchestrierung“). Zur Verifikation dieser Annahme könnte ein Setup ähnlich konfigurierter Maschinen genutzt werden, die sich lediglich in der Anzahl der CPU-Threads unterscheiden. Alternativ könnten auch das Erfassen der CPU-Auslastung, heruntergebrochen auf einzelne logische Threads, einen tieferen Einblick in die Parallelisierung der Verarbeitung von Dateigraphen geben.

Abschließend lässt sich sagen, dass der Performance-Vorteil des Tower-Client zu den niedrigeren Ausführungszeiten geführt hat. Da die durchschnittlichen Ausführungszeiten von 21,9 bis 35,5 Sekunden jedoch weit unter der konfigurierten Timeout-Zeit von 120 Sekunden liegen, gab es in den Experiment-Reihen keine Vorteile in Bezug auf Timeouts für den leistungsfähigeren Client. Der Tower-Client hatte für einzelne Experimente mehr Timeouts als der Laptop-Client. Die Gründe dafür liegen wahrscheinlich außerhalb der Instrumentierung oder SUT und können am wahrscheinlichsten durch weitere erfasste Health-Metriken näher untersucht werden. Insgesamt hat der Tower-Client niedrigere durchschnittliche Ausführungszeiten als der Laptop-Client, sowohl für die Baseline-, als auch für die Multifile-Experimente. Dies kann in zusätzliche Parallelisierung begründet sein. Diese Vermutung muss jedoch durch Experimente mit Clients noch ähnlicherer Konfiguration untersucht werden. Zur Beantwortung von RQ3 „Welche Rückschlüsse können aus unterschiedlich leistungsstarken Systemen mit der selben SUT gezogen werden?“ sind weitere Experimente mit verändertem Versuchsaufbau notwendig.

5.3 Gültigkeitsrisiken

Bezüglich der internen Gültigkeit ist besonders die zufällige Natur des Fuzzing hervorzuheben. Zu diesem Zweck wurden die Experimente zehn Mal mit der selben Anzahl von Dokumenten durchgeführt und ihre Ergebnisse nach den Vorgaben von Klees et al ausgewertet [Kle+18]. Es wurden Exception-Types betrachtet, Mediane,

Minima und Maxima verwendet und über Mann-Whitney-U-Tests die statistische Signifikanz bestimmt. In allen Experimenten war die Instrumentierung und die Art der Metrik-Erfassung konsistent. Die Daten wurden automatisiert erhoben und ausgewertet. Durch die plattformunabhängige Instrumentierung wurde die Beeinflussung einzelner Versuche untereinander minimiert und die Reihenfolge der Versuche durch erhobene Metadaten in den Ergebnissen dokumentiert. Ein wesentliches Risiko der internen Gültigkeit, ist die mit 1000 Versuchsdokumenten pro Experiment, beziehungsweise 10000 Dokumente pro -Reihe, geringe Anzahl an Versuchsdurchführungen. Im Zuge des Multifile Fuzzing wurden durch die SUT ebenfalls die tausenden anderen generierten Dokumente des Linkpools geladen. Des Weiteren bauen viele Kriterien auf Messungen der anwendungsspezifischen Instrumentierung auf. Können Fehler in Zellen oder Crashes innerhalb eines Versuches nicht zuverlässig erkannt werden, ergeben sich daraus Unschärfen für die Versuchsauswertung.

Risiken der externen Gültigkeit sind zum einen die Art des Versuchsaufbaus mit unterschiedlich leistungsstarken Systemen, unterschiedlichen Betriebssystemen und sehr spezifischen SUT. Zum anderen stellt der Bezug zur besonderen Struktur der generierten Dateigraphen ein Risiko dar. Dass intermediäre Dateien innerhalb der gerichteten Graphen lediglich Verweise und keinerlei Daten beinhalten, ist keine zwingende Notwendigkeit. Denkbar ist ein Multifile Fuzzing Ansatz, in der die Dokumente auch gemischt Daten und Verweise beinhalten. Auch die Art der zufälligen Datenerzeugung könnte einen weiteren Einfluss auf die Effektivität des Fuzzers gehabt haben. Vorstellbar wäre ein Generator, welcher in der Lage ist komplexe, semantisch korrekte Formeln zu erzeugen und so schon mit unverzweigten Dateien mehr Crashes zu provozieren. Die Art der zufälligen Zelleninhalte kann großen Einfluss auf ihre Wirksamkeit beim Laden über einen Verweis in einem anderen Dokument gehabt haben.

5.4 Fähigkeiten der Realisierung

Im Rahmen dieser Arbeit hat die realisierte Software, bestehend aus Server- und Client-Anwendung, in 35 konfigurierten Experimenten über 51000 Office OpenXML Workbooks erzeugt. Diese Workbooks sind summiert über 60 GB groß und beinhalten über 34 Millionen einzigartige Verweise. Die reine Versuchsdurchführung, ohne Versuchsvorbereitungen, für 67000 Versuchsergebnisse dauerte summiert über 32 Tage. Währenddessen wurden 12 Millionen Datenpunkte für Health-Metriken erfasst. Dies unterstreicht die Skalierbarkeit der Realisierung, da sowohl Frontend als auch Backend weiterhin erwartbar reaktionsfähig sind.

Aus der Nutzerperspektive erscheint der Dokument-Server zuerst über das Frontend. Dieses ist in der Lage Experimente und Clients zu verwalten. Experimente können für jeden Client zugewiesen und jederzeit neu priorisiert werden. Ein wesentliches Merkmal ist die Visualisierung von Event-Daten, wie dem Experiment-Fortschritt

oder die grafische Darstellung von aktuellen Health-Metriken. Der Generierungsfortschritt kann durch den Konsolen-Output verfolgt werden. Eine detaillierte Visualisierung der Ergebnisse einzelner Experimente kann gesammelt eingesehen oder für jeden Client separat heruntergebrochen werden. Für jedes Ergebnis kann der Dokumentgraph des zugehörigen Versuchs als Archiv heruntergeladen werden, um ihn separat zu untersuchen.

Die asynchrone Generierung neuer Versuchsdokumente erlaubt eine unterbrechungsfreie Versuchsdurchführung durch die Clients. Darüber hinaus kann die Generierung der Versuchsdokumente jederzeit unterbrochen und wieder aufgenommen werden, sollte ein Neustart des Dokument-Server notwendig sein. Zusätzlich ist die Art der Persistenz der Versuchsdateien vollständig abstrahiert und so jederzeit austauschbar. Des Weiteren werden die Experimente und ihre korrespondierenden Versuchsergebnisse, abstrahiert von der eigentlichen Datenbank, relational gespeichert. Der andauernde Fortschritt der Clients wird über Watchdogs überwacht, ungewöhnliche Health-Metriken oder ausbleibende Rückmeldung von Ergebnissen werden dem Nutzer gemeldet. Die Veröffentlichung von Event-Daten über Websockets und eine strukturierte REST-API machen die Daten für Drittsysteme, wie alternative Frontends, zugänglich.

Der Generierungsvorgang bietet besonderes Potential für zukünftige Verbesserungen. So kann zwar die Generierung neuer Versuchsdokumente unterbrochen werden, nicht jedoch die, des initialen Linkpools. Wird der Dokument-Server bei der Generierung des Linkpools unterbrochen, wird dieser Vorgang nach der erneuten Anfrage des betreffenden Experiments neu gestartet. Es verbleiben keine Fragmente des vorherigen Vorgangs in der Datenbank oder der Datei-Persistenz, da der Linkpool erst nach seiner vollständigen Generierung an andere interne Services des Dokument-Servers weitergegeben wird. Auch der Zeitpunkt der Linkpool-Generierung kann weiter optimiert werden. In der Realisierung beginnt die Vorbereitung eines Experiments, wenn dieses zum ersten Mal durch einen Client angefragt wird. Es ist denkbar diesen Vorgang bereits zu starten, wenn ein Client ein vorhergehendes Experiment fast vollständig abgeschlossen hat. Dadurch würden nicht ausschließlich Versuchsdateien, sondern auch Linkpool-Dokumente, parallel zur Versuchsdurchführung generiert werden.

Die clientseitige Realisierung erfüllt ihre Kernaufgaben robust. Health-Metriken werden über einen best-effort Mechanismus erhoben und gemeldet. Die anwendungsspezifische Instrumentierung wird über ein einheitliches Interface der Client-Implementierung zugänglich gemacht. Dieser Ansatz erleichtert es künftigen Nutzern eigene anwendungsspezifische Instrumentierungen zu implementieren. Die anwendungsspezifische Instrumentierung für Excel erfasst Timeouts und Crashes zuverlässig. Im nächsten Schritt sollte die Versuchsdurchführung, in Bezug auf die anwendungsunabhängige Instrumentierung, so erweitert werden, dass es möglich ist, die SUT über die Prozessverwaltung des Betriebssystems zu beenden. Tritt in

der realisierten Software ein Timeout während des Öffnens der eines Versuchsdokuments auf, wird frühestens danach die Versuchsdurchführung unterbrochen. Eine optimierte Implementierung könnte eine SUT nach einer gewissen Wartezeit gezielt beenden, um das übermäßige Warten auf ein Versuchsdokument zu verhindern.

6 Zusammenfassung

In dieser Arbeit wurden verschiedene Gesichtspunkte des Multifile Fuzzing von Office Open XML Workbooks betrachtet. In Kapitel „2 Grundlagen“ wurde auf maßgebliche Konzepte eingegangen. Dieser Abschnitt erläutert Konzepte des Software-Testing, Fuzzing und des „ISO/IEC 29500:2008“ Standards die für das Verständnis des Inhalts von Bedeutung sind. Das Kapitel „3 Architektur“ beschäftigt sich mit den abstrakten Architekturanforderungen, welche für die zu implementierende Software essentiell waren. Es werden Freiheitsgrade und Abwägungsprozesse des gewählten Software-Designs aufgezeigt und daraus folgende Designentscheidungen begründet. Die Abschnitte bezieht im Rahmen des Studienprojektes getroffene Annahmen mit ein und vereinbart diese mit den neuen Anforderungen der Aufgabenstellung.

Das Kapitel „4 Realisierung“ beschreibt das Ergebnis der Software-Entwicklung innerhalb dieser Ausarbeitung. Neben den grundsätzlichen Technologien werden die Fähigkeiten der einzelnen Komponenten des verteilten Systems erläutert und detailliert beschrieben. Im Vordergrund stehen dabei die maßgeblichen Bestandteile dieser Komponenten. Das Systemdesign folgt den Abwägungen aus „3.3 Kommunikationsmodell und Zustandsverwaltung“. Der für die Umsetzung zentrale Fuzzing-Generator, dessen Konzept unter „2.2.2 Semantisches Fuzzing“ beschrieben und in „3.1 Fuzzing-Generator“ spezifiziert wurde, ist in „4.2.1 Fuzzing-Generator“ in seiner umfangreichen Implementierungstiefe beschrieben. Auf die unterschiedliche Instrumentierung der beiden gewählten SUT „Microsoft Excel“ und „LibreOffice“ wird unter „4.3.2 Plattformunabhängige Instrumentierung“ näher eingegangen.

Das Ergebnis dieser Arbeit ist ein funktionierendes Fuzzing-Werkzeug, welches anwendungs- und plattformunabhängig Multifile Fuzzing-Kampagnen durchführen kann. Die Fähigkeiten dieses Werkzeugs wurden durch verschiedene Experimente explorativ erprobt. In Kapitel „5 Auswertung“ wird der konkret gewählte Versuchsaufbau beschrieben und die so erhobenen Daten strukturiert untersucht. Es werden drei Forschungsfragen gestellt, auf deren Beantwortung unter „5.2 Ergebnisse der Experimente“ eingegangen wird.

Die Abschnitte „5.4 Fähigkeiten der Realisierung“ und „6.1 Fazit und Ausblick“ zeigen auf, welche weiteren Schritte unternommen werden können, um die Effizienz des Systems zu steigern und die Aussagekraft der erhobenen Daten weiter zu erhöhen. Aufgrund der dokumentierten Architekturentscheidungen und durchdachten Implementierung sind diese Änderungen einfacher realisierbar. Auch eine Anpassung an zukünftige Anforderungen wurde während der Realisierung bedacht. Zukünftige Nutzer sind in der Lage über die Implementierung der definierten Schnittstellen, gezielt eigene Generatoren, Datei-Persistenzen oder anwendungsspezifische Instrumentierungen zu integrieren, ohne andere Teile der Software verändern oder verstehen zu müssen.

6.1 Fazit und Ausblick

Die Versuchsergebnisse, welche während der einzelnen Experimente erhoben wurden, zeigen einen positiven Einfluss auf die Effektivität beim Fuzzen von Office Open XML Workbooks durch die Nutzung von Multifile Fuzzing Methoden. Es können ebenfalls vorsichtige Annahmen über das Verhalten der SUT getroffen werden, welche die Entwicklung zukünftiger Fuzzer für Anwendungen dieser Art bereichern kann. Die umfangreichen, erhobenen Health-Metriken waren zur Überwachung der reibungslosen Versuchsdurchführung durch die Clients geeignet, jedoch für die Beantwortung der Forschungsfragen dieser Arbeit nicht relevant. Während der Versuchsdurchführung haben sich die Vorteile der umfangreichen Architekturüberlegungen gezeigt. Das verteilte System ist in der Lage seine Kernaufgabe zu erfüllen. Durch das Zusammenspiel zuverlässige Zusammenspiel der einzelnen Komponenten, können sich zukünftige Verbesserungen insbesondere mit der Komponenten-Funktionalität und nicht mit ihren Schnittstellen oder Interaktionen beschäftigen.

Die vollständig fehlenden Crashes in den Versuchsergebnissen des Linux-Clients mit LibreOffice als SUT zeigt zusätzliches Potenzial der realisierten Software. Zuerst sollte die Instrumentierung von LibreOffice gezielt untersucht werden, um zu evaluieren, ob eventuell aufgetretene Crashes nicht aufgezeichnet wurden. Ein weiterer Ansatz kann sein, andere Aspekte der Workbooks, wie Kommentare, Bilder oder Formeln, in den Fokus des Fuzzing-Generators zu rücken. Denkbar ist ein Multifile Fuzzing Ansatz über externe Bild-Referenzen [Int16, 15.2.14 Image Part]. Diese Aspekte der Workbooks könnten lohnende Ziele zukünftiger Erweiterungen sein. Das zusätzliche Aufschlüsseln einzelner Metriken der Versuchsergebnisse könnte ihre Vergleichbarkeit erhöhen. So kann es sinnvoll sein, die reinen Ausführungsdauern der Versuche in mehrere Phasen zum Öffnen und Auslesen der Dateiinhalte zu gliedern. Somit wird klarer, ob auftretende Timeouts mit der Dateigraph-Größe oder der Effizienz der Instrumentierung zusammenhängen. Außerdem können zusätzliche Health-Metriken bessere Einblicke bei Überlastungen der Client-Systeme gewähren. Da beim Multifile Fuzzing grundsätzlich ein Dateisystem involviert ist (siehe „2.2.3 File Fuzzing“), sollten neue Health-Metriken neben dem relativen, genutzten Festplattenspeicher zusätzlich die Festplatten-Auslastung, mit Bezug zu Lese- und Schreibvorgängen, sowie die Festplattentemperatur, umfassen.

Eine zentrale Datenhaltung der Versuchsergebnisse, durch den Dokument-Server in einer relationalen Datenbank, erleichterte die strukturierte Auswertung. Die umfangreiche REST-API des Dokument-Servers ist gezielt auf die Anforderungen des Frontends und der Client-Implementierung ausgelegt. Neben der aktuellen Frontend-Implementierung ist auch eine Referenzimplementierung in Python denkbar, um die Nutzung durch wissenschaftliche Evaluationspipelines zu erleichtern. Die umfangreiche Software, welche das Resultat dieser Arbeit bildet, ermöglicht es

Fuzzing-Kampagnen zu automatisieren. Ihre Komponenten bilden ein funktionierendes Gesamtsystem das seiner Kernaufgabe zuverlässig nachkommt.

Literaturverzeichnis

[But09] Paul N Butcher. *Debug it! : find, repair, and prevent bugs in your code / Paul Butcher*. eng. The Pragmatic programmers. Raleigh, NC [u.a.], 2009. ISBN: 9781934356289.

[Dav22a] Dr. Andrew Davison. *Andrew Davison's Home Page at PSU*. 7. Okt. 2022. URL: https://web.archive.org/web/20221007092620/http://fivedots.coe.psu.ac.th/~ad/index.html (besucht am 23. 08. 2023).

[Dav22b] Dr. Andrew Davison. *Java LibreOffice Programming*. 26. Juni 2022. URL: https://flywire.github.io/lo-p/index.html (besucht am 23. 08. 2023).

[Etz11] Opher Etzion. *Event processing in action / Opher Etzion ; Peter Niblett*. eng. Stamford, CT, 2011. ISBN: 9781935182214.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[Gir20] Arindam Giri. *LibreOffice Calc Reference to Another External Sheet or Workbook*. 16. Okt. 2020. URL: https://www.libreofficehelp.com/libreoffice-calc-reference-to-another-external-sheet-or-workbook/ (besucht am 17. 09. 2023).

[Gir23] Arindam Giri. *List of LibreOffice Calc Error Codes*. 30. Mai 2023. URL: https://www.libreofficehelp.com/libreoffice-calc-error-codes/ (besucht am 24. 10. 2023).

[Han02] Handelsblatt. *Sun will Microsoft jagen*. 21. Mai 2002. URL: https://www.handelsblatt.com/neues-office-paket-analysten-raeumen-dem-herausforderer-gute-chancen-ein-sun-will-microsoft-jagen/2167372.html (besucht am 08. 10. 2023).

[Hol20] Paul Holser. *Generator (junit-quickcheck-core 1.0 API)*. 21. Nov. 2020. URL: https://pholser.github.io/junit-quickcheck/site/1.0/junit-quickcheck-core/apidocs/index.html (besucht am 30. 10. 2023).

[Int06] ECMA International. *TC45 - Office Open XML Formats*. 7. Feb. 2006. URL: https://web.archive.org/web/20060901131601/http://www.ecma-international.org/memento/TC45-M.htm (besucht am 08. 10. 2023).

[Int16] ECMA International. *ECMA-376-1:2016 Office Open XML File Formats — Fundamentals and Markup Language Reference*. Dez. 2016. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-376-1_5th_edition_december_2016.zip (besucht am 08. 10. 2023).

[Jün22] Lukas Jünemann. *Mofuzz – Grammarbased Fuzzing referenzierbarer Dateien unter Nutzung von AFL*. 2022. URL: https://box.hu-berlin.de/f/65d9b96704f14f2da43a/.

[Kan99] Cem Kaner. *Testing computer software / Cem Kramer ; Jack Falk ; Hung Quoc Nguyen*. eng. 2. ed. New York [u.a.], 1999. ISBN: 0471358460.

[Kle+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei und Michael Hicks. „Evaluating Fuzz Testing“. eng. In: *CoRR* abs/1808.09700 (2018). ISSN: 2331-8422. arXiv: 1808.09700. URL: http://arxiv.org/abs/1808.09700.

[Mic08] Microsoft. *Microsoft Expands List of Formats Supported in Microsoft Office*. 21. Mai 2008. URL: https://web.archive.org/web/20090220141616/http://www.microsoft.com/Presspass/press/2008/may08/05-21ExpandedFormatsPR.mspx (besucht am 20. 03. 2023).

[Mic18] Microsoft. *Microsoft.Office.Interop.Excel Namespace*. 8. Juni 2018. URL: https://learn.microsoft.com/en-us/dotnet/api/microsoft.office.interop.excel (besucht am 22. 08. 2023).

[Mic21a] Microsoft. *Create an external reference (link) to a cell range in another workbook*. 30. Apr. 2021. URL: https://support.microsoft.com/en-us/office/create-an-external-reference-link-to-a-cell-range-in-another-workbook-c98d1803-dd75-4668-ac6a-d7cca2a9b95f (besucht am 17. 09. 2023).

[Mic21b] Microsoft. *XlCorruptLoad enumeration (Excel)*. 13. Sep. 2021. URL: https://learn.microsoft.com/en-us/office/vba/api/excel.workbooks.open (besucht am 23. 08. 2023).

[Mic22] Microsoft. *Workbooks.Open method (Excel)*. 30. März 2022. URL: https://learn.microsoft.com/en-us/office/vba/api/excel.workbooks.open (besucht am 23. 08. 2023).

[Mic23] Microsoft. *Zellfehlerwerte*. 7. Apr. 2023. URL: https://learn.microsoft.com/de-de/office/vba/excel/concepts/cells-and-ranges/cell-error-values (besucht am 24. 10. 2023).

[Mit+16] Brent Daniel Mittelstadt, Patrick Allo, Mariarosaria Taddeo, Sandra Wachter und Luciano Floridi. „The ethics of algorithms: Mapping the debate“. In: *Big Data & Society* 3.2 (2016). DOI: 10.1177/2053951716679679. eprint: https://doi.org/10.1177/2053951716679679. URL: https://doi.org/10.1177/2053951716679679.

[Moz23] MozDevNet. *MDN Web Docs – IRI*. 26. Aug. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/SVG/Content_type#iri (besucht am 17. 09. 2023).

[Pad+19] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis und Yves Le Traon. *Semantic Fuzzing with Zest*. ISSTA 2019. Beijing, China: Association for Computing Machinery, Juli 2019, S. 329–340. ISBN: 9781450362245. DOI: 10.1145/3293882.3330576. URL: https://doi.org/10.1145/3293882.3330576.

[Pao+05] Jean Paoli, Isabelle Valet-Harper, Adam Farquhar und Brian Jones. *Ecma TC45 - Office Open XML Formats*. 8. Dez. 2005. URL: https://web.archive.org/web/20111021022009/http://www.ecma-international.org/activities/Office%20Open%20XML%20Formats/TC45_GA_Dez05.pdf/ (besucht am 08. 10. 2023).

[Proa] LibreOffice Project. *com::sun::star::document::MacroExecMode Constant Group Reference*. URL: https://api.libreoffice.org/docs/idl/ref/namespacecom_1_1sun_1_1star_1_1document_1_1MacroExecMode.html (besucht am 23. 08. 2023).

[Prob] LibreOffice Project. *com::sun::star::document::UpdateDocMode Constant Group Reference*. URL: https://api.libreoffice.org/docs/idl/ref/namespacecom_1_1sun_1_1star_1_1document_1_1UpdateDocMode.html (besucht am 23. 08. 2023).

[Pro21] Google Guava Project. *CachesExplained*. 19. Apr. 2021. URL: https://github.com/google/guava/wiki/CachesExplained (besucht am 22. 08. 2023).

[Pro23a] Java Native Access Project. *Java Native Access (JNA)*. 14. Jan. 2023. URL: https://github.com/java-native-access/jna (besucht am 22. 08. 2023).

[Pro23b] LibreOffice Project. *LibreOffice Timeline*. 2023. URL: https://wiki.openoffice.org/wiki/Uno (besucht am 22. 08. 2023).

[Red19] Tony Redmond. *Office 365 Reaches 180 Million Monthly Active Users*. 2019. URL: https://office365itpros.com/2019/04/25/office-365-reaches-180-million-users/ (besucht am 18. 03. 2023).

[Şev23] Seyfi Şevik. „Thermal performance of the aluminum fin-copper wool heat sink for cooling Solid-State Drives“. In: *Icontech International Journal* (4. Juli 2023). ISSN: 2717-7270.

[SGA07] Michael Sutton, Adam Greene und Pedram Amini. *Fuzzing : brute force vulnerabilty discovery*. eng. Upper Saddle River, NJ [u.a.]: Pearson Education, 2007. ISBN: 9780321446114.

[Tan09] Andrew S. Tanenbaum. *Moderne Betriebssysteme / Andrew S. Tanenbaum*. ger. 3., aktualisierte Aufl. it, Informatik. München [u.a.], 2009. ISBN: 9783827373427.

[Vou88] M.A. Vouk. *On back-to-back testing*. 1988, S. 84–91. DOI: 10.1109/CMPASS.1988.9641.

[Win23] Rob Winch. *Introduction to the Spring IoC Container and Beans)*. 4. Mai 2023. URL: https://docs.spring.io/spring-framework/reference/core/beans/introduction.html (besucht am 30. 10. 2023).

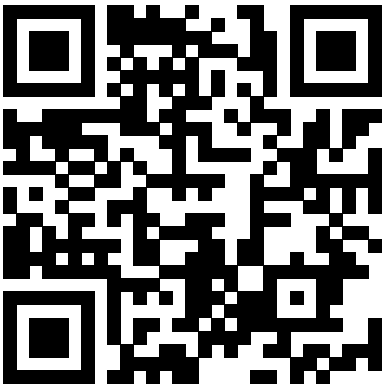
[Zal21] Michal Zalewski. *american fuzzy lop*. 2021. URL: https://lcamtuf.coredump.cx/afl/ (besucht am 10. 09. 2023).

Abbildungsverzeichnis

Quelltextverzeichnis

Zugang zum Sourcecode

Quelltext der realisierten Software ist über das öffentliche GitHub-Repository zugänglich. Es ist erreichbar unter:



`https://github.com/HU-Mofuzz/mofuzz-mf`

Nutzung von Markenzeichen

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig 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 3. November 2023 .....