TU WIEN Informatics

TECHNISCHE UNIVERSITÄT DARMSTADT

Improving REST API Robustness Through Continuous Fuzzing: A Case Study

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Daniel Haider
Matrikelnummer 01429078

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Florian Fankhauser

der Technischen Universität Darmstadt

Betreuung: Matthias Hollick
Mitwirkung: David Noel Breuer

Wien, 10. Juni 2023

Unterschrift Verfasser | Unterschrift Betreuung

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

TECHNISCHE UNIVERSITÄT DARMSTADT

Improving REST API Robustness Through Continuous Fuzzing: A Case Study

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Daniel Haider
Registration Number 01429078

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig
Assistance: Florian Fankhauser

at the TU Darmstadt

Advisor: Matthias Hollick
Assistance: David Noel Breuer

Vienna, 10th June, 2023

Signature Author   Signature Advisor

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Improving REST API Robustness Through Continuous Fuzzing: A Case Study

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Daniel Haider
Matrikelnummer 01429078

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 10. Juni 2023

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Daniel Haider

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. Juni 2023

Daniel Haider

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Fuzzing oder Fuzz Testing wurde in den letzten Jahren immer beliebter, da es eine leistungsfähige Ergänzung im Software Development Lifecycle (SDLC) bietet, um Robustheitsprobleme in Software-Artefakten frühzeitig ausfindig zu machen. Zusätzlich ist Continuous Integration (CI) mittlerweile der De-facto-Standard in modernen Software-Entwicklungsprozessen. Continuous Fuzzing, also die Integration von Fuzzing in die CI Pipeline, ist somit eine vielversprechende Ergänzung zur kontinuierlichen Lieferung von robusten Software-Artefakten.

Diese Arbeit untersucht anhand einer Fallstudie an einem Projekt im Bereich von Web-Application Programming Interfaces (APIs), wie frei verfügbare Fuzzing-Tools in eine kontinuierliche Entwicklungs-Umgebung integriert werden können. Durch eine Literaturrecherche wurde ein Design für eine Continuous Fuzzing Lösung ermittelt, welche anschließend anhand des Beispiel-Projektes implementiert wurde. Die implementierte Lösung führt eine kurze, 10-minütige Fuzzing-Kampagne bei jedem Commit unter Verwendung von zwei parallel laufenden Fuzzern durch, um Entwicklern rasches Feedback zu geben. Darüber hinaus wird bei jedem Merge-Request eine 50-minütige Fuzzing-Kampagne unter Verwendung eines White-Box-Fuzzers durchgeführt. Die Fuzzing-Ergebnisse werden zu einem Bericht zusammengefasst, der klare Anweisungen zur Reproduktion aller gefundenen Probleme enthält.

Die Evaluierung der implementierten Lösung simuliert die Verwendung von Continuous Fuzzing in einem Entwicklungsprozess, der sich über einen Zeitraum von über zwei Jahren erstreckt und 22 Commits enthält. Dabei wurden 51 Robustheitsprobleme im Projekt entdeckt, 13 davon einzigartig für alle Commits. 2 Fehler wurden in der neuesten verfügbaren Version der Software gefunden, wobei einer der Fehler nicht nur ein Robustheitsproblem, sondern auch ein Sicherheitsproblem darstellt, was die Relevanz der implementierten Lösung unterstreicht.

Diese Arbeit stellt den grundlegenden technischen Baustein bereit, der zeigt, dass Continuous Fuzzing ein vielversprechendes Werkzeug zur Steigerung der Software-Robustheit ist. Dies bildet die Basis für weitere Untersuchungen, um festzustellen, wie Continuous Fuzzing in verschiedenen Softwareprojekten in realen Entwicklungsworkflows erfolgreich angewendet werden kann.

Keywords: *Continuous Fuzzing, Fuzzing, Continuous Integration, Software Testing, Software Security*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

Fuzzing or fuzz testing has gained a lot of popularity in recent years as powerful addition to the Software Development Lifecycle (SDLC) to find robustness issues in software artifacts. In addition, Continuous Integration (CI) is now the de-facto standard in modern software development processes. Hence, bringing fuzzing into the CI pipeline is the next step towards delivering more robust software on a continuous basis.

In this thesis, a case study is conducted using a real-world inspired software project in the domain of web Application Programming Interfaces (APIs). The study investigates the feasibility of integrating readily available fuzzing tools into a continuous development environment. Through a thorough literature research a design for a continuous fuzzing solution was determined. The implemented solution fuzzes the test target in a quick, 10 minute long fuzzing campaign on every commit using two different fuzzers running in parallel to give developers rapid feedback. In addition, when issuing a merge request to merge the changes from a branch back into the main branch, a 50 minute long fuzzing campaign employing a white-box fuzzer was implemented. The fuzzing results are combined in a single report that provides clear instructions on how to reproduce any found issues.

An evaluation of the proposed solution which simulates the use of continuous fuzzing in a development process containing 22 commits stretching over the course of over two years detected 51 different robustness issues in the project, 13 of them being unique across all commits. 2 of the faults were discovered in the latest available version of the software, with one of the faults being not only an issue of robustness, but one which impacts the project's security, thus, demonstrating the usefulness of the implemented solution.

This study establishes the technical basis that demonstrates that continuous fuzzing can serve as a promising tool for enhancing the robustness of software. Building upon this, further investigations can be carried out to explore how this approach can be applied to various software projects in actual development workflows.

Keywords: *Continuous Fuzzing, Fuzzing, Continuous Integration, Software Testing, Software Security*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Motivation . . . 2
1.3 Expected Results . . . 2
1.4 Structure . . . 3

2 Theoretical Foundations **5**
2.1 Modern Development Process . . . 5
2.2 Web Application Interfaces . . . 18
2.3 Software Security . . . 20
2.4 Fuzz Testing . . . 26
2.5 Classification of Fuzzing Techniques . . . 30
2.6 Bug Oracles and Sanitizers . . . 33

3 From Fuzzing to Continuous Fuzzing **37**
3.1 The Idea of Continuous Fuzzing . . . 37
3.2 Requirements and Trade-Offs . . . 38
3.3 Readily Available Fuzzing Tools . . . 40
3.4 Existing Continuous Fuzzing Solutions . . . 60

4 Case Study: Proof of Concept Solution for Continuous Fuzzing **63**
4.1 Description of the Program Under Test . . . 63
4.2 Choosing Suitable Fuzzers . . . 64

5 Integration into GitLab CI/CD Pipeline **75**
5.1 GitLab CI/CD Concepts . . . 75
5.2 Continuous Fuzzing Design . . . 76
5.3 Results . . . 77

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.4 Evaluation 80

6 Discussion and Limitations **89**
6.1 Discussion 89
6.2 Limitations 92

7 Conclusion and Further Work **93**
7.1 Conclusion 93
7.2 Further Work 94

List of Figures **97**

List of Tables **99**

List of Algorithms **101**

List of Listings **101**

Acronyms **103**

Bibliography **105**

Web Resources **113**

Appendix **117**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

In this introductory chapter, the problem statement and the motivation of the thesis are laid out. Based on this information, the expected outcomes are outlined and four research questions are formulated. Lastly, the structure of the further chapters in this work is described.

1.1 Problem Statement

Software development processes have undergone major transformations since their beginnings. From classical waterfall models to the rise of agile methodologies, testing became more and more important and developers are now getting more feedback on their code artifacts than ever, especially with the rise of Continuous Integration (CI) and rigorous unit and integration testing, as shown by Fitzgerald and Stol [1].

One testing methodology that received more attention recently is fuzzing or fuzz testing. Miller et al. [2] describe it as a process which automatically tests software artifacts with randomly generated input data to reveal unexpected behaviour, software faults or vulnerabilities. While the idea of fuzzing is rather old in computer science terms (it was first proposed in the 1990s), the core concept has not changed much since then, according to Li et al. [3]. The way fuzzing is performed, however, has made drastic improvements [3].

Continuous fuzzing combines fuzz testing and CI, where CI is the practice of collecting code changes from multiple sources and integrating them into a single software project on a continuous basis, from which then automated builds and software tests can run [1]. By combining these two techniques, fuzz tests are run automatically during or after software integration, thereby providing valuable feedback to developers early on in the development cycle. Since Tassey [4] shows that finding faults early in the development process saves time and money, it is beneficial to fuzz-test software artifacts as soon and as

often as possible, making continuous fuzzing a valuable tool in modern software projects to ensure better software robustness and security.

1.2 Motivation

Programs like Google's OSS-Fuzz have been successful in identifying over 8900 software vulnerabilities [W1], and NIST's Recommended Minimum Standard for Vendor or Developer Verification of Code now mandates running a fuzzer [W2], underscoring the importance of fuzz testing. However, existing solutions focus on long-running fuzzing campaigns instead of providing rapid feedback to developers, and Rindell et al. [5] show that fuzzing solutions have a high perceived impact on the security of the software to be developed but are limited in adoption due to the high effort needed to build and maintain a fuzzing environment. Furthermore, as modular applications that communicate with each other via web Application Programming Interfaces (APIs) tend to replace large, monolithic applications (Kim et al. [6]), it becomes imperative to consider how fuzzers can be applied to such projects. The most common architectural style in this field is Representational State Transfer (REST), which leverages HTTP requests to access and manipulate resources [6]. Several fuzzing tools have been introduced by researchers and practitioners that exploit the implicit semantics introduced by the REST standard [6]. Nonetheless, to the best of our knowledge no existing continuous fuzzing solution provides an option to include such specialized fuzzers.

Moreover, according to Wang et al. [7], there exists a gap between academic evaluation of fuzzing tools/techniques and their real-world bug finding ability. Many papers in recent literature proposed new fuzzing tools and techniques, as shown by studies from Zhang and Arcuri [8] and Liang et al. [9]. However, as stated by Klooster et al. [10], not many studies deal with the real-world application of fuzzers and even less so with the continuous integration of them. This work tries to mitigate this gap between research and practice by creating a Proof of Concept (PoC) solution that integrates fuzzing tools into the Continuous Integration/Continuous Deployment (CI/CD) pipeline of a real-world application in a way that does not introduce too much configuration overhead.

The motivation behind this work is to increase assistance to developers during the software development process, resulting in safer and more robust code artifacts, without introducing too much configuration overhead. With modern fuzzing techniques utilizing evolutionary algorithms, feedback-driven fuzzing and by incorporating domain-specific knowledge, it is now possible to achieve high code coverage with fuzzing tests in reasonable time frames. This work shall evaluate how applying the concept of fuzz testing in a periodic manner through continuous fuzzing can benefit modern development processes.

1.3 Expected Results

The aim of this work is to explore the state of the art of fuzzing techniques, to find out which readily available fuzzers exist and how they can be integrated into a modern

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

development process, resulting in continuous fuzzing. It shall explore how different fuzzers can be integrated into a CI/CD pipeline and determine their advantages, drawbacks and limitations. A special focus lies on how the fuzzer's findings can be evaluated and incorporated into the development process in such a way that even developers without specific security or testing background can benefit from it.

The core hypothesis this work studies is that integrating continuous fuzzing solutions into a modern development process helps with detecting faults in code, therefore, improving software security and robustness in an early stage of the development cycle. To support this hypothesis the following Research Questions (RQs) shall be answered:

RQ1: How can state of the art fuzzers be integrated into a development process in a way that does not introduce too much configuration overhead?

RQ2: What benefits can short fuzzing campaigns in an early stage of the development life cycle provide?

RQ3: How can this process be automated in a CI/CD pipeline?

RQ4: How can the fuzzing tool results be evaluated and incorporated into the development process?

1.4 Structure

Chapter 2 establishes the fundamental principles and definitions that will serve as the basis for the subsequent chapters.

In Chapter 3, the idea of continuous fuzzing is introduced and the requirements and trade-offs of a continuous fuzzing solution are mapped out. Several readily available fuzzing tools are examined from a theoretical perspective and existing continuous fuzzing solutions are presented.

After these theoretical chapters, Chapter 4 introduces an example project on which a case study that explores the integration of fuzzers into the development process is conducted. Several experiments using readily available fuzzing tools are carried out to select the best suitable fuzzers for a continuous fuzzing integration.

Chapter 5 continues the case study by proposing a final design for a continuous fuzzing solution and implementing it using the GitLab CI/CD pipeline. Furthermore, the developed PoC is evaluated on the test target by simulating the development workflow using the project's history.

In Chapter 6, the results and the research questions are discussed. Furthermore, the limitations of the implemented PoC and the conducted case study are addressed.

Finally, Chapter 7 summarizes the key findings and offers some prospects for further research as well as possible extensions to the implemented PoC.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Theoretical Foundations

This chapter provides the theoretical background needed for the remainder of this work. To understand the context in which the case study is conducted in Chapter 4, a modern development process with its agile methodologies, software testing approaches, and CI is described. The basics of software security, attacks on software, and a means to defend against them via software security engineering are introduced. Lastly, fuzz testing including its different techniques and classifications is explained.

2.1 Modern Development Process

As highlighted by Leau et al. [11], software development processes have undergone major transformations since their beginning. From classical waterfall models to the rise of agile methodologies, testing became more and more important, and developers are now getting more feedback on their code artifacts than ever, especially with the rise of CI and rigorous unit and integration testing.

The Software Development Lifecycle (SDLC) is a crucial aspect of software development as it outlines the essential stages involved in creating software systems [11]. Traditional software development methodologies like the waterfall method or V-model follow a sequential process of requirements gathering, solution building, testing and deployment [11]. These methodologies require defining and documenting a stable set of requirements at the start of a project [11].

Leau et al. [11] and Stoica et al. [12] further describe the steps taking place in each of the characteristic phases in the traditional software development process [11]:

- Requirements: The requirements and a schedule for the implementation of the various phases of development are established. Potential risks that might arise during the later phases are assessed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Design: The design and architecture of the product is developed. Technical models or diagrams are created that serve as a road map for developers.
- Implementation: In this phase, code is produced for the first time as developers start implementing the previously established design.
- Testing: The testing phase often overlaps with the implementation phase to catch issues early on. After the project nears completion, the project is tested and reviewed by the customer before delivery.
- Deployment and maintenance: Once the product is fully tested, it is launched and maintenance is performed [12].

According to Stoica et al. [12], the requirements, design, implementation and testing phases are still present in incremental processes. However, instead of going through these steps only once, they are applied in each development cycle.

As laid out by Moniruzzaman and Hossain [13], the value, visibility, and adaptability of the project can be further increased by including the deployment phase in the incremental process to deliver a working and tested software on a continuous basis. The details of such agile methodologies are laid out in the following section.

2.1.1 Agile Methodology

„We are uncovering better ways of developing software by doing it and helping others do it" [W3]. With this sentence, a group of seventeen people introduced the *Manifesto for Agile Software Development* which had a lasting effect on how software was – and still is – developed. In it, they introduced four central values they favour over traditional ones: (i) Individuals and interactions over processes and tools, (ii) working software over comprehensive documentation, (iii) customer collaboration over contract negotiation and (iv) responding to change over following a plan.

In an analysis of agile software development methodologies and trends, Al-Saqqa et al. [14] describe these four values as follows:

First, focusing on abstract and formal processes as well as their technical surrounding environment is wrong. Instead, the quality of the human software developers and their relationships, interactions and communication is more important [14].

Second, while software documentation is a vital and valuable component in the agile software development process, it should not be exaggerated. If tested, working software is a more meaningful measure of progress than extensive documentation, since it answers immediately and is less ambiguous. This is especially important if requirements are changing frequently because writing up-to-date documentation is a time-consuming process [14].

Third, agile software development was invented to cope with changes in the requirements even late in the development cycle, hence, customer feedback and collaboration with the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

development team are essential. This is in contrast to traditional methodologies where a formal agreement and contracts dictated the stiff development process [14].

Lastly, since both the developers and customers will gain more knowledge of the system as the software development process progresses, it might be necessary to change the requirements. If this is the case, instead of following a strictly defined plan, it is encouraged to respond positively to the change to increase customer satisfaction [14].

Al-Saqqa et al. also emphasize that there are use cases where it is better to apply traditional software development methods instead of agile methods. For example, large projects with high budgets and multiple medium-sized teams working together should be thoroughly planned in advance, which contradicts agile principles. This is especially the case if formal processes or compliance standards have to be followed [14].

Hoda et al. [15] showed that 97% of organizations responding to a survey from the Annual State Of Agile Report 2018 practiced agile methodologies somewhere within their organization. In the latest Annual State Of Agile Report from 2021, 86% of software development teams now include agile practices in their development process, with Scrum being the most closely followed one at the team level with 66% [W4]. The rapid growth of agile practices also led to a new sub-discipline of software engineering in the last two decades and continuing today, named agile research [15].

The lasting transformation in how software is being developed also necessitated a change in how software is being tested.

2.1.2 Software Testing

Software testing is a crucial part of every software development process which aims to deliver high-quality software. The IEEE Standard Glossary of Software Engineering Terminology [16] defines software testing as „the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items“ [16]. Therefore, the goal of software testing is to find bugs, where a bug is a difference between an existing and the required condition. Since it is not always trivial to distinguish between a desired and an incorrect software behaviour, an additional mechanism is needed to make this decision. This is called the test oracle problem, as described by Barr et al. [17] (see Section 2.6).

At this point, it is also important to note the difference between errors, faults and failures. Although Rosenberg et al. [18] argue that these terms are often used interchangeably, they do have different meanings. According to the IEEE standard 610.12-1990 [16], an error is a „difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition“. A fault is an incorrect step, process or data definition, e.g., an incorrect instruction in a computer program [16]. Failure is defined as an incorrect result, e.g., a computed result of 12 when the correct result is 10 [16]. Thus, a software failure results due to the execution of a fault in the code which a developer placed there by mistake. However, not every fault must necessarily

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

lead to failure. For example, if a faulty instruction is unreachable and, therefore, never gets executed, it cannot lead to a failure. Bourque et al. [19] use the word *bug* as a more informal synonym of fault and if the distinction between fault and failure is not important, the more generic term *defect* can be used.

While software testing is a very broad subject, Graham et al. [20] argue that there are general guidelines applicable to all types of testing that have been proposed over the last four decades. They summarize 7 principles of testing introduced by the International Software Testing Qualifications Board (ISTQB) as follows [20]:

- Testing shows presence of defects: While testing can uncover defects in software, it cannot prove the absence of defects. Testing does, however, lower the likelihood of undiscovered defects persisting in the software.
- Exhaustive testing is impossible: Testing every possible input and precondition combination is not possible, except for simple cases. Therefore, testing efforts are prioritized and focused based on risks and priorities.
- Early testing: Testing should start as early as possible in the software or system development life cycle, with a clear focus on defined objectives.
- Defect clustering: A majority of defects discovered during pre-release testing or operational failures are found in a small number of modules.
- Pesticide paradox: Repeatedly running the same tests can lead to the pesticide paradox where no new bugs are discovered. To avoid this, test cases should be regularly reviewed, revised, and new tests should be developed to target different areas of the software or system to potentially uncover more defects.
- Testing is context dependent: Testing methods vary depending on the context. For instance, safety-critical software undergoes different testing than an e-commerce website.
- Absence of errors fallacy: Fixing defects is meaningless if the resulting system is unusable and fails to meet user requirements and expectations.

While the seven testing principles provide a solid foundation for software testing, there are many other aspects to consider. In the context of this work, the most relevant ones are detailed on the following pages.

Test Levels

It has been shown many times that the sooner a fault in software is found and corrected, the less costly it is [4]. The importance of the timing of finding bugs is also described by Takanen [21]. As the cost of bugs includes repair costs for developers and costs from damages to end-users, testing early in the product lifecycle reduces the cost per bug

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

compared to finding a flaw after release [21]. This is because the costs from damages are reduced or even eliminated if a fault never makes it in a build that gets released.

As a result, numerous software testing methods and tools have emerged which can be applied at different test levels. Test levels are typically distinguished based on the object of testing or test target, which can be single modules, a group of models or an entire system [19]. Accordingly, three common stages of testing are possible [19]:

- Unit tests are tests of isolated software elements that are separately testable [19], for example, functions or classes. They are mostly conducted by the developers that wrote the code.
- Integration or service tests aim to verify the interaction or interfaces between software components. This can either happen iteratively or at once („Big Bang").
- Lastly, the goal of system or User Interface (UI) tests is to verify the behaviour of the whole integrated system and assure that the entire system meets its requirements.

While Bourque et al. do not deem any of these stages to be more important than another one, Cohn [22] defines a test pyramid where such a distinction is indeed made. Figure 2.1 shows an illustration of the test pyramid.

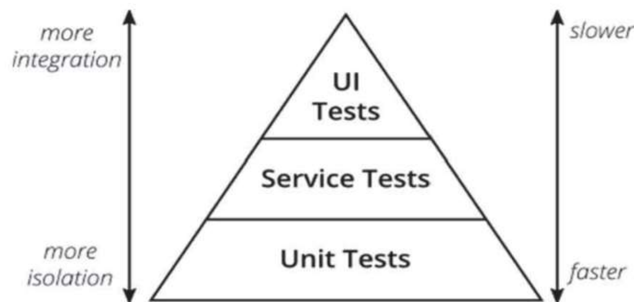


Figure 2.1: Cohn's Test Pyramid Illustrated by Mukhin et al. [23]

Since unit tests are relatively cheap to write, execute fast and must not be re-written that often if requirements change, they are a solid foundation at the bottom of the pyramid [24]. On top of that, selectively chosen component and integration tests should be placed and only a few end-to-end system tests should be on the top of the pyramid [24]. Because they are expensive to write, execute slowly and must be adapted even if only a small change in the UI is made, they should be carefully selected and used scarcely [24].

Note that there also exists a fourth test level called acceptance tests, however, this is fundamentally different than the previous ones, as it is usually undertaken by the customer and developers may or may not be involved in this testing activity [19]. It is conducted to ensure that a system meets its acceptance criteria, which involves comparing desired system behaviors against customer requirements [19].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Invalid Partition	Valid (3% interest)		Valid (5% interest)		Valid (7% interest)
-$0.01	$0.00	$100.00	$100.01	$999.99	$1000.00

Table 2.1: Examplary Equivalence Partitions by Graham et al. [20]

Equivalence Partitioning

As pointed out previously, exhaustive testing is impossible. Thus, it is essential to limit the test cases while still comprehensively testing the software. One intuitive approach to achieve this is by dividing or partitioning a set of test conditions into groups or partitions that can be considered equivalent [20]. The technique can be applied at any test level and assumes that if one condition in a partition works, all the conditions in that partition will work, and vice versa. By testing only one condition from each partition, a wide range of scenarios can be covered effectively [20].

To illustrate the technique, Graham et al. [20] describe a scenario in which a software is tested that calculates interest rates for a savings account based on the account balance. The various interest rates are determined based on the balance ranges. From $0 to $100, the interest rate is 3%. For balances over $100 up to $1000, the interest rate is 5% and for balances of $1000 and above, the interest rate is 7%. In this case, there are three valid equivalence partitions, as shown in Table 2.1. Note that there is also one invalid partition, i.e., if the balance is less than zero. By selecting test cases from these partitions, such as -$10.00, $50.00, $260.00, and $1348.00, comprehensive coverage is ensured [20].

This particular testing approach, known as a black-box or specification-based technique, does not rely on any knowledge of the software's implementation [20]. In contrast, there are white-box testing techniques that utilize the internal structure of the software to generate test cases [20].

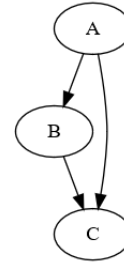
Test and Code Coverage

One important white-box technique in software testing is test coverage analysis as explained by Spinellis [24]. The test coverage indicates which parts of the code got executed during tests and, therefore, can give valuable information on which parts of the code need to be tested more thoroughly [24]. This is best explained with the Control Flow Graph (CFG) of a program. Oh et al. [25] define a program graph or CFG as a graph consisting of a set of vertices representing basic blocks and a set of edges representing possible control flows between basic blocks. A basic block is „a maximal set of ordered instructions in which its execution begins from the first instruction and terminates at the last instruction" [25]. An example for a simple CFG with its corresponding program's source code is given in Figure 2.2. The first basic block in the program consists of line numbers 1–3, the second one is line number 4 and the last basic block is line number 6. These basic blocks are mapped to the nodes A, B and C in the CFG respectively and the edges in the graph correspond to the control flows, i.e., the first basic block (A) ends with a branch that either goes through B to C (if x equals 42), or directly to C.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1 x <- INPUT()
2 y <- 0
3 if (x == 42) {
4   y <- y + 1
5 }
6 z <- x / y
```

(a) Example Program



(b) Resulting CFG

Figure 2.2: Example Program and Corresponding CFG

Based on the CFG, Weiser et al. [26] define three different coverage metrics:

- Basic Block Coverage: Also known as statement or line coverage, this simply divides the number of blocks (or statements) that got executed at least once, by the total number of blocks (or statements).
- Branch Coverage: Measures which branches of the program, i.e., which edges in the CFG were executed at least once.
- Path Coverage: Measures the execution of edges too, but additionally takes into account the number of times an edge got executed. As programs containing loops have potentially infinite possible paths, it is hardly possible to test all of them.

The program in Figure 2.2 initially assigns 0 to y and if a user-supplied value equals 42, y is increased by 1. At the end, x is divided by y and the result is assigned to z. 100% basic block coverage can be achieved with one single test where x is set to 42. However, if any other value is chosen for x, the program crashes due to a division by zero. This demonstrates that achieving 100% basic block coverage is not a guarantee for bug-free programs.

Code Instrumentation

A way to obtain information on code coverage about a test target commonly used by fuzzing tools (see Section 3.3) is code instrumentation, as described by Tikir and Hollingsworth [27]. In static code instrumentation, code is added during program compilation or linking, thus, adding it to the binary executable file. Most tools simply add code (e.g., a counter) at the beginning of each basic block to measure if a block got executed or not [27].

Another approach shown by Ramasamy and Hundt [28] is to dynamically instrument a binary during runtime. This eliminates the need for recompilation of the test target and is especially useful if its source code is not available. With dynamic instrumentation, it is also possible to remove instrumented code during runtime [28]. For example, when

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

measuring basic block coverage, the instrumented code that marks the block as executed can be removed after its first execution. Thus, for blocks that are executed many times during a program run, computational overhead is reduced.

Symbolic Execution

To further decrease the test cases needed while still covering the whole program, King [29] introduced a mechanism called symbolic execution. Like the previously described equivalence partitioning technique, it involves dividing the input space into representative classes. However, with symbolic execution, this is done by analyzing the code in a white-box approach, whereas equivalence partitioning is a pure black-box technique [20]. In symbolic execution, a program is not called using normal inputs (e.g., numbers) but with symbolic formulas over the input symbols. At each node in the CFG that has multiple output edges, a distinction is made. For example, the statement `if x > 10` leads to two possible branches: In the first, x is greater than 10, in the second one, x is less than or equal to 10. This results in two classes of inputs where in each one, x must satisfy different constraints. During symbolic execution, the program explores different execution paths and collects the constraints for each path, which then form a logical formula that represents the conditions required to reach a specific program state or line of code. The symbolic execution engine can then attempt to solve these constraints to generate concrete test inputs that exercise different program paths. Therefore, instead of executing a program with a set of sample inputs, it is tested with a set of classes of inputs, where each class represents a different branch in the program's CFG [29]. While this set of input classes can still be practically infinite, it „provides better results more easily than normal testing for most programs“ [29].

Test Processes

Kettunen et al. [30] showed that agile development methodologies like Scrum and Extreme Programming (XP) require different testing strategies than traditional, plan-driven approaches. Since with agile methods the software is built and delivered piece by piece in smaller iterations, developers are encouraged to test their components parallel to development so that after each iteration a tested product can be delivered. With a plan-based approach, it is often the case that there is one large final testing phase which might be scaled down at the end of the project if there is not enough time to reach the deadline [30].

One paradigm that has emerged parallel to agile methodologies is the test-driven development process, which promised increased developer productivity and improved code quality. The core principle of Test-Driven Development (TDD) is the test-first approach, whereby unit tests are written before the code. The test should then fail, i.e., be red. After that, the functionality is implemented in such a way that only a minimal amount of code is added for the previously implemented test to succeed. Then, the code is refactored and the unit tests ensure that functionality is preserved, and no new bugs are introduced.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Despite its many promises, Karac and Turhan [31] have shown that after 15 years of studies on TDD there is no clear evidence of improved quality and productivity. They claim that the test-first approach is not the most important aspect of TDD. Instead, working on well-defined, small tasks in short and steady development cycles brings more benefits than obsessing over the test order [31].

Spinellis [24] summarized the state of the art in software testing and named several best practices that are applied in industrial practice. Besides the already described unit testing, test-driven development, establishing a test pyramid and test coverage analysis, successful test strategies should also include test automation and CI. The importance of test automation and CI in agile testing is also highlighted by Stolberg's [32] paper on enabling agile testing through CI. These concepts are described in detail in the following pages.

Test Automation

Given the fact that „testing consumes 30 to 60 percent of all life-cycle cost, depending on product criticality and complexity" (Polo et al. [33]), it is essential to manage and minimize expenses related to testing. Since testing cannot be eliminated altogether due to its importance, utilizing test automation is crucial. As opposed to manually testing that a software meets its requirements, tests should be automated according to the previously described test pyramid.

A more advanced degree of automation can be reached by automatically generating the test cases themselves, as outlined by Takanen [21]. Since test case generation can be modelled as an optimization problem in which a minimal set of small tests that covers the maximum amount of objectives is searched, different kinds of search algorithms can be applied to solve it, as described by Arcuri [34]. The set of tests is called the test suite and objectives to optimize for can be, e.g., line or branch coverage.

One such search algorithm is the Whole Test Suite (WTS) algorithm invented by Fraser and Arcuri [35]. Previous algorithms generated a test case for each coverage objective (such as branches in branch coverage) and then merged them into a single test suite. However, estimating the size of the resulting test suite can be challenging because a test case designed for one specific goal may implicitly satisfy a multitude of other coverage goals [35]. Moreover, the presence of infeasible coverage goals can pose a challenge, as targeting them would result in wasted effort. With the WTS algorithm Faser and Arcuri proposed, they improved on this approach by evolving all the test cases in a test suite at the same time while the fitness function takes into account all the testing goals at once [35].

However, according to Arcuri, the WTS algorithm does not scale well if the search budget is limited and if there is a large number of objectives, which is often the case for system level tests. Thus, he created the Many Independent Objective (MIO) algorithm which improves on these kinds of problems by exploiting characteristics that are specific to test suite generation [34]. Its core improvement lies in the dynamic tradeoff between

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

exploration and exploitation of the search landscape. At the beginning of the MIO algorithm, randomly generated test cases are preferred over mutated ones to cover a larger part of the search space. Then, as the search progresses, exploitation is preferred over exploration to increase the likelihood of covering single targets. This is because a test suite that actually covers one target is preferred over a test suite that heuristically almost covers 100 targets [34].

Another improvement Arcuri implemented to increase the time spent on targets that have a higher chance to be covered is a technique called feedback-directed sampling. It involves assigning counters to each non-covered/non-empty target and increasing the counter each time a test is sampled from the target's population. When a new better individual is added to the population, the counter is reset to 0 and the target with the lowest counter is chosen instead of choosing randomly. Thus, this technique concentrates on easier targets that are not covered yet [34].

Further Testing Techniques

Over the years, several different techniques for testing software artifacts more efficiently have emerged. Those relevant for this thesis are explained briefly:

- Combinatorial Testing (CT) leverages the fact that most software bugs or failures are triggered either by a single parameter, or the interaction between a pair of parameters, as shown by Kuhn et al. [36]. Interactions between three or more parameters gradually cause fewer bugs, while being exponentially harder to find. Thus, by focusing on covering all combinations of two or three parameters, most bugs can still be reliably found, using only a fraction of the resources compared to testing all combinations of all parameters (i.e., the whole input space) [36]. Wu et al. describe this technique more formally [37]. Given a set of n parameters $P = \{p_1, p_2, p_3, \ldots, p_n\}$, a test case is constructed by assigning each parameter p_i a value from a finite set V. Then, a combination of t parameter values is a t-way combination and a t-way covering array is a set of test cases, where every t-way combination is covered at least once. The parameter t is also called coverage strength [37].
- Property-based testing: As explained by Padhye et al. [38], in property-based testing the method to test is not called with absolute values defined by the tester, but with random values generated by the test framework. It must then be checked that if certain preconditions are met, the properties hold. Properties for a method concatenating two strings a, b could be that the length of the concatenated string must be $len(a)+len(b)$, but also that a and b must be a substring of the concatenated string.
- Constraint-based testing is described by Malburg and Fraser [39] as a method of generating test data by solving constraints produced by symbolic execution. Constraints introduced along a path in a program (e.g., by if-clauses) are collected,

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

after which a constraint solver can derive inputs that satisfy the constraints to follow the path [39].

- Adaptive random testing: As described by Huang et al. [40], the random testing technique generates test cases by randomly selecting values from the input domain, i.e., the set of all possible program inputs [40]. They show that this technique can be improved by exploiting the fact that failure-causing inputs tend to cluster into contiguous regions [40]. This means that if an input causes a failure, then its neighbors are also highly likely to cause a failure. Moreover, the same holds for non-failure-causing inputs: If an input does not cause a failure, then its neighbors are also highly likely to not cause a failure [40]. Adaptive random testing takes this into account when generating test cases to improve its failure-detection capability compared to random testing [40].

2.1.3 Continuous Integration/Continuous Deployment

The concept of CI is one of the key elements in a modern development process [24], especially if agile methodologies are applied [32]. It was first introduced by Booch [41] and then adopted by Beck [42] when he proposed XP as a new style of developing software that is more flexible to changes in requirements. He suggests integrating newly written code with the current system after no more than a few hours. During this process, the system must be built from scratch and all tests must succeed, otherwise, the integration fails and the changes are discarded [42].

In the following years, this concept was refined and widely adopted, as shown by Zhao et al. [43], with Martin Fowler's practices of CI [W5] being now widely considered as the de-facto standard for an effective CI. These 11 best practices are described briefly [W5]:

i) Maintain a Single Source Repository: Source Code Management Tools like Git allow many developers to work on the same code base. Everything needed to build the software has to be put into a single repository and one should not overuse the branch feature. Most importantly, there should be one branch called mainline (in Git this branch is commonly called main or master) from which every developer should work off.

ii) Automate the Build: It should be possible to bring in a freshly set up machine, check out the repository and build and run the system with a single command. No manual tasks (like copying files or loading database schemas) should be necessary. Build time should be kept to a minimum, e.g., by selectively choosing which targets are built based on changed files.

iii) Make Your Build Self-Testing: In addition to building and running the system, it should also be tested automatically. This is done via an extensive test suite that checks a large portion of the code for faults. If any test fails, the build should be aborted.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

iv) Everyone Commits To the Mainline Every Day: The typical commit process starts by updating the local working copy with the changes from the mainline. Thereby, updates from other developers are merged and locally built. If this passes, it can be pushed to the mainline. By going through this process frequently, conflicts between two developers are found early and can be fixed promptly.

v) Every Commit Should Build the Mainline on an Integration Machine: To keep the mainline healthy and take environmental differences from developer's machines out of the equation, the mainline should be built on a neutral machine after every commit. Only after this build passes, the commit should be considered done. This can either be done manually or automatically via a CI server. A CI server monitors the repository for new commits and on detection, it checks out the sources, builds it on an integration machine and notifies the developer about the result.

vi) Fix Broken Builds Immediately: If the mainline build breaks, it should be addressed immediately to get it running again. Most often, this is done by reverting to the latest running build and then debugging the problem on a development workstation.

vii) Keep the Build Fast: Rapid feedback is one of the key aspects of CI, therefore, builds should finish within a reasonable timeframe, e.g., 10 minutes. Since the usual bottleneck here is testing, Fowler proposes a deployment pipeline or staged build. Here, multiple builds are done in sequence, where the first one is called the commit build which should be done quickly. This can be achieved by omitting certain longer-running tasks, for example, system tests. After the commit build (including the quick unit tests) passes, other developers can work from the updated mainline with certain confidence since most bugs should be caught in the first stage. If it happens that a bug is found only in a later stage, it should also be addressed with high priority, although not as promptly if it occurs in the first stage. After the bug is fixed, a test should be added to the first stage so that it will not slip through in future builds.

viii) Test in a Clone of the Production Environment: Since the point of testing is to find problems that might occur in the system in a production environment, the testing environment should resemble the production environment as closely as possible. Otherwise, with every gap, a risk is added that the system behaves differently during testing and in production.

ix) Make it Easy for Anyone to Get the Latest Executable: Every stakeholder in a software project should be easily able to get the latest executable of the system. This helps especially in agile environments where requirements can change often, because any stakeholder can see the latest changes and intervene if something seems not quite right.

x) Everyone can see what's happening: The team should always be aware of the current build state, for example, via a continuous display that everyone can see.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

If an automatic CI process is used, a CI server's web page can display additional information like changes that were made.

xi) Automate Deployment: Since the project probably needs multiple environments to do different stages of testing, the process of deploying the system to a new environment should be automated. Therefore, it makes sense to reuse these scripts to also automate deployment to production to speed up the process and reduce errors.

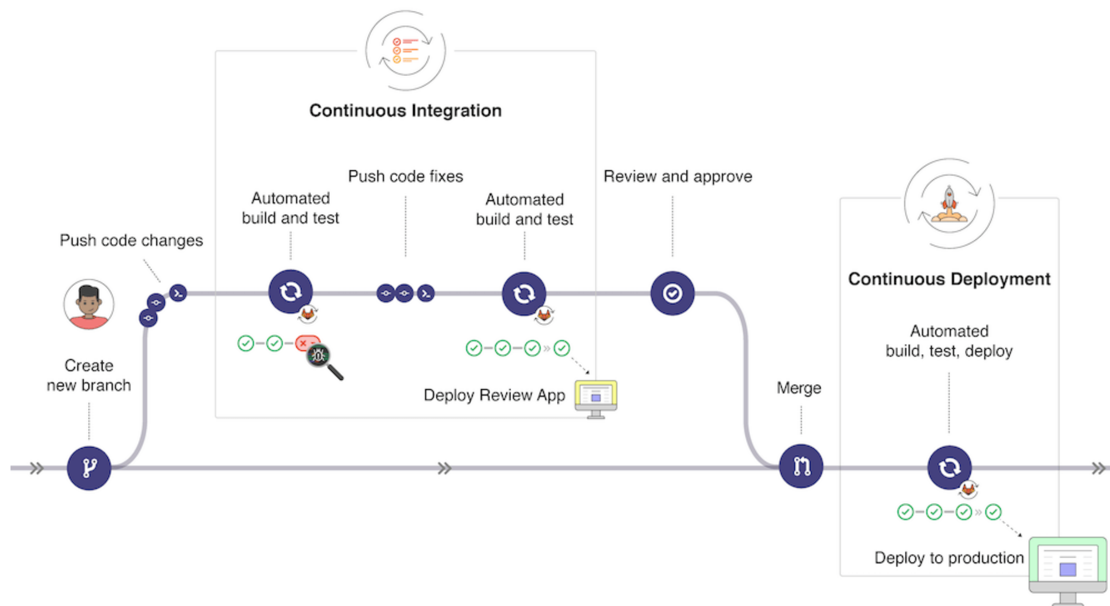


Figure 2.3: GitLab CI/CD Workflow [W6]

The core workflow of CI/CD following the described best practices is illustrated in Figure 2.3 [W6]. First, a new branch based on the mainline is created. Upon pushing code changes to this new branch, a set of scripts that build and test the application is executed. If the build or the test suite fails, the process is aborted and commits fixing the code must be pushed. When all errors are fixed and the build and tests succeed, the application can optionally be deployed to a staging environment. Before merging the branch back into the mainline, it must be reviewed and approved. Furthermore, there also exists an optional continuous deployment step, in which the application is built and tested again (on a neutral machine) and then deployed to production.

In contrast to TDD, there is clear empirical evidence supporting the claimed benefits of CI, as Hilton et al. [44] have shown. The benefits they found include helping detect faults early and making developers less worried about breaking the build [44]. In addition, projects using CI release more than twice as often and integrate pull-requests faster than projects without CI [44]. Because of these positive outcomes, CI should be treated as best practice and be widely adopted [44].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Note that, while this covers the basic theoretical concepts of CI, a more practical view on CI and especially GitLab's CI/CD pipeline is given in Section 5.1.

2.1.4 Pull-Based Development Model

With the trend of distributed software development, a new strategy for merging code changes from different developers was needed. The so-called pull-based development or fork and pull model is now widely used in such situations, especially in the open source community, as shown by Yu et al. [45].

Its core concept is that changes are not pushed to a central repository. Instead, a developer clones (i.e., forks) a public project to their local machine and implements their changes. This can be done by anyone without the need for any access rights or permissions to the source repository (as long as the source repository is public). After finishing their work, they issue a pull-request (also called merge-request) to the source repository in order for their changes to be merged back. The maintainers of the project can thus selectively choose the changes they want to implement by accepting or rejecting pull-requests. As keeping up with the volume of pull-requests can be difficult, CI is widely adopted to build and automatically test all incoming pull-requests [45].

2.2 Web Application Interfaces

Kim et al. [6] show that the trend in software development to move away from large, monolithic applications in favour of smaller services that communicate with each other, as well as the rise of cloud computing, has led to APIs such as REST and GraphQL becoming quite popular. In addition, clients often can access modern web services via an API they provide [6].

2.2.1 REST

The REST architectural style was described by Fielding in his PhD thesis [46]. Rodríguez et al. [47] summarize its core principles, which are „aimed at fostering scalability and robustness of networked, resource-oriented systems based on HTTP“:

- Resource addressability: APIs manage resources, where a resource can be „any information that can be named“ [46], e.g., a person, a pet or a collection of other resources. Each resource is uniquely identifiable, e.g., via a Uniform Resource Identifier (URI) or Uniform Resource Locator (URL).
- Resource representations: Clients work with representations of resources and do not need to know the internal format. Content-type headers in the HTTP messages enable the client and server to negotiate representations to use (e.g., JSON or XML).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Uniform interface: The HTTP protocol with its standard methods (GET, POST, DELETE, etc.) dictates the behaviour of access and manipulation of resources. For example, a GET request should be used to retrieve a resource, while a POST request should be used to create a new one.
- Statelessness: The server does not store any state information. Every request must contain every information needed by the API to process it.
- Hypermedia as the engine of state: Resources can be linked to other resources, allowing the client to navigate relationships.

While there exist many different specification formats to describe REST APIs, the OpenAPI specification is by far the most widely used one (see Hatfield-Dodds and Gygalo [48]). Its details are described in the next section.

2.2.2 OpenAPI

An OpenAPI (formerly known as Swagger) specification describes a REST service in a machine-readable format, e.g., YAML or JSON. It defines the requests available to retrieve or modify resources and which responses to expect from the server. An example for a pet store API from the official OpenAPI documentation [W7] is given in Figure 2.4.

Figure 2.4a describes the representation of a pet. It is of type object with the properties id, name, tags and status. The id is of type integer with the format int64. Name and status are strings, whereby status is an enum that can be assigned the values available, pending or sold. The tags property is of type array which contains several Tag items, denoted by the $ref key. Thus, a Tag is another defined schema just as the described Pet schema.

Figure 2.4b defines a resource available in the API, more specifically, the endpoint for retrieving the representation of a single pet object. Line 3 describes the HTTP method to use, lines 5–10 define that a path parameter with the name petId of type integer and format int64 is required. The endpoint has three expected responses: Code 200 with the representation of the retrieved pet in JSON format, code 400 if an invalid id (e.g., a string) was supplied, and code 404 if no pet could be found for the given id.

In addition to schemas and paths, an OpenAPI document has an OpenAPI object as its root node, defining important meta data like the version number of the OpenAPI specification the document uses, a summary and description of the purpose of the API and servers it is reachable at [W8]. To specify security mechanisms (e.g., authorization via a token in the HTTP header) used by the API, a security requirement object can be declared [W8].

There are many tools that build upon the OpenAPI specification to automatically generate clients that consume the API, auto-generate documentation from source code, or provide a UI to interact with the API in the browser [W9].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  Pet:
2   required:
3   - name
4   type: object
5   properties:
6    id:
7     type: integer
8     format: int64
9     example: 10
10   name:
11    type: string
12    example: doggie
13   tags:
14    type: array
15    items:
16     $ref: '#/components/
         schemas/Tag'
17   status:
18    type: string
19    enum:
20    - available
21    - pending
22    - sold
```

(a) OpenAPI Schema Definition

```
1  paths:
2   /pet/{petId}:
3    get:
4     parameters:
5     - name: petId
6      in: path
7      required: true
8      schema:
9       type: integer
10      format: int64
11    responses:
12     "200":
13      content:
14       application/json:
15        schema:
16         $ref: '#/components/schemas/Pet'
17     "400":
18       description: Invalid ID supplied
19     "404":
20       description: Pet not found
```

(b) OpenAPI Path Definition

Figure 2.4: OpenAPI Specification Document [W7]

2.3 Software Security

McGraw defined software security in 2004 as „the idea of engineering software so that it continues to function correctly under malicious attack“ [49]. With the increasing use of software components in basically every field of modern society, it is now more clear than ever that software security should be an integral part of modern software development, as shown by Khan et al. [50].

A related concept is that of software robustness which is defined by the IEEE standard 610.12-1990 as „the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions“ [16]. There are more formal and exact definitions of software security and robustness, however, for the scope of this work these informal definitions are adequate. Although both definitions deal with software functioning correctly under malicious attack or in presence of invalid inputs or stressful conditions, there is still a slight semantic difference. As shown by Laranjeiro et al. [51] and Miller et al. [2], issues in the robustness of a software component can lead to security issues. However, not every security issue can be found by testing the underlying software's robustness.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

According to Samonas and Coss [52], the main goals of software security from a practitioner's point of view are to protect the confidentiality, integrity and availability (also known as CIA triad) of the system. Confidentiality ensures that only authorized parties can access information, whereas integrity guarantees that no unauthorized party can modify information. A system functioning correctly is subsumed into the term availability, meaning that an intruder cannot prevent authorized users from accessing information [52]. As this is a rather limited and technical view, over the years socio-technical extensions, for example, authenticity and non-repudiation, have been introduced [52]. The inclusion of additional tenets (e.g., privacy or trust) is an ongoing debate. As such additions are often directly related to one of the original security goals or the intersection of two of them, Samonas and Coss [52] plead to re-define the semantics of the original three security goals to incorporate these enhancements [52].

2.3.1 Attacks on Software

AlBreiki and Mahmoud [53] argue that software is the main cause of computer security issues and that hackers exploit security vulnerabilities rather than creating them. These vulnerabilities in software applications stem from poor design and implementation of software systems [53]. Therefore, it is crucial to understand which types of vulnerabilities can be present in a software.

Since the case study presented in Chapter 4 deals with vulnerabilities in web applications, the most important threats present on the web are briefly explained. One project which classifies the most critical security risks in web applications and is also widely accepted in academic literature according to Fredj [54], is the Open Web Application Security Project (OWASP) [W10]. As laid out by Böhme et al. [55], many of the present works on fuzzing use simple bug oracles (see Section 2.6) to, e.g., detect program crashes. Thus, from the top ten web application security risks according to OWASP, the following are highly relevant for this work because they can be detected via such simple strategies.

Broken Access Control

This refers to a situation where a resource is inadequately safeguarded, resulting in users being able to access, modify or even destroy it, despite not having the appropriate authorization. A classic example is a resource that should only be accessible by user A, but can also be directly accessed by user B via its unique identifier [W10].

Consider, for example, a user with an internal integer ID with the value 42 in an application. They can update their profile via a POST request to /users/42. Now a curious adversary might try what happens if they send the same POST request to /users/41. If the application does not correctly control the access, a user might be able to update another user's profile this way. This can even lead to an adversary taking over other users' accounts if they can update the victim's e-mail address to one they control.

Another exploit of broken access controls is the path traversal attack as described by Doupé et al. [56]. In their example, a user can upload a photo which a vulnerable web

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

application saves to a subdirectory inside an upload folder. The user can specify a tag for the photo which the application will use as name for the subdirectory. If the user supplies a tag starting with "../../", they can escape the subdirectory and potentially overwrite files outside the upload directory [56].

Another type of attack is the exploitation of the redirection to untrusted sites, i.e., open redirects [W11]. Websites can redirect to other websites by setting the Location header in the response, upon which the browser receiving this header will follow the redirect. If a redirect URL can be manipulated by an attacker and is not validated properly, they can redirect to arbitrary websites. For example, an open redirect in `https://trusted.example?redirect_url=attacker.example` could be sent to phishing victims of `trusted.example`. Due to the initial link coming from a trusted website, „phishing attempts have a more trustworthy appearance" [W11].

Injection

If user-supplied input is not properly validated, filtered, or sanitized and then used in dynamic queries, system calls or similar, this might lead to injections. The OWASP project recommends including static, dynamic and interactive application security testing tools in the CI/CD pipeline to prevent injection vulnerabilities from entering production [W10]. Two common examples of this category are Structured Query Language (SQL) injections and Cross-Site-Scripting (XSS).

An application is vulnerable to SQL Injection (SQLi) if it directly uses unsanitized user input in an SQL statement, such that an attacker can „manipulate, create or execute arbitrary SQL queries" [56]. For example, if a query is created with simple string concatenations, e.g., "SELECT * from users where id = " + user_input it is vulnerable if no measures are taken to limit the user input to not contain characters that are interpreted as SQL. If no sanitizing is happening, a malicious user can directly manipulate the SQL statement by changing the contents of the user_input variable. To show all users, they can make use of the fact that "1=1" is always true and combine this with an OR clause. The payload would, therefore, be "1 OR 1=1" which executes the statement "SELECT * from users where id = 1 OR 1=1", thus, retrieving all users from the database. By using payloads with the UNION operator they could even retrieve records from different tables. An SQLi can be mitigated by filtering or sanitizing user input before the query is executed. This eliminates or escapes dangerous characters that would otherwise be interpreted by the database.

XSS attacks exploit unsanitized user input that is rendered in the Document Object Model (DOM) of a web browser, thus, allowing an attacker to execute code on a victim's machine in the context of the application [56]. A simple example is a website including a script tag as follows: <script>document.write("<h1>Hello, "+getQueryParam("name")+"</h1>");</script>, where the method getQueryParam retrieves the value of the "name" query parameter from the URL. If a malicious user calls the website with the query parameter "name=</script><script>document.alert(document.domain);",

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

it will be rendered inside the browser and the newly created script will be executed. An attacker can send a link containing such a malicious query parameter and execute arbitrary JavaScript in the context of the user who opens the link. Payloads include sending cookies to an attacker-controlled webserver to steal the user's session or interacting directly with the application in the user's context to perform malicious actions. Whereas in the illustrated example (called reflected XSS) a user must open a link, in a stored XSS the source of the untrusted input comes directly from the application itself instead of from the query parameters. This makes every user that visits a site of an application where the stored XSS payload gets rendered vulnerable. Again, this can be mitigated by escaping dangerous characters like "<" and ">" by using their HTML entity codes "<" and ">".

Memory Management Errors

Although not included among the top ten most significant risks, OWASP still considers memory management issues „well worth the effort to identify and remediate" [W10]. While web applications are usually written in memory-safe languages, these languages are written in languages that are not memory-safe and might contain memory issues that can be exploited [W10]. In addition, web applications using libraries that execute a binary application (e.g., an image manipulation tool that scales images uploaded by users) may expose themselves to potential buffer overflow attacks [W12]. Furthermore, Butt et al. [57] show that despite buffer overflow attacks existing for more than three decades, they are still a major threat and were the most frequently reported vulnerability to the Common Vulnerabilities and Exposures (CVE) program in 2019 [57]. Hence, the most common memory management errors are explained briefly:

- Buffer Overflow: The simplest form of a buffer overflow is the stack overflow [57]. When programs call a function, the address to be returned to is saved on the function stack frame. Local buffers are created on the same stack and if the bounds are not checked properly when writing to these buffers, an attacker can overwrite the current function stack frame [57]. By using a well-crafted payload, an attacker can write malicious code (e.g., a shellcode which opens a remote shell on the victim machine) into the stack frame and set the return address to the beginning of this payload [57]. Thus, when the function returns, the overwritten return address is read from the stack and the program continues with executing the malicious code [57].
- Use-after-free: Akritidis [58] describes use-after-free vulnerabilities as just as dangerous as buffer overflows, because they might allow an attacker to execute arbitrary code. Whereas in a buffer overflow attack the memory is accessed outside its prescribed bounds (i.e., spatial memory safety violation), in a use-after-free attack the memory is accessed after it is no longer valid (i.e., temporal memory safety violation) [58]. If a pointer is left pointing to freed memory and an attacker can control the memory area the pointer is referencing, they can manipulate the data

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

being read after the pointer is de-referenced again by the program [58]. While this was long dismissed as mere denial-of-service threats, Akritidis shows that this type of vulnerability has been used in the wild to execute arbitrary code on vulnerable targets [58].

- Integer Overflow: An integer overflow occurs when trying to store values in integer variables that are greater than the maximum value the variable can hold. For example, a signed integer with 32 bits has a maximum value of 2,147,483,647. If this value is exceeded, it will wrap around and start at the minimum value. As described by Dietz et al. [59], this is not a problem if the behaviour is well-defined, as is the case with most integer representations. However, with unsigned integers in C and C++, the behaviour is undefined which can result in serious vulnerabilities and even arbitrary code execution [59]. For example, an integer overflow vulnerability in OpenSSH which led to a buffer overflow allowed unauthenticated remote attackers to execute arbitrary code [W13].

2.3.2 Software Security Engineering

In the beginnings of software engineering, security has not been a concern for a long time. The academic discourse on how to build software securely started only in 2001 [49]. Over 20 years later, the field has come a long way and various methods, techniques and models have been proposed. Jafari and Rasoolzadegan [60] argue that „software security patterns are now a well-established means to encapsulate and communicate proven security solutions and introduce security into the development process".

Nevertheless, Khan et al. [50] have shown in a systematic mapping study that despite this variety of security approaches exists, they are often not explicitly included in most software development processes. They make a strong argument for security being integrated into every step of the SDLC [50]. Traditionally, this is done by following one of the many available models and frameworks, e.g., McGraw's Secure Systems Development Life Cycle or the Systems Security Engineering Capability Maturity Model [50].

However, according to Rindell et al. [5], these security practices are often in conflict with agile software development methods. More specifically, the use of different agile tools had a measurable effect on the choice of security engineering practices. Security frameworks often come with the need for extensive security reviews, audits and documentation, thereby violating the Agile Manifesto's core values. The key results of their study are that most respondents use agile and security activities together, whereby agile activities mostly focus on requirements engineering, implementation and extensive testing, and security activities concentrate on the verification phase [5]. They found that the perceived security impact of security engineering activities is bigger the earlier it is applied and that automated testing tools in combination with specific security testing methods and release-time security audits are regarded as most effective [5].

Fitzgerald and Stol [1] even coined the term „Continuous Security" which „transforms security from being treated as just another non-functional requirement to a key concern

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

throughout all phases of the development lifecycle and even post-deployment, supported by a smart and lightweight approach to identifying security vulnerabilities".

2.3.3 Software Security Testing

Testing software for security is an incredibly difficult task, particularly due to two of the testing principles outlined in Section 2.1.2. These principles state that testing can only reveal the presence of bugs, not their absence, and that exhaustive testing is impractical. This means that it cannot be completely ruled out that there are bugs in the software, even after thorough testing, since most of the time, the input space is just too large so that one cannot test all possible input combinations in their various sequences in a reasonable timeframe.

Thompson [61] further describes why security testing is hard and also highlights the differences between traditional software testing and security testing. In Figure 2.5, the intended functionality of an application is depicted as a circle and the actual behaviour is illustrated by an amorphous shape superimposed on it. As a project progresses throughout

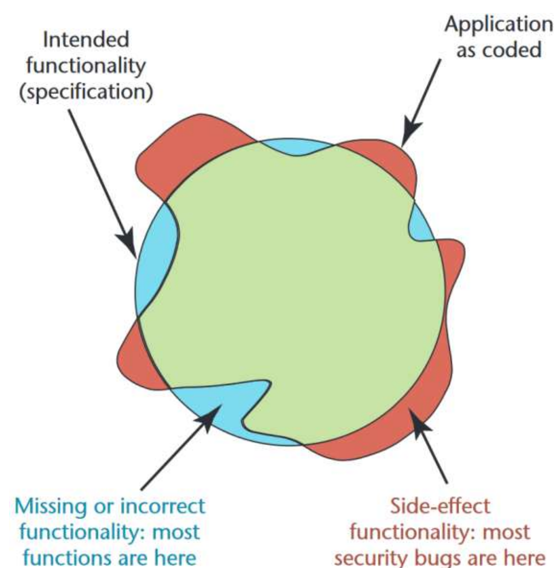


Figure 2.5: Intended Versus Implemented Software Behavior in Applications by Thompson [61]

its development stages, bugs are inevitably introduced [61]. This is represented by the blue section where the actual behaviour does not meet that of the specification. These type of deviations are fairly easy to detect via traditional testing in which the intended functionality is verified [61]. However, there is another type of bugs that are not as easy to detect because they introduce unanticipated functionality, i.e., side-effects, which are illustrated by the red area in Figure 2.5. Because they actually do cover the intended functionality as per the specification, traditional testing often misses such cases. Even as

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

early as 2003, Thompson advocated for the development of new tools that concentrate on this particular area [61].

Over the next two decades, several testing tools and techniques specializing on finding security issues have emerged. More specifically, security testing methods in the verification and release phase of the SDLC are now perceived as the most effective means to secure software components by software developers [5]. In the verification phase, common testing practices include automated testing tools, security-specific test cases, penetration testing, dynamic analysis and fuzz testing (see Section 2.4). However, while the use of automated testing tools has the highest usage rate and the highest perceived impact, there is a large discrepancy between the use and perceived impact of fuzz testing, i.e., it is deemed highly effective but is never used by most respondents [5]. Rindell et al. [5] explain this with the high effort needed to build and configure a fuzzing environment. As laid out in Chapter 1, one of the core objectives of this work is how to overcome this discrepancy and provide a way to apply fuzzing continuously without introducing too much configuration overhead.

Similar to Rindell et al. [5], Takanen [21] describes fuzzing as a very effective, proactive method to discover software vulnerabilities. He explains how the fuzzing process fits into the SDLC as described in Section 2.1, which also fits the problem described by Thompson. In the requirements gathering phase, it is defined how the software should function, resulting in positive requirements. Negative requirements, on the other hand, define how the software should not behave. This is similar to Thompson's illustration, whereas positive requirements can be seen as the circle, and negative requirements are everything outside of it, i.e., unintended side-effects. The goal of fuzzing is not to test the program's correct behaviour [21], which should be covered by manually written unit, integration or system tests (see Section 2.1.2). Instead, the challenging area of negative requirements is the main focus of fuzzing [21], which is described in detail in the next section.

2.4 Fuzz Testing

The term fuzz testing or fuzzing was first introduced by Miller et al. [2] during a class project at the University of Wisconsin in 1988. They describe it as a process which automatically tests software artifacts with randomly generated input data (called the fuzz) to reveal unexpected behaviour, i.e., crashes or the program becoming unresponsive.

As shown by Kelly et al. [62], random testing dates as far back as the 1950s when computers were as big as a room and punched cards were the main source of their programs and data inputs. Back then, it was standard practice to test early computer programs with random number punch cards or cards taken from the trash [62]. While this random testing methodology was not always seen as effective, Duran and Ntafos [63] showed in 1984 that random testing can indeed be a cost-effective and useful validation tool.

Despite these earlier publications on random testing, Miller et al. are now widely consid-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

ered the inventors of fuzz testing. According to them, the idea stems from the early days of the internet when one of the authors was logged on to his work computer via a dial-up line on „a dark and stormy night“ [2]. Due to the rain affecting the phone line and the resulting noise scrambling the commands the author was typing, several programs crashed or hung, including basic Unix utilities. This was because they could not handle these unusual characters introduced by the noise. Motivated by this scenario, Miller et al. conducted an experiment in which they were able to crash more than 24% of 90 different utility programs on seven different versions of Unix by testing them with random input strings. They also noted that one of the bugs the Morris worm exploited could have been found via fuzz testing, thus, highlighting the relationship between software robustness and software security.

In 1995, Miller et al. [64] published a follow-up paper in which they revisited fuzz testing. They found that the software reliability of Unix utility programs improved compared to their previous versions. However, with „only“ 18–23% crashing or hanging in 1995, compared to 25–33% in 1990, the failure rate still was severe. In addition to commercial systems, they tested the freeware GNU and Linux and were surprised that the freely available ones with a failure rate of 7–9% were much more reliable. They attributed this to the fact that commercial software must run on many platforms, configurations and operating system versions. Another explanation they pointed out was that long delays in corporate software (from an initial bug report to testing to releasing a repair one year can go by) discouraged programmers from fixing bugs. Hence, by the time a fix was released to the public, the programmer has long forgotten the bug and got no gratification from solving a particular person’s problem.

Over the years, the term fuzzing has evolved slightly and there are now multiple definitions and methods for it. For example, in the beginning, only random data was used to test programs [2], whereas modern fuzzers apply more enhanced test case generation strategies, as demonstrated by Beaman et al. [65]. Bekrar et al. [66] summarized different definitions into a single one: „Fuzzing is a security testing approach based on injecting invalid or random inputs into a program in order to obtain an unexpected behaviour and identify errors and potential vulnerabilities.“

Since this definition is still rather ambiguous, the terminology of fuzzing as described by Manès et al. [67] which will be used in the remaining of this work is explained briefly.

Definition 1 (Fuzzing) *Execution of the Program Under Test (PUT) using input(s) sampled from the fuzz input space that protrudes the expected input space of the PUT [67].*

Definition 2 (Fuzz Testing) *Fuzz testing is the use of fuzzing to test if a PUT violates a correctness policy [67].*

Definition 3 (Fuzzer) *A fuzzer is a program that performs fuzz testing on a PUT [67].*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Definition 4 (Fuzz Campaign) *A fuzz campaign is a specific execution of a fuzzer on a PUT with a specific correctness policy [67].*

Definition 5 (Bug Oracle) *A bug oracle is a program, perhaps as part of a fuzzer, that determines whether a given execution of the PUT violates a specific correctness policy [67].*

Definition 6 (Fuzz Configuration) *A fuzz configuration of a fuzz algorithm comprises the parameter value(s) that control(s) the fuzz algorithm (see Algorithm 2.1) [67].*

Furthermore, Klooster et al. provide two definitions regarding the compilation and execution of fuzzers:

Definition 7 (Fuzzing Harness) *A fuzzing harness acts as the main entry point for the fuzzer to reach the functionality intended to test [10].*

Definition 8 (Fuzz Target) *A fuzz target is a compiled fuzzing harness, i.e., the executable file resulting from the compilation of the harness [10].*

For each functionality in a software component that should be fuzzed, a harness needs to be written which provides access to this functionality [10]. Hence, many fuzzing harnesses and, therefore, fuzzing targets can exist for a single software library [10].

In addition to a clear definition of fuzzing terms, Manès et al. also developed a general algorithm of fuzz testing that accommodates existing fuzzing techniques [67].

Algorithm 2.1: Fuzz Testing Algorithm by Manès et al. [67]

Input: $\mathbb{C}, t_{limit}$
Output: $\mathbb{B}$ // a finite set of bugs

1 $\mathbb{B} \leftarrow \varnothing$;
2 $\mathbb{C} \leftarrow$ `Preprocess`($\mathbb{C}$);
3 **while** $t_{elapsed} < t_{limit} \land$ `Continue`($\mathbb{C}$) **do**
4 conf $\leftarrow$ `Schedule`($\mathbb{C}, t_{elapsed}, t_{limit}$);
5 tcs $\leftarrow$ `InputGen`(conf);
6 // O_{bug} is embedded in a fuzzer
7 $\mathbb{B}'$,execinfos $\leftarrow$ `InputEval`(conf,tcs,O_{bug});
8 $\mathbb{C} \leftarrow$ `ConfUpdate`($\mathbb{C}$,conf,execinfos);
9 $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$;
10 **end**
11 **return** $\mathbb{B}$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The Algorithm 2.1 takes as input a set of fuzz configurations $\mathbb{C}$ and a time limit t_{limit} after which the fuzzing campaign is stopped. The output is a finite set of bugs $\mathbb{B}$.

`Preprocess` is called on the set of fuzz configurations which may or may not update the fuzz configurations, depending on the fuzzer implementation. Code instrumentation (see Section 2.1.2) can be added to the PUT or the execution of seed files can be measured in this step [67].

The core of the algorithm is the while loop in line 3. Each iteration of it is called a fuzz iteration. The execution of `InputEval`, which runs the PUT on a test case, is called a fuzz run. A fuzzing campaign is stopped if the elapsed time reaches the specified time limit or if `Continue` ($\mathbb{C}$) returns false, which might be the case in white-box fuzzers that reached every possible path.

The `Schedule` function takes as input the set of fuzz configurations as well as the elapsed time and the time limit. It returns the single fuzz configuration that will be used in the current fuzz iteration.

`InputGen` takes the current fuzz configuration conf as input and returns a set of specific test cases tcs to be used for the evaluation of the PUT. There exist several different strategies for how to derive tcs from conf, e.g., grammar- or mutation-based. They are described in more detail in Section 2.5.3.

The actual execution of the PUT happens in `InputEval`. It takes as input the current fuzz configuration, the concrete test cases tcs and a bug oracle (see Section 2.6). It executes the PUT using tcs as input and the bug oracle evaluates if any of the executions violate the given correctness policy. The set of discovered bugs $\mathbb{B}'$ and some additional information on each of the fuzz runs collected in execinfos are returned.

In `ConfUpdate`, the fuzzing configurations can be updated based on the information on the fuzz runs in execinfos. For example, many grey-box fuzzers remove unnecessary fuzz configurations in this step [67].

A flowchart visualizing the described algorithm is given in Figure 2.6. The `Preprocess` step in the algorithm translates to the first step after start in the visualization. In the next step, the main fuzzing loop starts. A fuzz configuration is selected with which the test inputs are generated by using a seed, model or grammar (`InputGen`). Then, the test cases are applied on the test program and the bug oracle decides for each execution if an issue was found. This step corresponds to the `InputEval` step in the algorithm and the last step of the main fuzzing loop translates to the `ConfUpdate` step in the algorithm. The main fuzzing loop is executed until a predefined time limit is reached or until the fuzz configurations are exhausted.

As exhaustive testing is rather impractical (see Section 2.1.2), for most programs the time limit is the decisive factor on when a fuzzing campaign is terminated. Within academic circles, it is generally recommended to allocate a minimum of 24 hours for a comprehensive fuzzing campaign [10]. Section 3.2 offers insights on how to reduce the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

duration of this resource and time-intensive process when fuzzing continuously, although it involves making certain trade-offs.

Figure 2.6: Fuzzing Algorithm Visualization [65]

2.5 Classification of Fuzzing Techniques

In the past years, several attempts were made to classify fuzzing techniques. The most recent classification by Beaman et al. [65] refines and combines the previous ones from Li et al. [3] and Manès et al. [67]. As shown in Figure 2.7, Beaman et al. classify fuzzers based on four aspects: (i) Test Case Feedback, (ii) Knowledge of Application Structure, (iii) Test Case Generation Method and (iv) Program Exploration Approach.

The following sections describe each of them in detail.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Test Case Feedback

Dumb
Does not use the information gathered from program execution to update test case generation

Smart
Uses the information gathered from program execution to update test case generation

Knowledge of Application Structure

White-box
Full awareness of code structure and execution state

Grey-box
Partial awareness of code structure and execution state

Black-box
No awareness of application structure or execution state

Test Case Generation Method

Random
Test cases are randomly generated

Mutation-based
Creates test cases by modifying a set of valid inputs

Generation-based
Creates test cases based on a model of what valid program input is

Program Exploration Approach

Coverage-based
Attempts to test as much as the code as possible

Directed
Attempts to test certain parts of the code

Figure 2.7: Classification of Fuzzers by Beaman et al. [65]

2.5.1 Test Case Feedback

Beaman et al. [65] differentiate between fuzzers based on their ability to adjust their test input depending on the PUT's execution. Thus, fuzzers can be classified either as smart or dumb.

Dumb fuzzers use exactly one strategy to sample inputs from the fuzz input space and do not change it, no matter how the PUT reacts to it [65]. They usually have one big advantage: Since they do not use any information from the PUT (except to detect when it failed), dumb fuzzing campaigns can easily be automated and applied to many different programs.

Smart fuzzers, on the other hand, do change their sampling strategy based on the PUT's behaviour, leading to test cases more often finding a bug and, therefore, decreased testing time [65]. For example, a smart fuzzer might detect that a test case containing the string „ABC" reaches deeper into the program and, therefore, achieves greater code coverage. It will then use this information and adjust its input sampling strategy to incorporate this string with greater probability.

Early fuzzers (e.g., Miller et al.'s *fuzz* tool [2]) tend to fall into the dumb category, whereas newer tools are mostly smart [65]. However, it is interesting to note that even in 2020, Miller et al. were still able to get failure rates from 12–19% (compared to 18–23% in 1995) with their original dumb fuzzing strategy applied to 80 utility programs on Linux, MacOS and FreeBSD [68].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.5.2 Knowledge of Application Structure

Another commonly used classification is the fuzzer's knowledge of the PUT's internal structure [65]. Based on the amount of source code or execution state information a fuzzer utilizes, it is classified as black-box, white-box or grey-box [65].

Black-box fuzzers do not have any information about the PUT besides the input with which it is executed and the corresponding output. They are the most simple ones and many early types of fuzzers fall into this category.

White-box fuzzers, on the other hand, utilize a mechanism called symbolic execution [65] as explained in Section 2.1.2. Using basic block or branch coverage information, a white-box fuzzer can systematically explore the state space of the PUT [67], thus, guiding it to increase code coverage.

In fuzz testing there also exists a grey-box approach which is somewhere in between white- and black-box [67]. Thus, grey-box fuzzers utilize *some* information about the PUT's internals, but in a more simplistic or approximated way than white-box fuzzers [67]. They can perform lightweight static analysis of the PUT or gather information on code coverage via instrumentation (see Section 2.1.2) to generate input that reaches deeper into the code faster than their black-box counterpart [67].

2.5.3 Test Case Generation Method

Fuzzers can also be classified by their test case generation method, which can either be random, mutation-based or generation-based [65].

A random generation method simply generates input at random [65]. Miller et al.'s *fuzz* tool [2] is a prime example of this since all it does is generate a stream of random characters. While a fuzzer with a random input generation approach is simple to implement, it has one big disadvantage. When testing a program that expects an input of a certain type, for example, a JPEG image, it most probably validates at least the magic bytes of the given input file to ensure the correct file type. With the naive, random approach it would take many iterations just to pass this initial sanity check, and then it would most likely fail the next one.

Another approach is the mutation-based one where an initial set of valid inputs called *seeds* is created [65]. In the example above, a seed could be a valid JPEG image accepted by the PUT. Starting with these seeds, the fuzzer then mutates them in various ways, e.g., by randomly flipping or shifting bits around [65].

Generation-based fuzzers also do not start from scratch. However, unlike the mutation-based fuzzers, they do not use an initial set of valid inputs but utilize an underlying model to generate these valid inputs [65]. For example, they could have a grammar for a valid JPEG image with which they can generate input accepted by the PUT [65].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.5.4 Program Exploration Approach

The last distinction Beaman et al. [65] make is the way a fuzzer explores a program, which can either be directed or coverage-based. They are similar in that they use code-coverage information, however, how they use this information is different.

The goal of coverage-based fuzzers is to test as much of the code base as possible, whereas directed fuzzers are driven to test specific parts of the code [65]. While directed ones are usually faster to reach a certain area of the code, coverage-guided fuzzers can potentially find more bugs [65].

There exist several strategies for coverage-guided fuzzers to keep track of their progress. They are often used in white- or grey-box fuzzers via code instrumentation [65] (see Section 2.1.2). Simple approaches such as line or basic block coverage track the discovery of new lines or basic blocks the fuzzer reached, where basic blocks are small amounts of code defined by the fuzzer [65]. More complex strategies are branch-based. Like basic block coverage, basic branch coverage also keeps track of discovered blocks. However, it also adds the previously discovered block, thus, creating tuples which are used to measure progress [65]. N-gram branch coverage takes this process one step further by tracking not only the last two blocks but the last N blocks [65]. With a large enough N, the whole path of visited blocks is considered, which is why this approach is often called path coverage metric [65].

If the fuzzer also analyzes the data present in the call to a branch, Beaman et al. speak of context-sensitive branch coverage [65]. Similarly, memory-access-aware branch coverage utilizes memory access or state information to improve coverage and identify additional vulnerabilities, such as memory corruption [65].

2.6 Bug Oracles and Sanitizers

In software testing, the test oracle problem describes the challenge of distinguishing a normal, desired software behaviour from an incorrect behaviour [17]. Similarly, in fuzz testing, the bug oracle is responsible for determining whether a given execution of the PUT violates a specific correctness policy [67].

Since the canonical security policy used in fuzzing is whether a program terminates unexpectedly [67], most fuzzers simply detect if a program crashes or not, as shown by Böhme et al. [55]. Naturally, this detects many memory errors, since overriding a data or code pointer with an invalid address causes a segmentation fault or abort when the pointer is de-referenced [67]. While this approach is simple to implement since no code instrumentation is needed, it also has the drawback that many possible faults which do not lead to program termination are not detected.

This is why researchers have proposed sanitizers, a way to further improve the detection of specific classes of bugs, such as buffer overflows, use-after-free errors or signed integer overflows by adding code during compilation that crashes the program when a failure is

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

detected [67]. Therefore, by combining fuzzers with sanitizers, fuzzers can not only detect program crashes but also undesired program behaviour which might lead to security vulnerabilities.

Manès et al. [67] have found four separate categories that well-used sanitizers or bug oracles fall into: (i) Memory and Type Safety, (ii) Undefined Behaviours, (iii) Input Validation and (iv) Semantic Difference.

2.6.1 Memory and Type Safety

Memory and type safety sanitizers are used to detect addressability issues such as use-after-free errors or buffer overflow errors as described in Section 2.3.1. Furthermore, they can detect initialization order bugs or memory leaks [67]. A classical sanitizer which detects both, spatial and temporal memory errors is AdressSanitizer or ASan [67]. It instruments the code to create a shadow memory region that is linked to the main application memory, thus, manipulating a byte in the main memory also writes a value to the corresponding shadow memory [W14]. This allows it to quickly check each memory access for validity [67]. Compared to the non-instrumented code, ASan slows down program execution by 73% [67].

2.6.2 Undefined Behaviour

Undefined behaviour sanitizers are used to detect behaviours that are ambiguous in their language specification. This might lead to different compilers (or even different versions of the same compiler) behaving differently, which is prone to bugs if a compiler does not match the developer's expectation [67]. The classical sanitizers in this category are Memory Sanitizer (MSan), Undefined Behaviour Sanitizer (UBSan) and Thread Sanitizer (TSan). MSan detects uses of uninitialized memory in C and C++, UBSan can detect undefined behaviour such as division by zero or integer overflows and TSan is able to detect data races and deadlocks [67].

2.6.3 Input Validation

The third category of sanitizers is input validation. Detecting injection vulnerabilities such as XSS and SQL injections (see Section 2.3.1) is not a trivial task since the bug oracle has to understand the behaviour of complex parsers used in browsers or database engines [67]. One approach to detect XSS attacks was introduced by Duchene et al. [69] with KameleonFuzz. They extract the DOM using a real browser and then compare the tree to manually specified patterns to decide whether a test input triggered an XSS vulnerability. A more modern approach is used by Rooij et al. [70] in their tool webFuzz. Their payloads are designed to execute the JavaScript *alert* function with a distinct identifier, e.g., *alert('distinct_id_1')* [70]. They retrieve HTML responses and then apply a lightweight JavaScript code analysis tool named esprisma to infer if a function call to alert is present in the response [70]. If it is, an XSS vulnerability was found and via the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

distinct identifier, they can match the response to the initial request that triggered the XSS [70].

To detect SQL injection attacks, similar tricks can be applied [67]. Since it is not reliably possible to detect SQL injections from the response of a web application, µ4SQLi utilizes a database proxy that acts as oracle [67]. This oracle needs to be trained on normal SQL queries, for example, those resulting from an existing test suite. It will then raise alerts if it identifies SQL statements it has not learned before. As this can lead to many false positives, the results must be manually reviewed.

2.6.4 Semantic Difference

The last category of bug oracles described by Manès et al. is semantic difference, also known as differential testing. This technique leverages the fact that if two similar but not identical programs produce different output on the same input, it might indicate a bug [67]. For example, one could use differential fuzz testing on two versions of a program implemented in different programming languages. Both programs are executed on the same fuzz input and if the output does not match, this may point to a fault in one of the programs [67].

Although Manès et al. have not considered REST-based fuzzers (see Section 3.3.2) in their study, tools comparing the PUT's response against a previously defined specification (e.g., an OpenAPI specification) also fall under this category. Utilizing this information, it is possible to detect complex vulnerabilities such as broken access controls (see Section 2.3.1). For example, if the specification defines that a valid access token must be sent along the request to receive a successful response but a fuzzer receives a successful response despite not sending an access token, the fuzzer can detect this discrepancy. Note that, while the semantic difference bug oracle is the most powerful one, a correct reference program or specification representing the ground truth must already exist, which is often impracticable, or not feasible at all. Table 2.2 shows the bug oracles explained above and the vulnerabilities presented in Section 2.3.1 which each oracle is able to detect.

Vulnerability	Mem+Type Safety	Undef. Beh.	Input Val.	Sem. Diff.
Broken Access Control	○	○	●	●
SQLi	○	○	●	●
XSS	○	○	●	●
Buffer Overflow	●	○	○	●
Use After Free	●	○	○	●
Integer Overflow	○	●	○	●

Table 2.2: Detectable Vulnerabilities per Bug Oracle

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

From Fuzzing to Continuous Fuzzing

In this chapter, the term *Continuous Fuzzing* and the benefits it brings are discussed. Several studies on fuzzing solutions used in real-world settings or in CI environments are presented to identify the requirements and challenges of continuous fuzzing. Then, several fuzzing tools are introduced and their theoretical functioning will be explained to understand how they work. Lastly, existing continuous fuzzing solutions and their architectures are explored.

3.1 The Idea of Continuous Fuzzing

There is clear evidence that short feedback cycles lead to greater software quality and developer productivity [31] and Miller et al. [68] conclude that „it is now well understood by the software community that reliability is the foundation of security and that fuzz testing is a powerful first means of exploration on the path to finding software vulnerabilities". In 2018, Li et al. [3] even called fuzzing „the most effective and efficient vulnerability discovery solution currently".

The idea behind continuous fuzzing is to combine the benefits of tight feedback cycles of CI and the vulnerability finding ability of fuzz testing by integrating a fuzzer into the CI/CD pipeline. The feedback loop is then ideally near real-time: If a programmer tries to push changes containing a bug to the repository, the CI/CD pipeline starts the build, test and fuzzing process, which in turn notifies the developer of the bug and breaks the build. Thus, due to the reduced scope, the bug can be found and triaged before it becomes problematic.

In recent academia, many papers have been published introducing new fuzzing techniques and tools (e.g., RESTler by Atlidakis et al. [71], EvoMaster by Arcuri et al. [72], RestCT

TU WIEN Bibliothek Your knowledge hub

by Wu et al. [37]), promising better and faster results. However, not many studies have been done on using fuzzers in practice and even less so on continuous fuzzing. There are even different interpretations of the term *Continuous Fuzzing* in academic literature. Klooster et al. [10] understand it as a very quick fuzzing campaign integrated into a CI/CD pipeline, which is also the definition further used in this work. Others, for example, Rindell et al. [5], used the term to describe a fuzzing process which runs continuously, i.e., permanently or all the time. As shown below, both interpretations are valid and not necessarily mutually exclusive.

3.2 Requirements and Trade-Offs

Requirements for continuous fuzzing are manifold. An ideal fuzzer to be integrated into a CI/CD pipeline should be easy to set up and configure. In addition, it should be able to find all bugs present in the software, including complex vulnerabilities like SQLi, XSS and broken access controls (see Section 2.3.1) deeply inside the code and it should report the test cases that led to these faults clearly and concisely. And all of that should happen in under 10 minutes to keep the build process fast and give developers rapid feedback.

However, due to the nature of real fuzzers, some trade-offs must be made, especially with regard to fuzzing campaign duration and initiation.

3.2.1 Fuzzing Campaign Duration

One of the main issues when incorporating a fuzzer into a CI/CD pipeline is fuzzing campaign duration. While a fuzzing campaign should be long enough to enable the fuzzer to find bugs deep in the program, it should also be quick enough to have short feedback cycles which enable developers to fix a bug before it becomes problematic.

A recent study by Klooster et al. [10] examined the effectiveness and scalability of fuzzing techniques in CI/CD pipelines. They point out that the recommended fuzzing campaign duration in academia is at least 24 hours, while a reasonable build time of a program (including testing) is just 10 minutes [10]. Böhme et al. [73] show that the bug finding ability of fuzzers decreases exponentially with time, therefore, finding a reasonable number of bugs in a reasonable timeframe should be possible [10]. To verify this, Klooster et al. conducted a benchmark test and showed that a fuzzing campaign duration of 15 minutes is a solid balance between the fuzzer's bug finding ability and the developer's needs for short build times. Additionally, they also recommend running a longer (e.g., 8 hour) fuzzing campaign every once in a while. This approach is also consistent with Fowler's best practices for CI where a staged build pipeline is suggested.

However, Klooster et al. only studied this for three common fuzzers, namely American Fuzzy Lop (AFL)++, libFuzzer and Honggfuzz. As the PUT used for the case study in this thesis (see Chapter 4) is in the domain of web applications for which specialized fuzzers exist, these results might not be directly comparable to the use case in this work. Fortunately, Kim et al. [6] compared 10 different fuzzers specializing in REST

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

API testing. Across all fuzzers, code coverage and the number of response 500 errors do not change drastically between 10 minutes, 1 hour and 24 hour fuzzing campaigns. This suggests that REST-based fuzzers also have benefits when only applied for 10 or 15 minutes. Though, Kim et al. also note that the overall coverage achieved was not great, with EvoMasterWB's ~53% line and method coverage and only ~37% branch coverage leading the benchmark.

3.2.2 Fuzzing Campaign Initiation

Related to the fuzzing campaign duration problem is that of the fuzzing process initiation. When and how often the fuzzing campaign is executed during development is a key question for continuous fuzzing. It can be executed on a single commit, upon a pull-request, or it can be not linked to the git development workflow at all, for example, when only fuzzing certain releases.

Because starting a fuzzing campaign after every single commit (of which there can be more than 20 per day) might not scale, Klooster et al. [10] looked at strategies to minimize the use of resources. Since Zhu and Böhme [74] conducted an empirical study of fuzzer-generated bug reports and found that „four in every five bugs have been introduced by recent code changes", Klooster et al. [10] focused on optimising bugs that are regressions, i.e., bugs that are related to a feature which worked before but broke after recent code changes. They found that by calculating checksums on the fuzz targets and only starting the fuzzing campaign for those with a different checksum than in the previous commit, over 50% of fuzzing campaigns could be skipped [10]. This directly translates into saved computational resources for fuzzing [10]. However, this optimization is only possible if there is no versioning information like compilation timestamp or git revision present in the fuzz targets, as otherwise the checksum of the targets changes with each commit, leading to zero skippable fuzz targets.

Klooster et al. [10] also apply two additional optimization strategies regarding the fuzzer's corpus which lets fuzzing campaigns build on the progress of earlier fuzzing campaigns. The corpus is a set of interesting inputs, where interesting means that it led to newly discovered code. With corpus sharing, this set is shared between fuzzing campaigns by adding all interesting inputs to the corpus during fuzzing [10]. The next fuzzing session then uses this corpus as a seed. As this set grows exponentially, it must be minimized to stay effective [10]. This is done by filtering out inputs that do not reach unique parts of a fuzz target [10].

A further strategy to maximize code coverage is ensemble fuzzing. As a fuzzing session can not only exist of a single process that fuzzes a single target but also of multiple processes fuzzing the same target using a shared corpus, it is possible to reduce the time it takes to reach certain parts of the code [10]. An interesting idea here is to not use the same fuzzer for each process, but different ones. Thereby, each fuzzer with its specialized behaviour can contribute to the fuzzing campaign by sharing their newly found interesting inputs with the other fuzzers [10].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3 Readily Available Fuzzing Tools

In recent years, fuzz testing has become increasingly popular, not only in academia but also in practice. To shed light on the plethora of different fuzzing techniques and tools that emerged from this, Li et al. [3] and Beaman et al. [65] conducted large surveys on existing fuzzing solutions. They identified smart, domain-specific fuzzing algorithms as a great potential for improving code coverage and, thus, bug finding capabilities of fuzzers.

One such domain where smart fuzzing tools can be applied is in web APIs, more specifically, REST APIs. In the following, tools that do not specialize in any specific domain are called traditional fuzzers, whereas fuzzers specifically developed for testing REST endpoints will be called REST-based fuzzers. For each category, tools featured in surveys and empirical comparisons are collected and those relevant for the case study conducted in Chapter 4 are described in detail. To further limit the scope of the tools, the focus for traditional fuzzers lies on tools that are able to fuzz programs running in the Java Virtual Machine (JVM) since the PUT presented in Chapter 4 is implemented in Java.

3.3.1 Traditional Fuzzers

Hazimeh et al. [75] tackled the problem of lacking metrics and benchmarks for a fair evaluation of fuzzing tools. Since fuzzers directly using program crashes as performance metric leads to an unfair comparison due to different fuzzers employing different deduplication strategies, Hazimeh et al. created Magma, a ground-truth fuzzing benchmark that „introduces real bugs into real software" [75]. However, the set of targets only consists of C programs and, therefore, fuzzers for programs written in languages for the JVM cannot be applied on them. Similar benchmarks focusing on traditional fuzzing tools (e.g., FuzzBench by Asprone et al. [76], FixReverter by Zhang et al. [77]) also suffer from this limitation and no empirical studies comparing JVM-based fuzzing tools could be found at the time of this writing.

Nonetheless, a thorough literature research revealed several fuzzing tools available for JVM-based languages, namely Jazzer [W15], Kelinci by Kersten et al. [78], Javafuzz [W16] and JQF by Padhye et al. [38]. As Jazzer is based on libFuzzer and Javafuzz and Kelinci are heavily based on AFL and AFL++, these tools are also briefly explained, although they can only be used on C programs. This results in 6 different fuzzing tools shown in Table 3.1 which are described on the following pages.

libFuzzer

libFuzzer [W17] is a fuzzing engine for the C programming language using a mutation-based test case generation method. It provides in-process, coverage-guided fuzzing and is, therefore, considered a grey-box fuzzer.

The coverage information is gathered by code instrumentation (see Section 2.1.2), hence, it must be linked and compiled with the C program to be fuzzed. When building the fuzz

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Fuzzer Name	Type	Paper/Doc.	Repo.
libFuzzer	Mutation-based grey-box	[W17]	[W18]
Jazzer	Interface to use libFuzzer with JVM	[W15]	[W19]
AFL/AFL++	Mutation-based grey-box	[W20][W21]	[W22]
Kelinci	Interface to use AFL with JVM	[78]	[W23]
Javafuzz	Heavily focused on AFL, grey-box	[W16]	[W24]
JQF	Property-based testing approach, grey-box	[38]	[W25]

Table 3.1: Comparison of Traditional Fuzzers

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

target, the sanitizers AdressSanitizer and UndefinedBehaviorSanitizer can be specified. MemorySanitizer can also be used, although its support is only experimental [W17].

To save coverage progress, libFuzzer relies on a corpus of sample inputs. When an input leads to the discovery of a new path, it is added back to the corpus. The project page recommends initially adding seeds to this corpus to utilize its full potential. However, libFuzzer can also be used without initial seeds. The tool provides a simple option to minimize the corpus if it gets too large while still preserving the full code coverage. This can be achieved by running the tool with the merge option [W17].

libFuzzer can also be used in ensemble fuzzing by running multiple processes of it in parallel. To further benefit from this, the corpus should be shared between the processes because then they all can benefit from the discovery of a new input leading to greater code coverage [W17].

Jazzer

Jazzer [W15] is heavily based on libFuzzer. It is also an in-process, coverage-guided fuzzer but for JVM-based languages instead of the C programming language. In fact, Jazzer uses libFuzzer under the hood via its Jazzer driver, a native library running the fuzz target with libFuzzer linked in [W15].

A Jazzer agent runs in the same JVM as the fuzz target and instruments its bytecode at runtime. The coverage information obtained by the Jazzer agent is then fed back to libFuzzer which in essence looks to libFuzzer as if it were directly fuzzing a native binary [W15].

The core of its coverage information gathering strategy are control flow edges between basic blocks (see Section 2.1.2). This works by adding a unique ID (along with other bytecode instructions needed) to the beginning of every basic block. Then, upon reaching a block, the shifted block ID from the previous block is XORed with the current one's ID to create a new ID for the control flow edge between the two basic blocks. Shifting the previous block ID is necessary because otherwise, a program containing a simple loop composed of only a single basic block would lead to XORing the block's ID with itself [W21]. Thus, if a program contained more than one such tight loop, their edges

would have the same ID. The second reason is that it would not be possible to differentiate between an edge from A to B and an edge from B to A as the result of the XOR operation would be the same [W21].

In addition to edges, bytecode-level and higher-level method-based (like String.equal) compares, switch statements, integer divisions and constant array indices are also instrumented in order to forward interesting values to libFuzzer. libFuzzer stores those values in a table of recent compares and uses these for further mutations, allowing it to pass these checks and reach further into the target's code branches [W15].

Jazzer instrumenting methods such as String.equal or String.startsWith to gather information about the data passed to these methods also opens up another possibility. In particular, Jazzer makes these hooks available to the fuzz target, enabling it to add custom sanitizers or stub-out methods where the fuzzer can get stuck easily (such as checksum verifications) [W15]. The feature can be used by using the @MethodHook annotation. The instrumented code can be placed before, after, or instead of the original method [W15].

Jazzer provides several example fuzz targets, one of them implementing a custom sanitizer that checks for path traversal attacks (see Section 2.3.1). A truncated version of this is given in Listing 3.1. Checking for this with Jazzer works by creating a method `fileConstructorHook` that hooks into Java's internal java.io.File class, more specifically its constructor. The HookType is set to `BEFORE`, meaning that the added code is executed before the original code. Inside the `fileConstructorHook` method, the path which comes from user input (or in this case from the fuzzer) is normalized and checked if it starts with a previously defined path. If it does not start with the previously defined path, `Jazzer.reportFindingFromHook` is called which makes Jazzer report the finding [W15].

```
1  static final String publicFilesRootPath = "/app/upload/";
2   @MethodHook(type=HookType.BEFORE, targetClassName="java.io.File",
       targetMethod ="<init>", [...])
3   static void fileConstructorHook([...], Object[] args, int hookId) {
4    String path = (String) args[0];
5    Path normalizedPath;
6    normalizedPath = Paths.get(path).normalize();
7    if (!normalizedPath.startsWith(publicFilesRootPath)) {
8      Jazzer.reportFindingFromHook(new FuzzerSecurityIssueHigh([...]);
9    }
10 }
```

Listing 3.1: Jazzer Fuzz Target [W26]

Fuzz targets that do not hook into existing methods are created by simply implementing a method "fuzzerTestOneInput(byte[] input)" from which the PUT should be called with arguments derived from the input byte array. If the PUT requires two or more independent inputs or if it must be converted to other valid classes, the input byte array can be further processed using the `FuzzedDataProvider`. This class provides methods

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

to consume booleans, integers, strings, etc. from the fuzzer input which can then be used to call the PUT [W15].

Another convenient feature of Jazzer is the autofuzz functionality. By setting the command line argument to a method (including its classpath) it will automatically infer suitable inputs. This eliminates the need to manually write fuzz harnesses/targets [W15].

AFL and AFL++

AFL is a fuzzer created by Michal Zalewski [W20] that aims to be practical. The coverage-guided fuzzer can fuzz programs written in C, C++, or Objective C efficiently by utilizing compile-time instrumentation and genetic algorithms. It has found many bugs in popular browsers such as Mozilla Firefox, Internet Explorer and Apple Safari as well as widely-used packages like curl, libpng, nginx and many more [W20]. Fioraldi et al. [79] named it „one of the most widely used and most successful coverage-guided fuzzers of all time“.

Zalewski published a simple text file explaining the internal mechanisms of AFL in detail [W21]. Coverage information is obtained very similar to that in libFuzzer, i.e., control flow edges between basic blocks are used which are obtained by XORing the shifted previous' block ID with the ID of the current block.

In terms of corpus minimization, AFL uses a fast algorithm that periodically re-evaluates the current input queue [W21]. It selects smaller subsets of test cases that still contain every edge discovered so far, whereby it favours test cases that cover the same or more edges with less latency and smaller size by applying a score to each test case. Test cases that are not favoured are still kept in the queue and are used with very low probability. With this method, AFL is able to work on a subset of the corpus that is 5–10x smaller than the starting data set [W21]. In addition, they provide the tool "afl-cmin" which minimizes the corpus in a more sophisticated but much slower way. Using this tool, the corpus is actually minimized, i.e., redundant inputs are permanently discarded.

Another performance improvement (usually between 1.5x–2x) is achieved by AFL via its fork server [W21]. This ensures that execve, linking and libc initialization only happen once and subsequent executions are cloned from this stopped process image. The fork server also comes with a deferred mode, in which users can manually specify initialization code areas in the target that the fuzzer should skip. Lastly, persistent mode enables a single forked process to execute multiple inputs, thereby greatly reducing the overhead needed for forking [W21].

Development of AFL was discontinued in 2017 but there exists a fork named AFL++ that promises more speed and better mutations and instrumentation [79]. A core feature of AFL++ is its Custom Mutator API with which the fuzzer can be extended at different stages, creating the possibility for researchers to evaluate their proposed improvements without much implementation effort.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kelinci

Kersten et al. [78] created an interface named Kelinci which enables running AFL on Java programs. It instruments a target Java application in much the same way AFL does with C programs and additionally adds a TCP server that is used to communicate with the C interface.

AFL expects the instrumented program to run a fork server and connect to the shared memory provided. Thus, Kersten et al. wrote an interface.c binary implementing a fork server identical to the one in AFL's instrumented programs. Upon AFL initiating the creation of a fork, the interface.c process forks itself and sends the input to the Java server which starts the instrumented target application in a new thread and monitors it. The result (OK, Error, or Timeout) is sent back to the interface.c process along with the shared memory bitmap. The interface.c process then writes the received memory bitmap to its shared memory provided by AFL and, depending on the received status code, exits normally, crashes itself, or keeps looping until AFL's timeout is reached [78]. Consequently, to AFL this looks just like it is fuzzing a regular fuzz target instrumented by its compiler [78]

They evaluated their tool by running it on a JPEG parser from the well-known Apache Commons Imaging library. Although Kelinci is approximately 500x slower than AFL, running on the native DJPEG utility due to the overhead resulting from the TCP communication as well as Java being generally slower than C, they were able to find a bug after approximately 20 minutes [78].

Javafuzz

Javafuzz [W16] was first developed by Fuzzit which was acquired by GitLab in 2020 [W27]. It is a coverage-guided fuzzer for Java that tries to mimic the options and output style of libFuzzer [W16].

Seed inputs are supported via a corpus directory. Unlike Jazzer, it does not feature a possibility to autofuzz methods, hence, fuzz harnesses must be written for each method to be fuzzed. It also provides no convenience wrapper for transforming the raw fuzzer input bytes into Java primitive types, so the byte array must be cast manually if needed. An example fuzz harness is shown in Listing 3.2 [W16].

The `AbstractFuzzTarget` is an interface containing the abstract method `void fuzz (byte[] data)` that is called by the fuzzer. In the example, the provided data byte array is then used to create a `ByteArrayInputStream` from which a `BufferedImage` is created via `ImageIO.read`. As the `ImageIO.read` throws an `IOException` on invalid input, the exception is caught and ignored. If another, unexpected exception occurs, the fuzzer will catch it and add the test case to the list of bugs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
1 public class FuzzExample extends AbstractFuzzTarget {
2   public void fuzz(byte[] data) {
3     try {
4       BufferedImage image = ImageIO.read(new ByteArrayInputStream(
            data));
5     } catch (IOException e) {
6       // ignore as we expect this exception
7     }
8   }
9 }
```

Listing 3.2: Javafuzz Fuzz Target [W16]

The documentation does not give any details about its coverage guidance other than using the JaCoCo Java Code Coverage library. Its main fuzzing loop generates an input, calls the PUT and retrieves the new line coverage from the JaCoCo agent. If the new coverage is greater than the previous total coverage, the input that led to this newly covered line is added to the corpus. Although the repository states that „Javafuzz tries to mimic some of the arguments and output style from libFuzzer“, no option is given in the documentation to minimize the corpus and an inspection of the code also did not reveal any method for this feature.

JQF

Padhye, Lemieux and Sen [38] created a platform for performing coverage-guided fuzz testing in Java, called JQF. It is designed for practitioners as well as researchers. That means that in addition to an easy-to-use fuzzing approach based on property-based testing, they provide a Guidance interface with which researchers can implement custom fuzzing algorithms.

The fuzzer is built on top of junit-quickcheck, thus, fuzz harnesses are written similarly to property-based unit tests (see Section 2.1.2). An example fuzz harness is shown in Listing 3.3. With the `@Fuzz` annotation, JQF automatically generates arguments for the method to test, guided by its Guidance interface. The precondition is given in line 5 via JUnit's Assume API which states that the `map` must contain the key `key`. After that, the actual method to test is called (here, the constructor of the `PatriciaTrie` class) and the properties that must hold are asserted. In this case, it is asserted that the new resulting instance of `PatriciaTrie` also contains the key `key`.

The Guidance interface defines four methods that can be implemented to create a custom fuzzing behaviour. The method `getInput()`, which returns the input for the next test case, builds the core of this interface. In junit-quickcheck, inputs of type T are randomly sampled via a `Generator<T>` to create many different test cases depending on a source of randomness. In its default case, this is backed by a pseudo-random stream of bytes. JQF overrides this behaviour to use `getInput()` of the Guidance interface instead, creating deterministic generators. It has five Guidance implementations built-in [38]:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```java
@RunWith(JQF.class)
class TrieTest {
  @Fuzz /* Arguments are generated randomly by JQF */
  public void testMap2Trie(String key, Map<String, Integer> map){
    assumeTrue(map.containsKey(key));
    Trie trie = new PatriciaTrie(map); // Map2Trie
    assertTrue(trie.containsKey(key));
  }
}
```

Listing 3.3: JQF Fuzz Target [38]

- No Guidance: This simply does not use coverage information and returns random inputs similar to using vanilla junit-quickcheck.
- Zest Guidance: The Zest algorithm was specifically designed for coverage-guided property testing by the authors of JQF. It maintains a set of parameter sequences of dynamic size which are randomly generated in its first iteration. Then, in further iterations the parameter sequences are randomly mutated, leading to the Generator to create new structures. For example, a random mutation in the parameter sequence might lead to a Generator of type Map to generate a map with an additional entry [38]. The code coverage for both, valid and invalid test cases is tracked. If a mutated parameter sequence reaches new code (valid or invalid) or if a valid test case reaches code that has not been reached by another valid test case previously, the sequence is saved.
- AFL Guidance: This mode uses the AFL binary and implements the AFL communication protocol using a proxy program, similar to Kelinci (see Page 44). Test cases generated by AFL are read by the Guidance interface and returned in `getInput()` to be used in the next test case. The instrumented test target collects code coverage which is written back to AFL's shared memory region using the proxy. As AFL does not differentiate between test cases that failed because of the precondition not being met and cases that failed due to an assertion violation or exception, it is most effective with test methods that do not have `assume` statements.
- PerfFuzz Guidance: This algorithm is designed to find hot spots or performance bottlenecks in a program. It is based on the AFL code coverage algorithm but in addition to branch coverage, it also saves how often a branch is called with a specific input. An input is saved if it leads to new code coverage or if already discovered code is executed more often with the new input. Another configuration enables the detection of memory consumption issues by tracking the number of bytes allocated at allocation sites.
- Repro Guidance: Here, `getInput()` returns the contents of a file and exits the fuzzing loop after one iteration. This enables easy debugging or reproduction of saved test cases.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

3.3.2 REST-Based Fuzzers

Web APIs – as all other software components – need to be tested for logical bugs and security issues to ensure their quality. Since creating these test cases by hand can be very time-consuming and error-prone [6], the need for automatic test case generation arose. Similar to traditional fuzzers, there also exist many different fuzzers specializing in REST API testing, each with its unique approach, resulting in different strengths and weaknesses.

The main difference to traditional fuzzers is that their starting point is not a binary application or a method of it, but an endpoint of an API. Therefore, those tools need some information about the target. Instead of having the user write fuzz harnesses for each API endpoint or running the fuzzer against the whole web application without any information about its endpoints, REST-based fuzzers leverage the fact that most often machine-readable documentation of the API already exists or can be created quickly and automatically for different frameworks. One such documentation format is OpenAPI [W28], previously known as Swagger. Its details are explained in Section 2.2.2. Another common web API schema supported by modern REST-based fuzzers is GraphQL.

The traditional fuzzers presented in Section 3.3.1 are all grey- or white-box tools and, therefore, can only be applied to the programming language they were written for. Thus, comparing the performance of fuzzers for different programming languages is not possible since they cannot be tested on the same PUT. For REST-based fuzzers, in contrast, there exist some papers that empirically compare their performance and bug finding abilities. These are briefly summarized and a set of tools to be further analyzed in this work is derived.

An empirical study by Corradini et al. [80] compares different black-box test case generation tools for RESTful APIs to help developers decide which tool best fits their needs. They tested the tools RestTestGen, RESTler, black BOX tool for Robustness Testing (bBOXRT) and RESTest on 14 real-world REST services and analyzed the results in terms of success rate and test coverage. Since most REST-based fuzzing tools are black-box, code coverage metrics are not obtained by the tools. Therefore, different metrics to compare fuzzers must be used. As fuzzing tools do not have a common fault model, use different oracles to detect bugs and do not apply the same deduplication strategies, directly comparing claimed detected bugs by the tools might be unfair [80]. Hence, Corradini et al. developed a measurement framework that „measures the testing coverage with respect to the specification of the REST API rather than to its actual code“ [80]. It is based on Martin-Lopez et al.'s [81] test coverage framework which uses the API interface description within the OpenAPI specification to create ten coverage metrics. As some of the metrics from Martin-Lopez et al. are too abstract for an empirical comparison, Corradini et al. adapted them to make them usable in practice [80]. Additionally, they skipped two metrics because they were either not operative or too complicated to implement [80]. This left them with five input coverage metrics (e.g., the ratio of API paths tested to total amount documented in the OpenAPI specification) and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

three output coverage metrics (e.g., the ratio of correct/incorrect status codes received from the application per operation) [80]. To automatically extract these metrics for different tools, Corradini et al. created a Python tool that computes the metrics from request-response pairs of an HTTP traffic log [80]. In addition to the comparison of coverage, they analyzed the fuzzers in terms of robustness, i.e., how many of the 14 case studies the tool was able to successfully test. The most robust tool is RESTler (14 successfully tested case studies), followed by RestTestGen (11) and bBOXRT (8). RESTest was only able to successfully test 2 out of 14 case studies and failed on all others. This is caused by the tool's inability to create body parameters when no examples are given in the OpenAPI specification [80]. For test coverage comparison, Corradini et al. removed RESTest from the test due to its low robustness. They also limited the case studies to eight which all of the remaining tools could test successfully. In terms of test coverage, RestTestGen is the tool with the best results, as it produced the best coverage for five of the coverage metrics, while RESTler and bBOXRT each outperformed the other tools for only one metric.

Hatfield-Dodds and Dygalo [48] created the tool Schemathesis and evaluated it by comparing it to seven other web API fuzzers: RESTler, Cats, TnT-Fuzzer, Got-Swag, APIFuzzer, Fuzz-lightyear, Swagger-conform, Fuzzy Swagger and Swagger fuzzer. They ran tests on 16 real-world open source web services to analyze defect detection, runtime and consistency of reporting [48]. In terms of robustness, Schemathesis is a clear winner, as it was able to test all 16 targets without crashing, followed by RESTler and APIFuzzer which were able to test 11 targets. As they deem „the schema no less important than the implementation of the service", they also count mismatches between specified and actual behaviour, such as unexpected status codes, as defects [48]. This results in Schemathesis finding 40% to 350% more bugs per target than the respectively second-best fuzzer [48]. However, because other fuzzers are not as restrictive in regard to checking semantic properties of the specification, this comparison is not of much value. Therefore, they also compared the total unique HTTP 500 server errors reported by the tools for each target. In this case, Schemathesis was still the tool finding the most defects for each target, followed by APIFuzzer and RESTler. Since not all tools were able to test all targets successfully and four of the seven tools (excluding Schemathesis) even failed on more than half of the test targets, it is hard to compare their performance and draw a conclusion [48].

Kim et al. [6] analyzed the strengths, weaknesses and limitations of 10 different REST fuzzers. They ran EvoMaster, RESTler, RestTestGen, RESTest, Schemathesis, Dredd, Tcases, bBOXRT and APIFuzzer against 20 real-world open source RESTful web services to compare their code coverage achieved as well as unique error responses triggered. Web services were limited to those implemented in Java/Kotlin to be able to collect their line, branch and method coverage with the JaCoCo code-coverage library [6]. EvoMaster is the only tool that offers a white-box testing feature. To utilize this, Kim et al. created driver programs for code instrumentation and database communication analysis. They ran each tool on each target ten times for one hour and calculated the average over these

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

results to account for the tool's in-deterministic behaviour. Figure 3.1 depicts the results of the study.

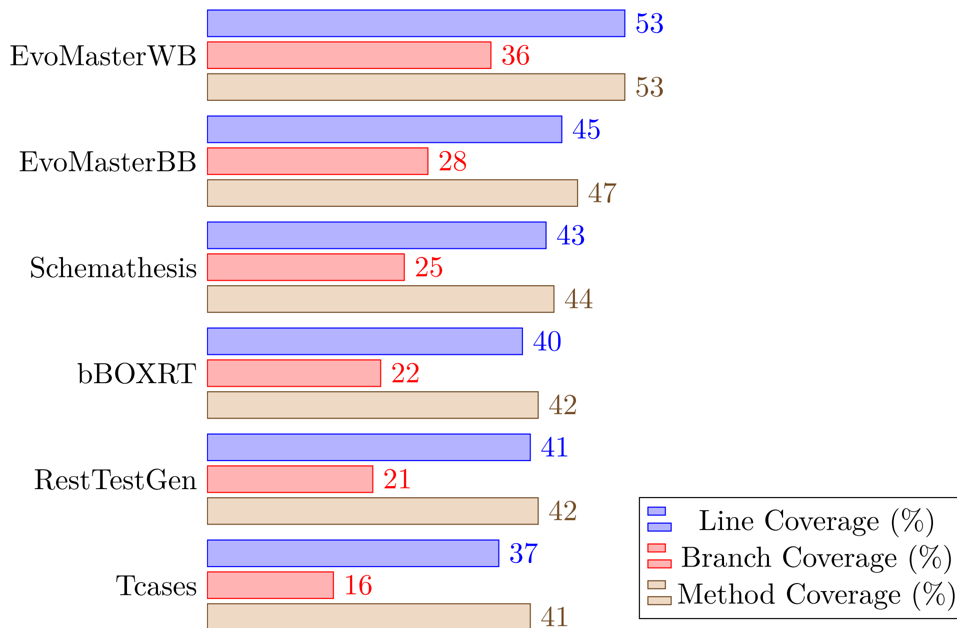


Figure 3.1: Code Coverage of REST-based Fuzzers in Empirical Study by Kim et al. [6]

EvoMaster in white-box mode (EvoMasterWB) achieved the best line, branch and method coverage with ~53%, ~36% and ~53% respectively. In black-box mode (EvoMasterBB), the tool still reached ~45%, ~28% and ~47% for line, branch and method coverage, followed by Schemathesis (~43%, ~25%, ~44%), bBOXRT (~40%, ~22%, ~42%), RestTestGen (~41%, ~21%, ~42%) and Tcases (~37%, ~16%, ~41%). Errors were measured as unique 500 errors (grouped by their stack traces), unique failure points (grouped by the top-most entries of the stack trace) and unique library failure points (same as unique failure points, but the failure point was a method from a third-party library used by the service). Again, EvoMaster in white-box mode performed best, finding 33.3 unique errors, 6.4 unique failure points and 3.2 unique library failure points. Tcases (18.5, 3.5, 2.1) ranked second, followed by Schemathesis (14.2, 2.8, 2), EvoMaster in black-box mode (16.4, 3.3, 1.8), RESTler (15.1, 2.1, 1.3) and bBOXRT (9.5, 2.1, 1.3) [6].

To gain insights into current limitations and actual effectiveness of state-of-the-art black-box REST fuzzers, Zhang and Arcuri [8] compared 7 fuzzing tools and analyzed the source code of 18 open source and one industrial RESTful API they were applied to. The tools included in their research are bBOXRT, EvoMaster, RESTest, RestCT, RESTler, RestTestGen and Schemathesis. Similar to Kim et al., they ran each tool on each API target 10 times and averaged the results to account for randomness in the tools [8]. As some of the tools exited before their one-hour time budget, they started each fuzzing campaign in a new thread, restarting the fuzzer if it exited earlier and terminating the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

fuzzing campaign after one hour. They obtained line coverage information via code instrumentation, using JaCoCo for their 13 targets running in the JVM and c8 for the 6 targets running on NodeJS. After an initial test run, only three of the seven tools were able to successfully test all 19 test targets. Since this was only due to wrong or unexpected schemas (e.g., OpenAPI v2 or v3, YAML or JSON, wrong host or server info in the schema), they developed a utility tool to automatically fix the schemas of their test targets [8]. The code coverages achieved in their black-box tool comparison are summed up in Figure 3.2.

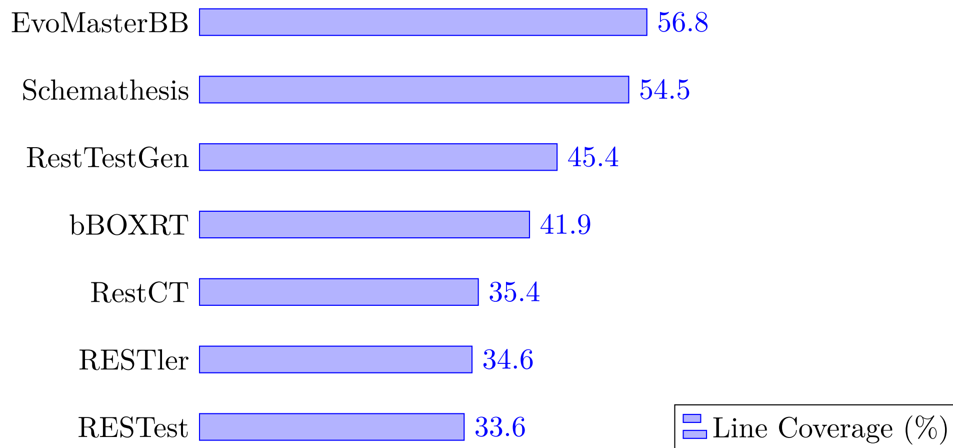


Figure 3.2: Line Coverage of REST-based Fuzzers in Comparison by Zhang and Arcuri [8]

EvoMaster achieved an average code coverage of 56.8%, being the best in 11 out of 19 PUTs), followed by Schemathesis (54.5%, best in 7), RestTestGen (45.4%) and bBOXRT (41.9%). The remaining tools (RestCT, RESTler and RESTest) only achieved between 33.6% and 35.4% [8]. Additionally, they compared black-box to white-box fuzzing tools, however, as EvoMaster is currently the only REST-based fuzzer that supports white-box fuzzing, they only compared EvoMaster's black-box mode to its white-box mode. On average, line coverage improved by 7.5% and faults detected by 9.3% in white-box mode [8]. However, Zhang and Arcuri also emphasized that in one case the tool performed worse than in black-box mode (line coverage from 35.3% down to 24.8%). They attributed this behaviour to the quality of the fitness function, as it got stuck in local optima, but also said they will address this problem in a future version of EvoMaster [8].

Table 3.2 shows a short overview of the different fuzzing tools included in the studies introduced above. Out of these 18 tools, 11 were discarded from further analysis in this work due to the following reasons:

1. foREST: Although Lin et al. [84] claimed in their paper from March 2022 that they „released it as open source on GitHub“ [84] and „the link will be released soon“ [84], they have not done so yet.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Fuzzer Name	Category	Test Generation Method	Latest Paper
EvoMaster WB	White-box	Evolutionary algorithm	[72]
EvoMaster BB	Black-box	Random Testing	[72]
Schemathesis	Black-box	Property-based	[48]
RESTler	Black-box	Dependency-based	[71]
RestTestGen	Black-box	Dependency-based	[82]
bBOXRT	Black-box	Robustness Testing	[51]
RESTest	Black-box	Model- & Constraint-based	[83]
RestCT	Black-box	Combinatorial Testing	[37]
foREST	Black-box	Dependency-based	[84]
Cats	Black-box	Random, Constraint-based	n/a
Got-Swag	Black-box	Random	n/a
APIFuzzer	Black-box	Random, Mutation-based	n/a
Tcases	Black-box	Model-based	n/a
Dredd	Black-box	Sample-value-based	n/a
Fuzz-lightyear	Black-box	Property-based	n/a
Swagger-conformance	Black-box	Property-based	n/a
Fuzzy Swagger	Black-box	Random	n/a
Swagger Fuzzer	Black-box	Property-based	n/a
TnT-Fuzzer	Black-box	Random	n/a

Table 3.2: Comparison of REST-based Fuzzers Based on Kim et al. [6]

2. Cats: No academic paper available, significantly outperformed by Schemathesis [6].
3. Got-Swag: No academic paper available, last updated in February 2018.
4. APIFuzzer: No academic paper available, significantly outperformed by other tools [8].
5. Tcases: No academic paper available, significantly outperformed by other tools [8].
6. Dredd: No academic paper available, significantly outperformed by other tools [8], last updated in December 2021.
7. Fuzz-lightyear: No academic paper available, last updated in October 2020.
8. Swagger-conformance: No academic paper available, last updated in June 2018.
9. Fuzzy Swagger: No academic paper available, last updated in December 2021.
10. Swagger Fuzzer: No academic paper available, fails on 15 out of 16 APIs in [48], last updated in October 2016.
11. TnT-Fuzzer: No academic paper available, fails on 15 out of 16 APIs in [48].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The remaining 7 tools are described in detail on the following pages.

EvoMaster

EvoMaster was initially created by Arcuri [34] to help researchers integrate a novel approach for automatic test suite generation called MIO algorithm (see Section 2.1.2) which he developed. Later it was introduced by Arcuri [85] as an automatic, white-box test case generation tool using evolutionary algorithms for JVM-based REST APIs, the architecture of which is discussed below. Note that this concerns only the white-box mode of EvoMaster, as its black-box mode was only added at a later time, though the main aspects of the following description still hold for the current version of EvoMaster. Furthermore, as EvoMaster does not utilize symbolic execution, in the classification presented in Section 2.5.2 it should actually be classified as a grey-box fuzzer. However, to avoid any confusion and as it is also called a white-box fuzzer by other studies [6], [8], the term white-box as suggested by the authors of EvoMaster will be used in the following.

The tool consists of two main components. The core process is responsible for functionalities like command-line and OpenAPI parsing, generation of test files and also includes the search algorithm, as explained by Arcuri [34]. The driver process starts, stops and resets the PUT and instruments its source code [34]. EvoMaster generates test suites optimized for code coverage and fault detection via HTTP 500 error responses and the generated tests use the libraries JUnit and RestAssured.

Core and driver processes communicate via JSON over HTTP, hence, adding support for a new target language only requires changes in the driver and not the core component [34]. To use EvoMaster in white-box mode, the EvoMaster client library must be imported and a driver class (specifying, e.g., where the OpenAPI schema can be found, how the PUT should be started and on what port it listens or which packages will be instrumented) must be implemented [34]. Because some PUTs can have side-effects that are persisted, e.g., in a database, there is also a method to reset the state of the PUT in which, for example, the database can be emptied and some previously defined test data can be inserted [34]. In addition, many RESTful APIs require some form of authentication. If this is the case, a method in the driver class must be implemented which returns valid credentials that are used in authentication headers or cookies. Since some PUTs cannot be started from a class, EvoMaster also provides the possibility to start it in a separate, external process from the driver process, in which case the driver class handles the technical details regarding starting and stopping of the process as well as collecting statistics from the spawned processes [34].

As it is very common for web applications to interact with databases, Arcuri and Galeotti [86] explored an approach to take the state of such databases into account, which they implemented in the form of an EvoMaster extension. With this extension, EvoMaster intercepts every single `SELECT` query and guides its search algorithm such that these SQL queries return non-empty responses [86]. Moreover, it automatically can

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

inject data directly into the database by utilizing observed SQL queries. For example, if a generated test case causes an SQL query with a `WHERE` clause that returns an empty set, the tool can inject data such that the same test case will return data, thus, possibly executing a different path in the test target. They tested their new approach on 5 different REST APIs, however, they were not able to achieve statistically significant improvements on 3 out of the 5 targets.

Schemathesis

Schemathesis was built by Hatfield-Dodds and Dygalo [48] and is, similar to JQF, based on property-based testing. It uses the Hypothesis library to create sophisticated test inputs using hybrid random generation and feedback-guided structured mutation [48]. The test functions and oracles it creates (both, for individual endpoints as well as sequences of requests to different endpoints) can be easily customized. In addition to its command-line interface, Schemathesis also provides a Python API. Hence, for PUTs that are also written in Python, communication can occur in-process instead of over the network, resulting in faster fuzzing and the option for coverage-guided fuzzing.

One of its core features is its differentiation between single-request tests and tests that make a sequence of requests to multiple endpoints. By using a state machine that tests the whole system instead of only single endpoints, it tends to report fewer bugs per run but is considerably faster. Hatfield-Dodds and Dygalo argue that this design makes Schemathesis better for „interactive use in a run-fix-rerun cycle, rather than long-running testing campaigns“ [48].

As the tool uses Hypothesis' data generation strategies, it must read the OpenAPI or GraphQL schemas and generate valid Hypothesis strategies from them. This is done via the hypothesis-jsonschema and hypothesis-graphql libraries. Since some schemas cannot easily be converted to Hypothesis strategies, especially when they contain intersecting constraints, Schemathesis applies some semantics-preserving transformations to the raw schema and canonicalizes the results before converting it to a Hypothesis strategy [48]. Non-recursive references to other schemas are inlined and overlapping subschemas are merged to create a schema of minimal form [48].

The tool was designed to be easily customizable, as such it features four main ways to alter the way it behaves [48]:

1. Hooks allow customization at different steps of the testing process. This allows the user to, e.g., change the API schema for certain endpoints to work around incompatibilities, adjust generated test data or filter undesired test cases.
2. Checks can be used to verify additional properties of responses received from the PUT. They can be used to implement custom test oracles.
3. Serializers transform the data before they are sent to the PUT. Default ones include application/json, multipart/form-data and text/plain but others can be implemented if needed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4. Format Strategies: Many OpenAPI specifications use custom formats to better describe string values (e.g., a value of type string with the format phone_number must be a valid phone number). As the OpenAPI specifications allow for arbitrary formats, Schemathesis allows the user to specify how data for a specific format should be generated.

RESTler

RESTler was written by Atlidakis, Godefroid and Polishchuk [71] at Microsoft Research. They introduced it in 2019 as the first automatic stateful REST API fuzzer. With stateful, Atlidakis et al. mean that the tool performs a lightweight static analysis of the OpenAPI schema and then tests sequences of requests to multiple endpoints rather than single endpoints [71]. Thus, it can explore states of the PUT that are only reachable by certain sequences. This is done by inferring dependencies declared in the OpenAPI schema as well as by analyzing dynamic feedback from observed responses of the PUT [71].

From the OpenAPI specification, RESTler generates a test-generation grammar encoded in executable Python code that will generate the HTTP request and process the expected response [71]. While doing this, it also infers dependencies. For example, if an ID parameter is necessary for retrieving a resource (via a GET request) and an ID is returned after creating a resource (via a POST request), RESTler takes note of this producer-consumer dependency and uses this information for test generation [71].

The main test generation algorithm computes a set of request sequences *seqSet* which is initially empty. A main loop, starting with $n = 1$ computes all valid request sequences of length n, where a valid request sequence is one in which every single response to a request returns a valid response code (i.e., in the 200 range) [71]. This is repeated until n reaches a user-defined *maxLength*, which denotes the maximum sequence length the tool should compute. Extending the *seqSet* in each iteration is done in two steps. First, the set of valid request sequences from the previous step (with length $n - 1$) is extended by appending to each sequence each request whose dependencies are satisfied. The dependencies of a request R are satisfied if all dynamic objects required in the request are produced by one of the responses from the requests preceding request R. Then, each newly-extended request sequence is rendered, meaning that the list of fuzzable types is computed and the values are substituted with a value taken from a user-configurable dictionary. This generates all possible combinations of values, e.g., if the request contains one fuzzable primitive of type integer and the defined values for integer types in the dictionary are 0, 1, -1 and 42, the algorithm will generate 4 requests corresponding to the values. These new request sequences are then executed, and if they receive a valid response code they are retained, otherwise, they are discarded and the error code is logged for analysis and debugging purposes [71].

As this described algorithm generates all request sequences of length $n + 1$ whose dependencies are satisfied and n is incremented by one in each loop, this can be seen

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

as a Breadth-First Search (BFS) algorithm (see Kozen [87]) [71]. In addition to that, RESTler features two further search algorithms [71]:

1. BFS-Fast: Here, instead of appending each request to every sequence in which its dependencies are met, it is only appended to at most one sequence. Thus, the resulting sequence set is smaller but every executable request type is still exercised at each iteration of the main loop, allowing it to go deeper faster.
2. RandomWalk: This algorithm extends a sequence set by randomly selecting a new request such that its dependencies are met. If it can no longer extend the current sequence, it starts from scratch.

Atlidakis et al. compared the three search strategies by running RESTler against the GitLab API. While the BFS algorithm achieved slightly better code coverage, RandomWalk was able to find the most bugs with 21, compared to 16 and 13 for BFS and BFS-Fast respectively.

RestTestGen

In 2020, Viglianisi, Dallago and Ceccato [82] presented RestTestGen, a tool to automatically generate test cases for RESTful APIs based on their OpenAPI definition. RestTestGen consists of three main components: The Operation Dependency Graph (ODG) generator, the Nominal Tester and the Error Tester [82].

The ODG generator is very similar to the first step of RESTler's algorithm where producer-consumer dependencies between endpoints are gathered from the OpenAPI specification. ODG is a directed graph $G = (N, V)$ where N is the set of operations according to the REST API specification and edges in V are dependencies between operations [82]. Two nodes have an edge $v = n_2 \rightarrow n_1$ when there is a data dependency between the two nodes, i.e., if there exists a common field in the response of n_1 and the input of n_2. Two fields are common if they are either of atomic type (i.e., string or numeric) and have the same name or if they are of a non-atomic type and are associated with the same schema. As developers are free to choose the names of their fields, some data dependencies might be missed using this matching approach if two common fields are not named exactly the same. For example, if an endpoint GET /users returns a list of users containing their IDs in a field named *id* and another endpoint is described as GET /user/$\{userID\}$, the dependency between the field *id* and the field $userID$ cannot be inferred. To mitigate this problem, they implemented a matching algorithm that is case-insensitive and prefixes fields named only *id* based on the name of the object they are a part of or based on the name of the operation. In addition, they apply a stemming algorithm so that some difference is tolerated, which makes the algorithm able to match common fields even if there is a small typo in of the fields [82].

The Nominal Testing module generates test cases adhering to the OpenAPI specification. It addresses three sub-problems:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Operation Testing Order: The order of operations is decided based on the generated ODG as well as the semantics of the REST standard. Because the standard defines that the POST operation should be used to create a resource while PATCH/PUT should be used to update it and DELETE to destroy it, RestTestGen always orders the operations adhering to these dependencies, i.e., POST before GET before PUT/PATCH before DELETE. Furthermore, the ODG is used to order operations such that those with the least input dependencies are executed first [82].

- Input Value Generation: This follows a probabilistic approach, reusing observed response data with high probability (80%). A response dictionary is used to save mappings between fields and their observed values. Similar to the matching algorithm in the ODG generation, a lookup in this dictionary allows for some tolerance. For the remaining 20%, a new value is generated based on its schema [82].

- Oracle: RestTestGen implements two oracles to assess generated test cases. The status code oracle simply uses the HTTP status code returned, where codes in the 200 range are treated as successful and codes in the 400 range as not successful. If the PUT returns a response code in the 500 range, this means the server encountered an error which RestTestGen will document. The second oracle performs response validation, meaning that it compares the observed response to that of the expected response stated in the API specification [82].

Error Testing is used to check if the PUT handles invalid input correctly. RestTestGen uses the test cases generated by the Nominal Tester and mutates them in order to create malformed and inconsistent input data. The three performed mutations are the removal of required fields, using wrong input types (i.e., a string where a numeric is expected) and violations of constraints (e.g., a numeric above its maximum allowed value as per the specification).

bBOXRT and EvoReFuzz

Laranjeiro, Agnelo and Bernardino [51] identified a lack of robustness testing tools in the domain of REST services compared to other domains such as communication software, embedded systems or SOAP services. Thus, they created bBOXRT, a black BOX tool for Robustness Testing that uses a service description document (like an OpenAPI specification) to generate a set of invalid inputs that are used to detect errors in either the specification or the implementation of the PUT. In contrast to existing tools, bBOXRT tries to be simpler, e.g., by following a simple rule-based approach instead of more complex white-box, state-based or model-based approaches [51]. The authors also emphasize their focus on testing the robustness of a PUT, although, they were still able to find some security vulnerabilities in the evaluation of their tool, highlighting the interconnectedness of robustness and security [51]. bBOXRT is implemented in Java and primarily consists of easily extensible components that work in 4 steps [51].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

First, it gathers basic information about the REST service from its interface description document. Currently it only supports the widely-used OpenAPI specification [51]. The PUT's endpoints, operations, input parameters, as well as expected responses are collected.

The second step deals with workload generation and execution. Here, valid requests according to the specification are generated by the workload generator and then sent to the PUT by the executor. If no pre-existing workload is found (e.g., by reading a set of stored requests), the tool will generate random requests to fill the workload. Valid requests (i.e., those that receive a response in the 200 range) are saved in a queue and will be used in the next step. As bBOXRT is not stateful, some requests might fail due to an unresolved data dependency, for example, when trying to delete a resource before creating it. In this case, the user can set a boolean configuration value to keep unsuccessful requests in memory and retry them after all the workload requests are finished. This step is executed until the workload is empty or if a user-specified time-out is reached [51].

In the third step, bBOXRT generates faulty requests by injecting a single error into the valid request data. A fault mapper stores the possible injection points gathered from the previous step as well as injected faults in order to avoid a fault being injected twice. The tool comes with different mutation rules for each of the datatype (All, Array, Boolean, Number, String) and format (e.g., Password or Date for strings, 32-bit integer for Numbers) combination that can be used in the OpenAPI specification. For example, for the Number datatype there are rules to add or subtract 1 unit, replace the value with -1, 0 or 1, or replace the value with the data type's maximum value + 1, i.e., try to trigger an overflow error in the service. The datatype String in conjunction with the format Binary has two rules to duplicate and swap random bits or bytes which can be used to test, e.g., image file uploads defined in the interface description document [51]. bBOXRT also comes with a set of over 800 malicious payloads for the datatype String that essentially try to cause SQL injections (see Section 2.3.1). This list could easily be extended with XSS payloads, however, as these most likely do not crash the application or cause other unexpected responses, a specialized bug oracle would be needed as well. For each request, each parameter is injected with the set of applicable faults for that data type, which is repeated a configurable amount of times. Hence, the user can analyze if the service responds to a specific fault consistently. In addition, for faults of stochastic nature, e.g., the removal of a random element in an array, this leads to a greater diversity of invalid requests. This step terminates after a user-specified time limit is reached, or if all applicable faults are injected and all the requests are executed [51].

Finally, the results (composed of service responses and test metadata) are stored for further analysis. The authors point out that this step requires some manual analysis as the responses received from REST services are too diverse to automatically distinguish between robust and non-robust behaviour [51]. This is, however, the case with all fuzzers relying on the OpenAPI specification, „with exception of cases where a perfect service specification exists“ [51].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In July 2022, bBOXRT received an update which implements a genetic algorithm. It was thereby rebranded to EvoReFuzz and „takes advantage of the components that compose a genetic algorithm while using the best of the bBOXRT tool to generate and send valid and invalid requests in an informed way to the system under testing".

RESTest

In 2021, Martin-Lopez, Segura and Ruiz-Cortés [88] presented RESTest, an open source, black-box testing framework for RESTful web APIs. Unlike other test case generation tools, it does not only rely on fuzzing, but also incorporates a constraint-based testing technique, as well as adaptive random testing (see Section 2.1.2). Test case generation and execution can be separately executed (offline testing) or they can be interleaved (online testing) in order to guide the test case generators with the gathered feedback. RESTest follows the following workflow [88]:

1. Test Model Generation: In this step, the OpenAPI specification is parsed into a system model. A second model, the so-called test model contains all test-related configuration settings for the PUT. This can also be manually extended to include, e.g., details about authentication, further data dictionaries to be used, or custom data generators that can be used for special formats like email addresses or phone numbers in string datatypes.

2. Abstract Test Case Generation: Based on the system and test models, abstract test cases are generated using one or more testing techniques. Abstract in this sense means that they are platform-independent and, thus, can later be transformed into executable test cases for different programming languages and testing frameworks.

3. Test Case Generation: The abstract test cases from the previous step are translated into specific test cases for a given testing framework. By default, RESTest supports the testing frameworks REST Assured and Postman.

4. Test Case Execution: In this optional step, the generated test cases are executed, i.e., the requests are sent to the PUT and the test results are saved in a machine-readable format. Those are then reported back to the user in a dashboard, using the test reporting framework Allure by default.

5. Feedback Collection: The test case generators can use feedback from previously executed tests to improve their code coverage using search-based techniques.

RestCT

Wu et al. [37] created RestCT, another tool for finding bugs in REST APIs. It employs a CT approach (see Section 2.1.2) to systematically and fully automatically test „not only the interactions of a certain number of operations in RESTful APIs, but also the interactions of particular input parameters in every single operation" [37].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In contrast to existing approaches like RESTler and RestTestGen, which focus only on creating valid operation sequences, RestCT also creates test cases where the order of operations in a sequence, as well as certain combinations of parameters are taken into account [37]. This means that if there are three operations A, B and C, and C must come after A, RestCT not only tests the operation sequence $A \to C$, but also $A \to B \to C$ or $B \to A \to C$. Additionally, the tool tries to automatically infer constraints between input parameters of a single operation using natural language processing, which existing tools lack [37].

RestCT employs a CT approach as described in Section 2.1.2, however, as REST API testing not only has parameters but also operations that can be executed in different orders, the concept of covering arrays is extended. Given a set of n events (or operations) $E = \{o_1, o_2, o_3, \ldots, o_n\}$, a test case is a sequence of $k \leq n$ distinct events and a t-way sequence covering array is a set of event sequences, in which every t-way sequence is covered at least once [37]. Wu et al. introduced constraints into this concept, where a constraint is either a dependency between input parameters or between events. Then, given a set of constraints C, the test cases in the t-way (sequence) covering arrays must also satisfy all constraints in C. With this approach, it is possible to model constraints between operations (e.g., o_1 must come before o_2) as well as between input parameters (e.g., if p_1 is a specific value, then p_2 must not be empty).

The overall workflow of RestCT includes two steps. First, during operation sequence generation, the OpenAPI specification is parsed, which results in a set of available operations and, by applying its hierarchical relations with its implied semantics, a set of constraints. The tool then applies a greedy algorithm to generate a constrained sequence covering array with coverage strength t_s, where t_s is a user-defined setting (or 2 by default) [37].

In the second step (called input parameter value rendering), RestCT generates the HTTP requests for the previously created operation sequences. It uses an adaptive strategy to generate constrained coverage arrays representing the concrete inputs for an operation, using the specification and test execution responses to infer proper values for the parameters [37]. For setting a specific value of a parameter, four strategies that are applied in decreasing priority exist:

1. Dynamic: Tries to match previously received response data for use in a new request.
2. Specification: Uses values described in the OpenAPI specification.
3. Success: Similar to dynamic, but only response data of successful requests are considered.
4. Random: If none of the strategies above can be applied, three random values are generated.

Just like operations, input parameters can also have dependencies between each other. However, OpenAPI does not offer any possibility other than using natural language

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

in a field's description to model these dependencies. Hence, Wu et al. use the natural language processing library spaCy to apply a pattern-based approach to automatically infer these constraints [37].

3.4 Existing Continuous Fuzzing Solutions

In the past years, and especially since the success of Google's OSS-Fuzz project in finding thousands of bugs in popular open source projects, other major organizations followed by creating their own continuous fuzzing solutions. Table 3.3 shows a quick overview of existing frameworks which are described on the following pages.

Organization	Name	Availability	Fuzzer Types
Google	OSS-Fuzz	Selected projects	Traditional
Google	ClusterFuzz	Open source	Traditional
Google	ClusterFuzzLite	Open source	Traditional
Microsoft	OneFuzz	Open source	Traditional
GitLab	Coverage-guided fuzz testing	Commercial	Traditional
GitLab	Web API Fuzz Testing	Commercial	REST-based

Table 3.3: Comparison of Existing Continuous Fuzzing Solutions

3.4.1 Google OSS-Fuzz / ClusterFuzz / ClusterFuzzLite

In 2016, Google has launched a service called OSS-Fuzz: Continuous Fuzzing for Open Source Software [W1] with the goal of making large-scale continuous fuzzing available to important OSS projects for free. It has since found more than 35.000 bugs in more than 550 open source projects, as shown by Klooster et al. [10]. Available fuzzing engines are libFuzzer, AFL++, Honggfuzz and Jazzer and supported programming languages are C/C++, Rust, Go, Python and Java/JVM code. The main focus of OSS-Fuzz is on long-running fuzzing campaigns with scalable, distributed execution [W1] that are run 1 to 4 times a day [W29]. It does provide an option to integrate it into a CI environment, however, these are rather limited as the only platform supported is GitHub.

Moreover, as described by Serebryany [89], this continuous fuzzing as a service is only available to open source projects Google deems important by having „a large user base and/or being critical to Global IT infrastructure“. The scalable fuzzing infrastructure behind OSS-Fuzz, named ClusterFuzz, is also open source, therefore, enabling projects ineligible for OSS-Fuzz (e.g., closed source projects) to help setup their own infrastructure and integrate fuzzing into their project's development process [W30]. However, with libFuzzer, AFL++ and Honggfuzz being the only fuzzers supported, it neither provides an option to fuzz JVM code, nor to run REST-based fuzzers.

For smaller code bases where high scalability is not a priority, Google has also released a lightweight version of ClusterFuzz, named ClusterFuzzLite [W31], which can be integrated

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

in CI/CD pipelines. It supports the major languages C, C++, Java, Go, Python, Rust, Swift and can be integrated in the CI/CD systems GitHub Actions, GitLab, Google Cloud Build and Prow. Features include quick code change fuzzing on pull-requests as well as longer-running fuzz tests to find deeper bugs, modular functionality and coverage reports showing which code parts the fuzzers reached. However, it does not support any REST-based fuzzers.

3.4.2 Microsoft OneFuzz

OneFuzz is a self-hosted Fuzzing-As-A-Service platform created by Microsoft [W32]. At the core of the tool, tasks are used to perform single units of work. An example task might be running the fuzzer AFL on a specific target. Jobs are used to describe a set of tasks and a template is a pre-configured job that includes common configurations for a fuzz job [W32].

OneFuzz is heavily dependent on Microsoft's Azure Cloud environment, i.e., targets for fuzzing (and supporting tasks) are deployed into an Azure Virtual Machine Scale Set. The scale set is connected to different nodes (i.e., Virtual Machines (VMs)) on which the tasks are run. For storage, a set of Azure Blob Storage Containers is used to which fuzzing tasks can connect using different contexts [W32].

While the solution is „multi-platform by design“ [W32], supporting the Linux and Windows operating systems for VMs, the nodes still have tun run an Azure VM agent and are required to connect to the Azure cloud. Similar to OSS-Fuzz and ClusterFuzz, it is built for larger-scale fuzzing campaigns, using multiple VMs for fuzzing and storage purposes. This, of course, comes with a lot of configuration overhead. Moreover, it only supports GitHub Actions to integrate it into a CI/CD workflow and does not support REST-based fuzzers.

3.4.3 GitLab Fuzz Testing

GitLab features two different continuous fuzzing solutions, Coverage-guided fuzz testing [W33] and Web API Fuzz Testing [W34]. Both are designed to be implemented in GitLab's CI/CD workflow. However, the fuzz tests are only available commercially to GitLab Ultimate users and their code is closed source, which makes its analysis rather difficult. Still, their core features and characteristics are outlined below.

Coverage-guided Fuzz Testing

Coverage-guided fuzz testing in GitLab Ultimate's CI/CD workflow supports 8 different fuzzers, including the two JVM-based fuzzers JQF and Javafuzz, where the latter is the recommended one [W33]. It has two of the main features needed for an efficient continuous fuzzing solution as presented in Section 5.2:

1. Two stages: It is possible to create two jobs to run synchronous regression fuzzing campaigns that block the build process and provide rapid feedback, as well as a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

long-running, asynchronous fuzzing campaign that is able to find bugs deeper in the code [W33].

2. Corpus sharing: By using a project-wide corpus registry, it is possible to reuse the corpus in different jobs. This enables, e.g., two stages to share their corpora, making the process more efficient [W33].

The found vulnerabilities are integrated into the GitLab UI and can be viewed directly in the respective commit or merge request. A security dashboard features additional information about vulnerabilities found in all projects and pipelines [W33]. The possibility to perform ensemble fuzzing is not documented, however, it might be possible by defining two jobs that access the same corpus registry.

Web API Fuzz Testing

The Web API fuzz testing solution from GitLab Ultimate currently is the only continuous fuzzing solution that supports REST-based fuzzers. It utilizes one of four ways to infer the URLs it automatically tests: OpenAPI Specification, GraphQL Schema, HTTP Archive, or a Postman Collection. There are several ways to specify authentication tokens, including the possibility to periodically call a user-defined script that creates a new token and writes it to a file, which in turn gets picked up by the fuzzer to replace tokens with a short expiration time. Although the documentation does not explicitly describe a two-staged fuzzing environment like in coverage-guided fuzz testing, it should be possible to manually create these. The solution uses checks (e.g., for form body, JSON or XML fuzzing) which in turn use assertions to detect faults. The assertions utilize the PUT's response data and status codes, as well as its log data to detect unusual behaviour. Unfortunately, the documentation does not say if it uses any of the previously introduced fuzzers or if they created their own.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

Case Study: Proof of Concept Solution for Continuous Fuzzing

This chapter introduces an exemplary project and experiments with various fuzzers presented in the previous chapter to compare their suitability to be integrated in a continuous fuzzing process. To that end, two approaches are explored to generate the OpenAPI specification document required to fuzz the target.

4.1 Description of the Program Under Test

The selected PUT is a reference implementation of an Identity Provider (IDP) server created by gematik, which is freely available on GitHub [W35]. It is based on OpenID Connect [W36], a protocol built on top of OAuth 2.0 that allows clients to verify the identity of end-users and obtain basic profile information. The IDP server project extends this protocol and provides the following attributes that render it compelling and representative for the case study:

- Easily deployable: The project is built with Maven [W37] and the Docker Maven plugin [W38], which makes it trivial to build the project and create a Docker container.
- Relatively isolated: The project does not have too many dependencies that would result in external services to be flooded with requests during fuzzing campaigns.
- No previous CI: One of the core requirements to be able to continuously fuzz a program is its integration into a CI process. Given the absence of a publicly accessible CI for the project, an exploration into the methods by which such an integration can be established can also be studied.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- Scale and complexity: Although the API is relatively small with only 19 exposed endpoints (see Table 4.1), its operations and their parameters are diverse and complex enough to show the limitations of current fuzzers.
- By following the REST protocol and using Spring Boot [W39] it implements state of the art architectures and frameworks.

The project is composed of multiple Maven submodules, however, three of them (idp-sektoral, idp-fedmaster, idp-fachdienst) are not included in the published GitHub project. Since the module of the project exposing the endpoints is the idp-server, this is not of significant concern. However, due to the idp-testsuite depending on those submodules, the integration tests would have to be untangled from them in order to execute the whole testsuite. Although the execution of the whole testsuite would have given an interesting indication of achievable test coverage, this inquiry was not pursued.

The existing unit tests for the idp-server module (version 21.0.22) runs 208 tests without errors or failures and uses the JaCoCo Maven plugin to create a coverage report in binary format. This binary coverage report can be turned into an HTML format using the JaCoCo Command Line Interface (CLI). Figure 4.1 shows the resulting test coverage of the unit tests for the idp-server module.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
de.gematik.idp.server.services		81%		69%
de.gematik.idp.server.controllers		78%		66%
de.gematik.idp.server.exceptions.handler		86%		75%
de.gematik.idp.server.data		59%		n/a
de.gematik.idp.server.validation.accessToken		64%		75%
de.gematik.idp.server.exceptions.oauth2spec		19%		n/a
de.gematik.idp.server		92%		50%
de.gematik.idp.server.exceptions		82%		75%
de.gematik.idp.server.validation.parameterConstraints		76%		70%
de.gematik.idp.server.exceptions.authentication		0%		n/a
de.gematik.idp.server.validation.clientSystem		94%		83%
de.gematik.idp.server.devicevalidation		100%		n/a
Total	875 of 4,469	80%	38 of 128	70%

Figure 4.1: Code Coverage of idp-server Module Achieved by Unit Tests

With 80% line coverage and 70% branch coverage, the unit tests cover the majority of the project. The subsequent section delves into a comparison between these manually composed test cases and those that are automatically generated by fuzzing tools.

4.2 Choosing Suitable Fuzzers

In Chapter 3, various fuzzing tools and approaches were introduced. The first and most important decision in terms of fuzzing tools selection is whether to use a traditional or

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

REST-based fuzzer.

As Rindell et al. [5] pointed out, fuzzing solutions have a high perceived impact on the security of the software to be developed but are not adopted broadly due to the high effort needed to build and maintain a fuzzing environment. Moreover, Böhme et al. [55] found that usability stood out as the primary concern for industry participants surveyed. Hence, the solution to be implemented should be as trivial to set up, configure and maintain as possible. This tendentially favours black-box fuzzers. Simultaneously, the fuzzer applied should achieve high code coverage in a short time frame, which suggests that a white- or grey-box fuzzer should be used.

Traditional fuzzers need complex fuzz harnesses and even Jazzer's autofuzz feature would need to be called on each method to be fuzzed, e.g., on each endpoint, which contradicts the requirement of a trivial setup. Given that EvoMaster utilizes domain-specific knowledge by testing the PUT's REST interface, it supports white-box mode and has been recognized as the best-performing fuzzing tool in two empirical studies [6], [8], it is deemed to be an excellent match. Though, none of the studies comparing REST-based fuzzing tools presented in Section 3.3.2 considers factors like false-positive rate, ease of use or how the fuzzer's results can be displayed and further utilized. Only Hatfield-Dodds and Dygalo [48] study the tool's ability to de-duplicate reports that cover the same fault. Another downside of the existing studies is that they only included REST APIs with already existing OpenAPI specifications, whereas for this case study automatically created specifications are used. Hence, the other fuzzing tools presented in Section 3.3.2 were analyzed briefly as well, taking these factors into account.

To run the REST-based fuzzing tools on the PUT, an OpenAPI specification document is needed.

4.2.1 OpenAPI Specification Document Creation

As the fuzzers should be integrated into the development process efficiently and usable, manual creation is not an option. For the automated creation of an OpenAPI specification document, there are generally two approaches available. First, the specification can be created by statically or dynamically analyzing the source code. The second approach utilizes the existing test suite and captures the traffic to infer the endpoints that got hit during test execution. While the first method is typically simpler and includes parameter constraints, the second approach has the advantage of incorporating preexisting information from the tests, e.g., as example values in the OpenAPI specification. In addition, the traffic capture approach can be employed even if there is no tool available for a project's framework that automatically creates the specification document via source code analysis. To get a comparison of these approaches, both methods were employed, resulting in the generation of two OpenAPI specification documents to be further used in experiments with fuzzers.

As the given PUT uses the widely-used Spring Boot framework for which the springdoc-openapi Java library [W40] exists, the creation via the first approach is as easy as adding

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

a new Maven dependency and rebuilding the project. This library „works by examining an application at runtime to infer API semantics based on spring configurations, class structure and various annotations“ [W40] and exposes a new endpoint `/v3/api-docs` where the specification can be accessed in JSON and YAML format. To generate the OpenAPI specification document, a Maven profile was created. If activated, it adds the `springdoc-openapi-starter-webmvc-api` dependency (version 2.0.2) and generates the OpenAPI specification document via the `springdoc-openapi-maven-plugin`. If the profile is not activated, a property is set, which skips the generation of the OpenAPI specification document. This ensures that the endpoint returning the specification is only exposed when necessary and not in the final application. The library created an OpenAPI specification document for the PUT containing 15 paths and 19 operations. In addition, 8 schemas specifying the different properties used in each operation were created.

To generate an OpenAPI specification document using the second approach, a traffic dump was created using the linux tool `tcpdump` while the unit tests were executed and the resulting pcap file was filtered so that it only contained HTTP packets. As there is no direct way to create an OpenAPI specification document from a pcap file, it was first converted to a HAR file using the pcap2har-go [W41] tool. Then, an OpenAPI specification in JSON and YAML format was created using har2openapi [W42]. Because the unit tests spawn several server instances listening on different ports, the resulting specification also contains different server and port combinations, resulting in many paths being duplicated. To combat this, the tool provides a search and replace functionality that can be added via its config file. The regex `localhost:[0-9]*` was used to replace the host and port combinations with an empty string, so that only the path after the base URL remained in the specification (the base URL can be set in most fuzzing tools via an option parameter). This resulted in an OpenAPI specification document with 18 operations on 13 paths. Contrary to the first approach, the resulting specification defines examples. However, it is not able to create schemas defining the exact formats and datatypes of parameters the server expects.

Table 4.1 shows an overview of the paths and operations that are defined in the resulting OpenAPI specification documents. The traffic capture approach did not pick up the path `/auth/realms/idp/.well-known/openid-configuration` because it is not tested in the unit tests. Furthermore, without additional configuration, the tool does not recognize ID parameters. Thus, an erroneous path `/pairings/654321` was created instead of correctly specifying the query parameter. The unit tests define one negative test on the endpoint `/.well-known/openid-configuration` with the method POST which returns a 405 (Method Not Allowed) error. Having such non-existent endpoints included in the specification is another disadvantage of the second approach.

4.2.2 Fuzzer Experiments

With the creation of the OpenAPI specification documents for the PUT, the REST-based fuzzing tools can now be run against the test target. To that end, the PUT was started

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Path	Method	Code Analysis	Traffic Capt.
/token	POST	●	●
/sso_response	POST	●	●
/sign_response	GET	●	●
	POST	●	●
/pairings	GET	●	●
	POST	●	●
	DEL.	●	●
/extauth	GET	●	●
	POST	●	●
/alt_response	POST	●	●
/jwks	GET	●	●
/idpSig/jwk.json	GET	●	●
/idpEnc/jwk.json	GET	●	●
/directory/kk_apps	GET	●	●
/discoveryDocument	GET	●	●
/.well-known/openid-configuration	GET	●	●
	POST		●
/a/r/i/.well-known/openid-configuration*	GET	●	
/pairings/	DEL.	●	
/pairings/{key_identifier}	DEL.	●	
/pairings/654321	DEL.		●

* /auth/realms/idp/ shortened to /a/r/i

Table 4.1: Comparison of Code Analysis vs. Traffic Capture Approach for OpenAPI Specification Document Generation

using the JaCoCo library to retrieve code coverage and the log level of the log4j2 library was set to debug. This enables the retrieval of the number of requests each tool issued to the API. Then, each tool was run in black-box mode two times, once with the OpenAPI specification created by code analysis, and once with the specification created through the traffic capture method. Since the goal of this comparison is to show the potential of quick fuzzing campaigns directly integrated into the CI/CD pipeline, as well as comparing the tools' practical application and their output on the different OpenAPI specification creation approaches, a time budget of 10 minutes was chosen for each fuzzing run. After each test run, the code coverage as well as the number of requests were retrieved and the PUT was restarted.

As the goal of the fuzzing campaigns is to find bugs in a short amount of time, the number of faults found by each tool is used as primary metric to determine the tools' bug finding capability. Moreover, similar to Kim et al.'s approach [6], code coverage is used as further indication of the tool's performance. This brings the additional benefit of being able to compare the value to the previous studies on EvoMaster's performance (see

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Section 3.3.2), as well as the manually created unit tests of the project. Furthermore, as an additional metric, the total number of requests issued by each tool is taken into account.

EvoMaster Black-Box

The two test runs with EvoMaster v1.6.0 [W43] in black-box mode achieved nearly identical results. Figure 4.2 shows the code coverage report after running EvoMaster on the PUT with the OpenAPI specification created by springdoc. With 37% line coverage and 15% branch coverage in 208516 requests it performed significantly worse than the unit tests.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
de.gematik.idp.server.services		11%		0%
de.gematik.idp.server.controllers		51%		50%
de.gematik.idp.server.exceptions.handler		70%		50%
de.gematik.idp.server.validation.accessToken		39%		50%
de.gematik.idp.server.data		38%		n/a
de.gematik.idp.server.exceptions		52%		75%
de.gematik.idp.server.exceptions.oauth2spec		9%		n/a
de.gematik.idp.server.validation.parameterConstraints		60%		30%
de.gematik.idp.server		94%		50%
de.gematik.idp.server.validation.clientSystem		89%		66%
de.gematik.idp.server.exceptions.authentication		0%		n/a
de.gematik.idp.server.devicevalidation		100%		n/a
Total	2,787 of 4,469	37%	108 of 128	15%

Figure 4.2: Code Coverage of idp-server Module Achieved by EvoMaster with OpenAPI Specification Created by springdoc

The test run with the specification derived from traffic capture covered 36% of instructions and 16% of branches in 282778 requests. One of the differences is in the ExceptionHandler class, where the traffic capture approach covered 80% of lines and 75% of branches, compared to 70% and 50% in the code analysis approach. This is, e.g., due to the method `handleMethodNotSupported` that gets called when tested with the specification file derived from the unit tests, because the tests contain such a request.

In both test runs, EvoMaster was able to detect a bug in the `POST /extauth` endpoint where the server returns an HTTP 500 error. A snippet of the generated JUnit test is given in Listing 4.1 and a detailed analysis of the underlying fault is presented in Section 5.4.1.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
1  ValidatableResponse res_0 = given().accept("*/*")
2    .post(baseUrlOfSut + "/extauth?" +
3        "code=IvCour5gAcCptS&" +
4        "state=prefix_Csf_postfix&" +
5        "kk_app_redirect_uri=n4efjquU2iYTu9z3")
6    .then()
7    .statusCode(500)
8    .assertThat()
9    .contentType("application/json")
10   .body("'error'", containsString("server_error"))
11   .body("'gematik_code'", containsString("-1"))
12   .body("'gematik_error_text'", containsString("Invalid Request"));
```

Listing 4.1: JUnit Test Generated by EvoMaster

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

EvoMaster White-Box

To test EvoMaster's white-box mode, a driver that extends the `EmbeddedSutController` and implements the required functionality (see Section 3.3.2) was created. The complete `EvoMasterController.java` file is listed in Appendix 1. The tool was run in its default configuration with a maximum time limit of 10 minutes. For the run with the springdoc documentation it covered 37% of lines and 17% of branches in 178084 requests, nearly identical to the test runs in black-box mode. This led to the finding of one new fault in the `GET /extauth` endpoint where an internal server error is returned (see Section 5.4.1). Again, with the specification generated from traffic analysis it achieved similar results (36% line, 17% branch coverage) in 232132 requests and found the same bug as in black-box mode but missed the second fault discovered in the first test run.

The documentation suggests running the tool between 1 and 24 hours in white-box mode to get better results. Thus, the test run using the springdoc OpenAPI document was repeated with a maximum time limit of 1 hour. However, with 38% line and 19% branch coverage, the longer fuzzing campaign did not bring any significant coverage improvements and did not find any new faults.

In Kim et al.'s and Zhang and Arcuri's empirical comparisons on multiple test targets, EvoMaster was able to reach between ~53% and ~57% on average (see Section 3.3.2). The discrepancy can be explained by the complexity of the PUT's parameters. Many endpoints in the idp-server project require JSON Web Tokens (JWTs) that carry encrypted content. Without giving the fuzzer additional information on the semantics and encryption schemes, it is not able to create meaningful inputs that are able to pass the first layer of parameter validation (e.g., the decryption of an input value) in the PUT, not even in white-box mode.

```
Response status: 500
Response payload: `{"timestamp":"2023-02-28T11:17:19.834+00:00","
   status":500,"error":"Internal Server Error","path":"/extauth"}`

Run this cURL command to reproduce this failure:
    curl -X GET -H 'X-Schemathesis-TestCaseId: 591
       e7fcf07a9447fa4c6666dc4a6897b' 'http://localhost:8080/extauth?
       client_id=0&code_challenge
       =0000000000000000000000000000000000000000000&
       code_challenge_method=S256&kk_app_id=0&redirect_uri=%3A&
       response_type=code0&scope=0&state=0'
```

Figure 4.3: Schemathesis Report for Found Defects

Schemathesis

Schemathesis [W44] in version 3.18.5 does not provide an option to set a time limit, instead one can specify the `--hypothesis-max-examples` which sets the maximum number of generated values per method/path combination. To make the test runs comparable to the EvoMaster tests, the value was adjusted a few times until the tool exited after about 10 minutes.

The first test was run with the hypothesis max examples set to 2000 and exited after 583 seconds with 20709 requests, achieving 38% line coverage and 17% branch coverage. It caught the same two defects as EvoMaster in white-box mode and provides a clear and concise output specifying the request, response status and body, as well as a cURL command to reproduce the error. A snippet of this report is given in Figure 4.3.

In the second test run, the hypothesis max examples needed to be set to 29000 for the tool to exit after 596 seconds with 58094 requests, covering 39% of lines and 19% of branches. The tool throws an `InvalidSchema` error for the `GET /sign_response` endpoint, indicating that the OpenAPI specification derived from traffic capture has some errors in it. It did not find the two faults as with the first run, however, it reported a response timeout error in the `GET /extauth` endpoint. This is because it uses a valid URI for the `redirect_uri` parameter after which the server responds with a redirect to a URI defined in the server. Due to the first test run not having any examples in the specification, no valid URI is generated for the parameter, thus, the fault is not found.

RESTler

As described in Section 3.3.2, RESTler [W45] first compiles an OpenAPI specification to a grammar and then generates requests from this. However, RESTler version 9.1.1 was not able to compile the OpenAPI specification generated by springdoc due to a NotImplementedException that states that objects in query parameters are not supported yet. For the specification derived by traffic capture the compile step succeeded, however, the test task failed with a NullReferenceException in /restler-fuzzer-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

9.1.1/src/driver/SpecCoverage.fs:line 321. This was fixed in a commit only a few days after the release of version 9.1.1 [W46]. Thus, the experiment was repeated, using the latest commit (50944ac) of the main branch.

With this version, the test with the OpenAPI specification succeeded (the springdoc-generated with objects in query parameters still is not supported and failed at the compile task), although it only was able to generate successful requests for 4 out of 18 endpoints. A fuzzing campaign running for 10 minutes resulted in 33016 requests with a line coverage of 36% and a branch coverage of 23%. However, RESTler's results analyzer failed with the message „Unexpected response without prior request [...]“ which apparently is a known issue since May 2021 [W47]. According to the authors, this does not affect the bugs reported by the tool, which in this case are 0. Manual inspection of the logs generated by RESTler revealed no responses with a 500 status code, thus, confirming this result.

RestTestGen

RestTestGen v23.02 [W48] does not provide any option to limit the time budget of its testing campaign and terminating the process early will not generate a report correctly. Thus, to get more comparable results, the tool was run until it finished for the first test and for the second test the tool was run several times successively on the same test target.

In the first test run using the springdoc-generated OpenAPI specification, RestTestGen finished after 9 minutes and 36 seconds with 32% line coverage and 16% branch coverage in 3176 requests. The low coverage and the low amount of requests can be explained by the tool following the redirects returned from the `GET /sign_response` and `GET /extauth` endpoints, which result in a timeout. These requests are re-tried 10 times, blocking the progress of other endpoints and without editing the source code, this behaviour cannot be changed.

Although RestTestGen did find one fault returning an error 500, interpreting this result is not trivial. It lists the status codes covered by each endpoint, which reveals a 500 response in the `POST /extauth` endpoint, but the corresponding request is not included. Instead, the generated reports in JSON files containing all requests and their corresponding responses must be manually searched or parsed to find the request responsible for the server error.

For the second test run, the server URL had to be added in the specification as the tool provides no option to set it on execution. After 11 fuzzing campaigns (taking 9 minutes and 41 seconds in total) the tool issued 33573 requests, covering 29% of lines and 14% of branches. However, no faults were encountered.

EvoReFuzz

EvoReFuzz (version 1.0) [W49], the advancement of bBOXRT, needs a manually created java file which transforms the OpenAPI specification into its internal RestApiSpecification

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

class. After implementing this file for both OpenAPI specifications, the tool was able to fuzz the target program. However, it consistently crashed after a few seconds at request 3468 with a NullPointerException at `pt.uc.dei.rest_api_robustness_tester.media.NoOpFormatter.Serialize(NoOpFormatter.java:83)`. An inspection of the file revealed a comment in the stated method: „FIXME: this should definitely be implemented in better way - VERY BAD PRACTICE“. Therefore, EvoReFuzz was deemed as unfinished and further experiments were abandoned.

RESTest

To install RESTest, the project (commit ca054a5) was cloned and the guide on its project page to run RESTest as a JAR was followed [W50]. As this resulted in a build error in the javadoc creation step, the flag `-Dmaven.javadoc.skip=true` had to be set. Two .properties files were created that contain the necessary configuration parameters, e.g., the location of the OpenAPI specification document and the output directory. The test case generator was set to FT, i.e., fuzz testing as per the tool's documentation [W50].

Similar to RESTler, an exception was thrown when using the OpenAPI specification created by springdoc because „The parameter type object is not allowed in query or path“. With the second specification document the test case generation step succeeded, however, the test execution step failed with the message: `0 tests run in 0 seconds. Successful: -1, Failures: 1, Ignored: 0`. Manual inspection of the generated test files revealed no obvious error and RESTest also does not provide any option parameters for more verbose error logging. Consequently, the current instability of the tool renders it unsuitable for practical application.

RestCT

The last tool that was analyzed is RestCT (commit 4dbd785) [W51]. As it only supports OpenAPI in version 2, the two documents were converted using the api-spec-converter tool [W52]. Then, it was run using its default configuration and a time budget of 600s. Using the first specification document created by springdoc, RestCT crashed with the message `AttributeError: 'NoneType'object has no attribute 'items'` in the file `RestCT/src/restct.py`.

The second specification was accepted, however, the tool exited after only 3 minutes of testing. Hence, the coverage strength of sequence covering arrays for operation sequences and the coverage strength of covering arrays for all input parameters were increased from 2 to 3 and the coverage strength of covering arrays for essential input-parameters from 3 to 4 using the `SStrength`, `AStrength` and `EStrength` command line options. This resulted in a more comparable test run of 10 minutes, achieving 30% line coverage and 9% branch coverage in 5023 requests. It was not able to find any bugs.

The results of the fuzzer experiments are summarized in Table 4.2. For both OpenAPI specification document creation approaches, line and branch coverage are shown, as well as the number of faults found and the number of requests issued by each tool.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

	Springdoc				Traffic Capture			
Fuzzer	LC	BC	Faults	Rqs	LC	BC	Faults	Rqs
EvoMaster BB	37%	15%	1	208516	36%	16%	1	282778
EvoMaster WB	37%	17%	2	178084	36%	17%	1	232132
Schemathesis	38%	17%	2	20709	39%	19%	1	58094
Restler	-	-	-		36%	23%	0	33016
RestTestGen	32%	16%	1	3176	29%	14%	0	33573
EvoReFuzz	-	-	-		-	-	-	
RESTest	-	-	-		-	-	-	
RestCT	-	-	-		30%	9%	0	5023

BC = Branch Coverage, LC = Line Coverage, Rqs = Total Requests

Table 4.2: Fuzzer Results After 10 Minutes with Different OpenAPI Specification Generation Approaches

Although the traffic capture approach led to slightly better code coverage, the springdoc-generated OpenAPI specification led to the finding of more faults. In addition, the traffic capture and subsequent transformation into two different formats is rather cumbersome and not as easy to automate as the springdoc approach. Furthermore, in springdoc one can enhance the generated specification with example values by defining them via Java annotations in the endpoint definition.

In terms of tools it is noticeable that half of them were not able to parse the OpenAPI specification document created by springdoc or otherwise crashed consistently. Despite the specification derived from traffic capture being converted two times using different open source tools, more fuzzing tools were able to parse it. This is due to them only supporting primitive types as query and path parameters, whereas springdoc generates an OpenAPI specification document that contains query parameters of type object. From those that are able to parse the springdoc-generated document, EvoMaster and Schemathesis achieved the highest coverage and found the most faults. This is in accord with the results in [6] and [8].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Integration into GitLab CI/CD Pipeline

This chapter explores how a fuzzing process considering the previously defined requirements and trade-offs (see Section 3.2), as well as the insights gained from the experiments with fuzzers in the previous chapter can be integrated into the GitLab CI/CD pipeline. A final design for the continuous fuzzing solution is established, implemented and evaluated.

5.1 GitLab CI/CD Concepts

The general workflow of GitLab's CI was already described in Section 2.1.3. To get an overview of possible solutions, its technical details and core concepts are explained further.

The top-level component in GitLab CI/CD are pipelines, which are comprised of jobs and stages [W6]. Jobs specify what to do, whereas stages define when the jobs are run. For example, a job that builds the PUT can be run in the stage build and a stage test runs multiple jobs that run a linter, unit and integration tests. By default, if a stage has more than one job, they are run in parallel and if any of them fails, the pipeline halts and the next stage will not be executed [W6]. Jobs whose output is needed in later jobs can output an archive of files and directories, called artifacts. These can also be downloaded through the GitLab web interface. A typical pipeline as described in the GitLab documentation might consist of the stages build, test, and deploy [W6].

Besides the basic pipeline type, which runs all stages concurrently, there also exist more sophisticated ones [W6]. More specifically, there exist two pipeline types that run only on merge requests, one of them acting as if the changes to be merged have already been integrated into the target branch [W6].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To execute jobs, GitLab uses a runner, an open source binary application written in Go [W53] that automatically picks up jobs as they appear. In the Software as a Service (SaaS) version of GitLab, runners inside a Linux VM are available. However, in the free tier of GitLab, only 400 CI/CD minutes (i.e., minutes of execution time by a single job) per month are included [W54]. For a runner, different executors are available which determine the environment each job runs in.

To implement the PoC, a local self-managed GitLab instance was set up, using the docker-compose installation method [W55]. In addition, two runner instances using the Docker executor were configured so that they pick up the jobs created by the test project.

5.2 Continuous Fuzzing Design

As illustrated in Section 3.2, an efficient continuous fuzzing solution should have two stages. The first one is a quick fuzzing campaign that gives developers rapid feedback on every commit. The second one executes a longer, asynchronous fuzzing campaign, which should be run every so often. For this, GitLab's merge request pipeline can be utilized.

In each stage, an up-to-date OpenAPI specification document must be provided before the fuzzing campaigns start. Since many development processes do not follow a specification-first approach, this document must be generated automatically. This step can be included in one of the stages preceding the fuzzing stage, i.e., in the build or test stage.

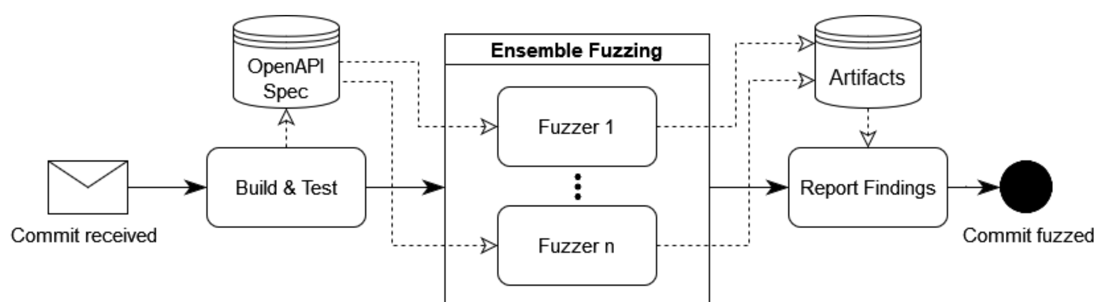


Figure 5.1: Continuous Fuzzing Design Based on Klooster et al. [10]

Figure 5.1 shows an illustration of the design based on a continuous fuzzing solution by Klooster et al. [10]. In Klooster et al.'s design, corpus sharing and minimization steps are applied that manage the seed corpus. As the presented REST-based fuzzers do not have seed corpora, these steps are swapped with a build and test step which outputs an up-to-date OpenAPI specification which will then be used by the fuzzers. The illustrated design works for both fuzz stages, the only difference is the time limit of the fuzzing campaigns and the fuzzers applied.

Since the OpenAPI documentation generation approach using springdoc found more faults in the comparison in the previous chapter and its integration is as simple as adding

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

a dependency, this method is preferred over the generation via traffic capture. As RestCT and Restler are not able to parse the OpenAPI specification document generated by springdoc, only the tools RestTestGen, Schemathesis and EvoMaster remain. Although RestTestGen was the best-performing tool in [80] and achieved not much less coverage than EvoMaster and Schemathesis in the comparison in the previous chapter, its applicability in context of a continuous fuzzing solution is limited due to it not having an option to set a time budget or otherwise limit its runtime. Furthermore, it only achieves a fraction of the requests compared to Schemathesis and EvoMaster. Thus, for the integration into a CI environment, EvoMaster and Schemathesis are deemed as best fit.

Note that, with stateful fuzzers, ensemble fuzzing only makes sense if they are run on a different target instance, because, otherwise, they would not work together, but against each other. For example, if fuzzer A tests a sequence where a resource is created, retrieved and then deleted, a parallel fuzzing process could interfere with this operation sequence and, e.g., delete the resource before fuzzer A can retrieve it. In addition, results might be irreproducible if a fault occurred due to unexpected interactions with the test target by other fuzzers.

Nonetheless, it is reasonable to utilize multiple fuzzers employing different test generation strategies concurrently, especially if a new instance of the test target can be easily started. Since this is the case with the given PUT, of which a new instance can be trivially started with a Docker command, two fuzzing jobs running in parallel were implemented. Note that this requires enough available runners for the project, otherwise the jobs are run consecutively.

5.3 Results

The proposed design for a continuous fuzzing solution as shown in Figure 5.1 was translated into two GitLab CI/CD pipelines, one that runs on every commit (basic pipeline) and one that runs on merge requests. The complete gitlab-ci.yml file is given in Appendix 2.

To avoid executing common installation steps (e.g., Python and Schemathesis) everytime a fuzzing job is run, a Docker image was created, which runs these steps once and provides simple utilities to run the fuzzers. This image comes with EvoMaster, Schemathesis, Maven and Java pre-installed and was pushed into the project's container registry to make it available to the Docker executors running the fuzz jobs.

5.3.1 Basic Pipeline

For the basic pipeline, the fuzzers EvoMaster and Schemathesis are run in parallel. Although EvoMaster generally performs better in white-box mode [6], [8] and a driver was already implemented in the previous Chapter, it is set to run in black-box mode for the basic pipeline and in white-box mode for the merge request pipeline (see Section 5.3.2). This has the reason that, as described by Zhang and Arcuri [8], the search algorithm

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

of EvoMaster's white-box mode can sometimes get stuck in local optima and in such cases its black-box mode performs better. Thus, to add more diversity to the continuous fuzzing solution, EvoMaster's black-box mode is utilized in the basic pipeline.

An overview of the stages and their dependencies is shown in Figure 5.2. The build stage takes roughly 2 minutes, the test stage 2 minutes and 36 seconds, and the report stage 22 seconds. Thus, to keep the total time of the pipeline to ~15 minutes, for EvoMaster a time budget of 600s was set and Schemathesis' max examples were set to 1500. As these values vary from project to project and depend on the hardware the runners run on, they can be easily changed by setting variables inside the pipeline definition (see Appendix 2).

Figure 5.2: Continuous Fuzzing Basic Pipeline Jobs and Their Dependencies

Build

In the build stage, the PUT is compiled using Maven. This builds a Docker image which is used in the later stages. To speed up the process, the unit and integration tests are skipped in this job, since they are executed in the next stage.

Test

The test stage executes Maven verify to run the unit and integration tests of the project. It sets two Maven system properties to skip the Docker build process and to generate the OpenAPI specification document. An artifact containing this document is created to make it available for the subsequent fuzzing jobs. The pipeline also defines a cache to make Maven dependencies available throughout the pipeline, thus, avoiding the test stage having to re-download dependencies that were already met in the build stage.

Fuzz

The fuzz stage consists of two jobs that each spawn an instance of the PUT and run a fuzzer on it. One runs Schemathesis and the other one runs EvoMaster in black-box mode. Both follow four simple steps:

- Start the PUT using Docker, joining it to the Docker network the Docker executor is running in.
- Wait for the PUT to be up and running.
- Run the fuzzing tool on the PUT.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Explicitly stop the PUT in an after_script section to ensure the Docker container is stopped, even if the fuzzer returns an error or crashes.

An artifact of the fuzzer's outputs assures that their results can be downloaded and viewed. Furthermore, the dependency on the previous' stage output (i.e., the OpenAPI specification) is configured using GitLab's needs keyword, assuring that the job waits for it. Since Schemathesis returns with a non-zero exit code if a fault is found, the command is extended with a || true statement. This guarantees that the pipeline always proceeds to the next stage in which the results are interpreted.

Report Findings

A last stage that depends on the fuzzing jobs aggregates the fuzzer's results and decides whether the pipeline fails or succeeds. To keep false positives to a minimum, only responses with a status code in the 500 range are considered as faults. Schemathesis already handles it this way in its default configuration. As EvoMaster also treats mismatches from the OpenAPI specification as bugs [W43], a script was created which parses EvoMaster's results. Thus, the stage only fails if one of the tools found a fault where the PUT encountered an internal server error.

To deactivate the pipeline failing if faults were detected, the allow_failure keyword can be set in the job to guarantee later stages to run despite found faults in the fuzzing stage. This can be helpful when initially integrating fuzzers into the pipeline to monitor their behaviour without affecting later (e.g., deploy) stages.

The report stage also writes an artifact with its output to a file to make it downloadable for further investigations.

5.3.2 Merge Request Pipeline

The merge request pipeline follows the same design as the basic pipeline but uses EvoMaster in white-box mode instead of two parallel fuzz jobs. Due to the workflow in pull-based development methods (see Section 2.1), the basic pipeline that runs EvoMaster and Schemathesis in a short fuzzing campaign is run on every commit in a branch. Then, upon issuing a merge request, EvoMaster is run in white-box mode for a 50 minutes long fuzzing campaign. This leaves enough time to the other jobs for the whole pipeline to finish in under one hour. Its jobs and stages are shown in Figure 5.3. The build and test jobs are the same as in the basic pipeline.

Figure 5.3: Continuous Fuzzing Merge Request Pipeline Jobs and Their Dependencies

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Fuzz

To run EvoMaster in white-box mode, a different approach than in the basic pipeline must be applied since the PUT must be started using the implemented EvoMaster controller instead of via Docker:

- First, the project is compiled using `mvn test-compile` to compile the project, including the implemented EvoMaster driver.
- Then, the classpath necessary to run the EvoMaster controller is built using `mvn dependency:build-classpath -Dmdep.outputFile=mvn_cp.txt`. This does not include the classes from the previously built target, thus, they are added to the classpath file in a separate command.
- The controller is started in the background using the previously built classpath and the job waits for it to start.
- Finally, EvoMaster is run in white-box mode to start the fuzzing process.

Just as in the basic pipeline, the EvoMaster output files are uploaded as an artifact to make it available to the report stage.

Report Findings

The report stage in merge request pipelines executes the same script as in the basic pipeline to generate the fuzzing report. In addition, it exposes this report on merge requests for the project maintainers approving the merge request to easily view the found faults, if there exist any. An image showing this feature where the fuzzing integration found a fault and failed the pipeline is given in Figure 5.4.

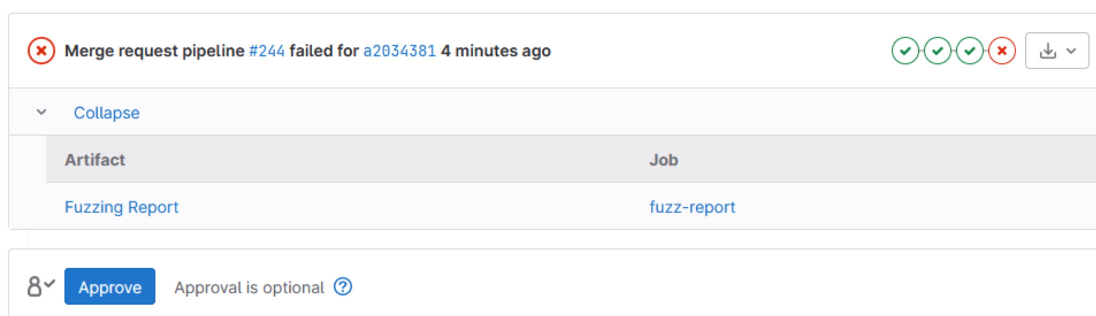


Figure 5.4: Exposed Artifact on Merge Request Approval Screen

5.4 Evaluation

To evaluate that the implemented solution works as intended, the workflow of a real development environment was simulated. To achieve this, the original ref-idp-server

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

project was cloned and reverted to version 4.0.0 as this is the first available version in the GitHub repository [W35]. Further goals of this evaluation are to find out which types of vulnerabilities the solution is able to detect and to measure its false positive rate.

The continuous fuzzing solution was re-added to the project in version 4.0.0 by copying the .gitlab-ci.yml file implemented in the previous section to the project's root directory and by copying the implemented EvoMaster controller. After this initial setup, a new branch was created and the changes from the next version (5.0.0) in the original project were applied by deleting all files and copying over the files of the new version. To not overwrite the changes in the idp-server/pom.xml needed for the OpenAPI generation, this file was manually merged. The changes were then committed in the new branch and a merge request was issued to merge the changes back into the main branch. These steps were repeated for each of the 37 available commits up to the latest available version (21.0.30).

For each commit and merge request the results of the CI/CD pipeline were examined. Every fault indicated by the fuzzers was analyzed in detail to be able to decide whether it is a false positive or not.

During the course of the evaluation, several issues with the historical state of the project were encountered:

- For the versions up until 21.0.0, a Java version 11 is needed to build the project, thus, the image used in the pipeline was set to maven:3.6-openjdk-11. For the remaining versions the image maven:3.8.7-openjdk-18 was used. In addition, the fuzz image in the project's container registry was also changed to use Java 11 instead of Java 17.
- Between versions 8.0.0 and 11.0.0 the project depended on an external GemLibPki library in versions that are not available on maven central. Thus, the 5 commits were skipped in the evaluation.
- Versions 11.1.0 to 13.0.1 have conflicting springfox dependencies which prevented the continuous fuzzing solution from generating the OpenAPI specification document. Therefore, the 4 commits were also skipped.
- Some versions contained tests that send an expired certificate to the server, which failed the pipeline in the test stage. These failing tests were removed from the project.
- As the versions up to 21.0.22 use springboot v2 instead of v3, the springdoc dependency which generates the OpenAPI specification had to be changed from `springdoc-openapi-starter-webmvc-ui` version 2.0.4 to `springdoc-openapi-webmvc-core` version 1.6.15. In addition, the database specification in the implemented EvoMasterController did not work with springboot v2 due to the major version change. Thus, it was removed for the evaluation in the versions up to 21.0.22.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- 3 commits between versions 17.0.0 and 18.1.0 and one commit for version 19.3.0 are missing the Dockerfile to build the image and were skipped.
- Versions 19.1.0 and 19.2.0 were skipped due to unsatisfiable dependencies.
- Schemathesis sometimes encountered errors, mostly in the POST /token and POST /sso_response in versions 20.0.9 up to 21.0.14. This is due to the server returning a redirect URL which Schemathesis cannot connect to because the URL is not reachable anymore. This has no effect on the other endpoints, however. For the evaluation, only the faults listed in Schemathesis' output in the failures section are considered as faults.

Following this approach, it was possible to successfully evaluate the continuous fuzzing solution on 22 out of 37 historical commits stretching over a development period of over 2 years.

5.4.1 Detected Faults

In the following, for each version in which faults were discovered, the faults are described briefly to determine whether a fault detected by the fuzzers is a false positive. Furthermore, each fault is evaluated whether it is a duplicate of an earlier detected fault.

V5.0.0

In version 5.0.0, the continuous fuzzing solution detected 3 faults.

The first fault (F_1) is in the `DELETE /pairing/{kvnr}/{id}` endpoint and concerns the id parameter. Since the endpoint definition does not limit this parameter to a specific format and then uses it in `Long.valueOf(id)`, an error is thrown if the provided value cannot be cast to the Long datatype.

F_2 is in `PUT /pairing/1` and occurs when no request body is sent. The server then sends an internal server error response indicating the missing request body but does not correctly return an error in the 400 range.

F_3 is in `GET /authorization` and occurs if the client sends an unsupported `code_challenge_method`, i.e., one that is not equal to S256. The server then fails to convert the given parameter to a `de.gematik.idp.field.CodeChallengeMethod` type because it is an enum with the values S256 and PLAIN.

F_4 occurred with a request to `PUT /pairing/3AekuGdSel0u` and returned the following error message: `could not execute statement; SQL [n/a]; constraint [null]; nested exception is org.hibernate.exception.ConstraintViolationException: could not execute statement.` The request body is mapped to a PairingDto which is then inserted into the database. However, only the kvnr value is checked for presence, thus, columns that have a `NOT NULL` constraint

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

on the database level will lead to the server throwing an `org.hibernate.exception.ConstraintViolationException` and returning a 500 error.

F_5 concerns the de-serialization of a date string in the `timestampPairing` parameter of the `PUT /pairing` endpoint. The value is mapped to a `ZonedDateTime` which throws an exception if the provided date string does not contain a zone identifier.

V5.1.0

Version 5.1.0 had only one error which is a duplicate of F_3.

V6.0.0

V6.0.0 introduced a new fault F_6 in the `GET /sign_response` endpoint. It is similar to F_3 in that it occurs upon sending an invalid `code_challenge_method` but a different error message is returned and it also occurs in a different endpoint. Hence, this is not counted as duplicate. Although the server correctly returns the error message „Invalid Request“, it fails to send an error code in the 400 range.

In addition, Schemathesis detected two response timeouts in the `POST /sign_response` (F_7) and `POST /sso_response` (F_8) endpoints. In these cases, the server returns with a redirect to a URL defined in the application settings upon encountering issues with the provided parameters. However, this URL is not reachable anymore. The server responding with a redirect to a non-existing location can indeed be interpreted as an issue, as such it is assessed as a true positive.

V7.0.0

In version 7.0.0, 5 faults were detected, two of them being the same response timeouts as F_7 and F_8, and one of them being the same as F_6.

A new fault F_9 is in the `DELETE /device_validation` endpoint. The server expects a `device_validation` parameter as query string upon which it tries to delete the given value by calling `deviceValidationRepository.deleteById(id)` without further input sanitization. Hence, if a string or a non-existing id is given, it throws an exception which is not processed further. Instead, the server responds with an error 500.

F_{10} is in the `PUT /device_validation` endpoint, which expects a deviceValidationDTO parameter as query parameter. Besides the parameter being required in the query string instead of as request body, if an arbitrary string is given, the server throws an error with the message „Failed to convert value of type 'java.lang.String' to required type 'de.gematik.idp.server.data.DeviceValidationDto'“. Thus, it fails to properly validate the input.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

V14.0.0 - V16.0.0

The three commits from V14.0.0 to V16.0.0 all encountered the same fault, which is a response timeout very similar to F_7 but in the `GET /extauth` endpoint instead of the `POST /extauth` endpoint. Thus, it is a new unique fault F_{11}.

V20.0.9

V20.0.9 introduced a fault that is still present in the latest version of the project. F_{12} was found in the `POST /extauth` endpoint, which is shown in Listing 5.1.

```
1  public void postAuthorizationRequestIncludingAuthorizationCode(
2  @RequestParam("code") @NotNull(message = "3005") final String
       authorizationCode,
3  @RequestParam(name = "state")
4  @NotEmpty(message = "2002")
5  @Pattern(regexp = ".+", message = "2006")
6  final String idpState,
7  @RequestParam(name = "kk_app_redirect_uri") @NotNull(message = "1004"
       ) final String kkAppUri,
8  final HttpServletResponse response) {
9  final FasttrackSession ftSession = fasttrackSessions.get(idpState);
10 log.info(
11 "idp-sektoral address: "
12 + getSekIdpLocation(ftSession.getUserAgentSekIdp())
13 + IdpConstants.TOKEN_ENDPOINT);
```

Listing 5.1: NullPointerException in POST /extauth endpoint in ref-idp-server

Because `fasttrackSessions.get(idpState)` in line 9 returns `null` for a state 0, `ftSession.getUserAgentSekIdp()` produces a `NullPointerException` for which no specialized ExceptionHandler is defined. Thus, the server returns an internal server error with the response code 500. As this fault has no other side effects, it is an issue of robustness and does not cause any implications to the project's security.

V21.0.0 - V21.0.14

The 7 fuzzing campaigns from version V21.0.0 to V21.0.14 all encountered the same three issues. One of them is a duplicate of F_{12} and the other two are the same response timeouts as F_6 and F_{11}.

V21.0.16 - V21.0.30

The project from version 21.0.16 onwards has two faults that were uncovered by the continuous fuzzing solution. One of them is a duplicate of F_{12}.

The other one, F_{13}, is in the `GET /extauth` endpoint and is triggered if the requested `redirect_uri` is set to ":" and if one of the constrained parameters (using `@Pattern`)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

does not follow the specified regular expression. Listing 5.2 shows the relevant parts of the endpoint in `IdpController.java`.

```
1 @RequestParam(name = "redirect_uri") @NotNull(message = "1004") final
      String redirectUri,
2 @RequestParam(name = "nonce", required = false)
3 @Pattern(regexp = "^[_\\-a-zA-Z0-9]{1,32}$", message = "2007")
4 final String nonce,
5 @RequestParam(name = "response_type")
6 @NotEmpty(message = "2004")
7 @Pattern(regexp = "code", message = "2005")
8 final String responseType,
9 @RequestParam(name = "code_challenge")
10 @NotEmpty(message = "2009")
11 @Pattern(regexp = SHA256_AS_BASE64_REGEX, message = "2010")
12 final String codeChallenge,
13 @RequestParam(name = "code_challenge_method") @Pattern(regexp = "S256
     ", message = "2008")
14 final String codeChallengeMethod,
15 @RequestParam(name = "scope") @CheckScope final String scope,
16 final HttpServletResponse response) {
17 idpAuthenticator.validateRedirectUri(clientId, redirectUri);
```

Listing 5.2: Request Parameters for GET /extauth endpoint in ref-idp-server

The underlying problem is that in line 1, the request parameter `redirect_uri` is not validated but is used in the method `buildForwardingError` which is called if one of the other parameters throws a `ConstraintViolationException`. If all the parameters are valid, `validateRedirectUri` is called and the `redirect_uri` is validated. Thus, this fault is only triggered in negative test cases.

Listing 5.3 shows the problematic use of the `redirect_uri`. In line 1, the parameter is extracted from the request parameters and in line 2 it is checked if the value is present. However, no other validation is performed and the value will then be used in line 9 to build a URI from it. If set to ":", the method call in line 10 will throw a `UriBuilderException` which results in the server returning an internal server error.

```
1 final String redirectUri = request.getParameter("redirect_uri");
2 if (redirectUri == null) {
3 final IdpErrorResponse body = getBody(exc);
4 if (!StringUtils.isEmpty(exc.getMessage())) {
5 body.setDetailMessage(exc.getMessage());
6 }
7 return new ResponseEntity<>(body, getHeader(), HttpStatus.BAD_REQUEST
     );
8 } else {
9 final UriBuilder uriBuilder = UriBuilder.fromPath(redirectUri)
10 [...]
11 final URI location = uriBuilder.build();
```

Listing 5.3: Unsafe Use of redirect_uri Parameter in ref-idp-server

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Further investigation of this fault led to the finding of an open redirect vulnerability (see Section 2.3.1). As the input parameter is not validated, an attacker can specify an arbitrary website to which the application will redirect to. A minimal working example using the cURL command is given in Listing 5.4. This vulnerability was responsibly disclosed to the developers of gematik who confirmed the issue.

```
curl "http://localhost:8080/extauth?client_id=0&redirect_uri=http://
    attacker.example"
```

Listing 5.4: Open Redirect Exploit for GET /extauth endpoint in ref-idp-server

5.4.2 Summary

In the 22 commits, the fuzzing integration detected 51 faults in total, 13 of which were unique across all commits. Note that for each fuzz report, faults detected by both, EvoMaster and Schemathesis, were counted as one. In addition, faults in the merge request pipeline that were already caught in the basic pipeline also do not increase the fault count per commit.

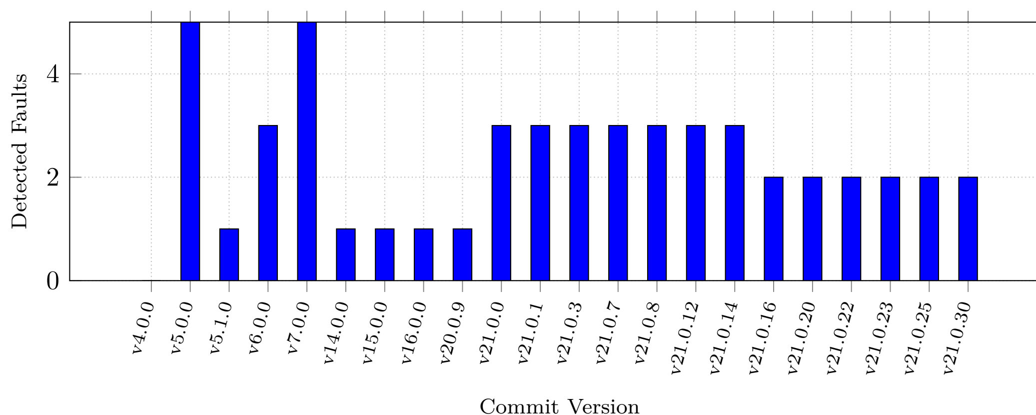


Figure 5.5: Total Detected Faults for Each Commit

Figure 5.5 shows the total number of faults found in each commit. The first 6 commits ranging from version 4.0.0 to 7.0.0 show a lot more variance in the detected faults than later commits.

The cumulative unique faults are shown in Figure 5.6. Out of the 13 unique faults, 11 were detected in the first 6 commits and 12 in the first 9 commits. In the versions from 20.0.9 to 21.0.30, only one new unique fault was uncovered.

Table 5.1 summarizes the results by showing the detected faults by the fuzzers for each commit. In the basic pipeline, EvoMaster discovered a total of 22 faults, 8 of which are unique faults across all commits. Schemathesis identified 37 issues, out of which 9 are unique across all 22 commits. There is an overlap of 9 faults (4 of them unique), which were detected by both fuzzers in the basic pipeline.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

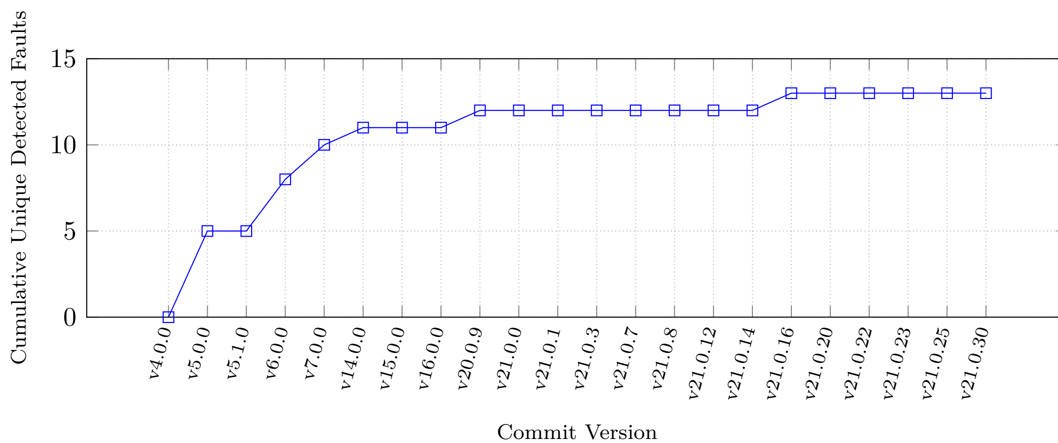


Figure 5.6: Cumulative Unique Detected Faults for Each Commit

The merge request pipeline employing EvoMaster's white-box mode uncovered a total of 28 faults, 10 of which are unique. Despite utilizing EvoMaster's white-box functionality and running 5 times longer, the faults found are identical to the results achieved by EvoMaster in the shorter basic pipeline in 16 commits. Only one of the faults detected in the merge request pipeline is an issue that was not already discovered in the shorter basic pipeline.

Lastly, the 13 unique faults the whole continuous fuzzing solution was able to detect are briefly summarized:

- In the 7 faults F_1, F_2, F_3, F_5, F_6, F_9 and F_{10}, the server does not properly validate the input, which results in it failing to cast the input string to a more specific datatype or class (e.g., Long). These are simple robustness issues where the actual behaviour does not lead to any unintended side-effects. However, the server still should properly validate the user input and return the correct status code (e.g., 400) if a user-supplied value does not meet the requirements.
- In fault F_4, the server does not properly validate the user input before using it to execute an SQL query. Albeit the returned error message leaks some information about the underlying technology stack, an SQLi is not possible because the correct methods of the framework are used which sanitize the user input. Again, such an error should be caught earlier and the correct status code (e.g., 404 not found) should be returned.
- The 3 faults F_7, F_8 and F_{11} are response timeouts where Schemathesis was not able to follow a redirect URL the server responded with[1].

[1]These URLs might be reachable from an internal test network only. In that case, the detected faults could be a false-positive.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

	Basic Pipeline (10min)		Merge Request Pipeline (50min)
Version	EvoMasterBB	Schemathesis	EvoMasterWB
4.0.0	-	-	-
5.0.0	F_1,F_2,F_3,F_4	F_1	F_1,F_2,F_3,F_4,F_5
5.1.0	F_3	-	F_3
6.0.0	F_6	F_7,F_8	F_6
7.0.0	F_6,F_9,F_{10}	F_7,F_8,F_9,F_{10}	F_6,F_9,F_{10}
14.0.0	-	F_{11}	-
15.0.0	-	F_{11}	-
16.0.0	-	F_{11}	-
20.0.9	F_{12}	F_{12}	F_{12}
21.0.0	-	F_6,F_{11}	F_{12}
21.0.1	F_{12}	F_6,F_{11}	F_{12}
21.0.3	F_{12}	F_6,F_{11}	F_{12}
21.0.7	F_{12}	F_6,F_{11}	F_{12}
21.0.8	F_{12}	F_6,F_{11}	F_{12}
21.0.12	F_{12}	F_6,F_{11}	F_{12}
21.0.14	F_{12}	F_6,F_{11}	F_{12}
21.0.16	F_{12}	F_{12},F_{13}	F_{12},F_{13}
21.0.20	F_{12}	F_{12},F_{13}	F_{12},F_{13}
21.0.22	F_{12}	F_{12},F_{13}	F_{12}
21.0.23	F_{12}	F_{12},F_{13}	F_{12},F_{13}
21.0.25	F_{12}	F_{12},F_{13}	F_{12}
21.0.30	F_{12}	F_{12},F_{13}	F_{12},F_{13}
Total	22	37	28

Table 5.1: Detected Faults by the Fuzzers for Each Commit

- Fault F_{12} produces a `NullPointerException` because the server fails to check if a resource depending on user input exists. As this does not have any side-effects, it is an issue of robustness.
- Finally, F_{13} is caused by the server trying to build a URL based on a parameter given by the user, which fails if the user-provided value specifies an invalid protocol, e.g., by starting with ":". This led to the finding of an open redirect vulnerability (see Section 2.3.1).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 6

Discussion and Limitations

This chapter provides a critical reflection on the implemented solution and discusses the research questions presented in Chapter 1, wherein the insights gained from the case study and its evaluation serve as a basis.

6.1 Discussion

Overall, the implemented PoC solution confirmed the core hypothesis that the integration of continuous fuzzing solutions into modern development processes helps with detecting faults in code, therefore, improving software security and robustness in an early stage of the development cycle. This is supported by the following answers to the research questions defined in Chapter 1.

> RQ1: How can state of the art fuzzers be integrated into a development process in a way that does not introduce too much configuration overhead?

To answer this question, first, a literature research was conducted to define the requirements of a possible solution. It was found that a usable integration means that for each commit, the pipeline should not run longer than 15 minutes in order to give developers rapid feedback. A second, asynchronous stage that should be run every so often can run longer fuzzing campaigns. For the integration into CI/CD pipelines, merge request pipelines were identified as a suitable initiation point for longer-running fuzz tests.

One of the main issues for existing fuzzing solutions not being adopted more widely is the large configuration overhead needed to set up the solution. Therefore, REST-based fuzzers were preferred over traditional ones because this eliminates the need for writing complex fuzz harnesses. Several experiments using readily available REST-based fuzzers were conducted. Similar to the results of Zhang and Arcuri [8], many tools proposed by

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

researchers lack robustness, as they are unable to parse OpenAPI specification documents or may crash during the fuzzing campaign. Nonetheless, two state of the art fuzzers were identified which are suitable to be integrated into a CI/CD pipeline.

The resulting PoC implementation utilizes the two fuzzing tools EvoMaster (in black-box mode) and Schemathesis to fuzz the target in quick, 10 minute fuzzing campaigns on every commit. Moreover, a driver was implemented to be able to perform white-box fuzzing using EvoMaster, which is employed in merge request pipelines to fuzz the target for 50 minutes on each merge request.

As the evaluation showed, the ensemble fuzzing approach proposed by Klooster et al. [10] proved to be useful. This is indicated by the fact that EvoMaster detected 8 unique faults and Schemathesis found 9 unique faults, whereas only 4 of the 12 total unique faults detected in the basic pipeline were discovered by both tools.

The results of the longer-running fuzzing campaign in the merge request pipeline, however, are rather discouraging as the second stage only uncovered one additional unique fault compared to the basic pipeline, despite utilizing a white-box fuzzing approach und running 5 times longer. Moreover, the code coverage experiments conducted in Section 4.2.2 only showed a minor increase in coverage for the longer-running fuzzing session. Thus, it is questionable whether this minimal advantage justifies the extra computational resources and the increased effort needed to set up this second stage. However, as the selected test target has relatively few endpoints and some rather complex parameters that, e.g., require encrypted JWTs, these results might vary greatly when applied to other test targets.

> RQ2: What benefits can short fuzzing campaigns in an early stage of the development life cycle provide?

As laid out in Section 2.4, finding bugs earlier in the development cycle saves time and money [4], [21] and one of the central test principles of the ISTQB [20] states that testing should start as early as possible in the SDLC. Moreover, fuzzing tools are largely successful in finding bugs [9], [10], therefore, one should strive for the earliest integration of fuzzers into the development process as possible. Similar to TDD, continuous fuzzing helps to bring software testing into the implementation phase instead of testing only in later testing and verification phases.

The prerequisites for a project using the developed continuous fuzzing solution are the ability to easily deploy the PUT, either locally or in a test environment. However, as outlined in Section 2.1.3, this is best practice and should be standard in modern development processes in any case. The second requirement for the fuzzers to be applicable is that an OpenAPI specification either exists beforehand, e.g., in specification-first development approaches, or that one can be easily generated automatically. For many frameworks, including Spring Boot, which is used by the project in the presented case study, tools and libraries for the automatic generation exist. In addition, a second

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

approach that captures the traffic of unit tests and transforms these requests into an OpenAPI specification was demonstrated.

Once these requirements are met, the steps to set up a continuous fuzzing solution using REST-based fuzzers are manageable. In the presented case study, this led to the finding of 51 total faults, 13 of them being unique across 22 commits. As shown in Section 5.4.2, nearly all the unique faults were uncovered in the first half of all commits and earlier versions of the project suffered from more issues than later ones. This also suggests that an integration of an efficient fuzzing process should occur as soon in the development life cycle as possible to uncover issues early on.

Moreover, as described in the previous answer, the second stage using a white-box approach and a 50 minute long fuzzing campaign did not lead to significantly better results than the quick basic pipeline. This highlights the value of short fuzzing campaigns during the implementation phase, emphasizing that software quality can benefit not only from the industry standard of 24-hour fuzzing campaigns (see Section 3.2.1) in the verification phase.

Although most of the faults detected were simple robustness issues that had no implications to the project's security, one of the issues led to the finding of an open redirect vulnerability, thus, further demonstrating the benefits a continuous fuzzing solution can provide.

RQ3: How can this process be automated in a CI/CD pipeline?

Provided that the PUT can be deployed effortlessly, an OpenAPI specification exists or can be automatically generated, and a driver exists (if white-box fuzzing is employed), integrating these components into a CI/CD pipeline is not too challenging.

To implement this, the test stage was configured to generate an up-to-date OpenAPI specification which is made available to the fuzz stage via GitLab's artifact feature. Furthermore, two additional pipeline stages were introduced: fuzz and report. If multiple runners are available, parallel execution of multiple fuzzing jobs using different fuzzers can further improve the fuzzing solution, as the relatively small overlap of detected faults between EvoMaster and Schemathesis for each commit (see Table 5.1) shows. The results of the fuzz jobs are again stored as artifacts, which the subsequent report stage utilizes to generate a fuzz report and to decide whether to fail the pipeline or not.

Additionally, several optimization strategies can be employed to further speed up the pipeline to spend more time fuzzing. In the case study, a caching strategy that avoids the re-download of dependencies that have been already met in previous jobs was employed. Furthermore, a Docker image that installs common packages needed in multiple jobs was created and pushed to the project's container registry, which further sped up the pipeline.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

RQ4: How can the fuzzing tool results be evaluated and incorporated into the development process?

To evaluate the fuzzing tool reports, a script was created that combines the fuzzer's outputs and fails the pipeline if a fault was found. Job artifacts are utilized to make the reports available through GitLab's UI. Furthermore, in merge requests the fuzzing report is exposed in the approval screen, thus, project maintainers can easily view the results of the fuzzing integration before approving a merge request.

Since the solution has been configured in a relatively conservative manner, limited to detecting errors solely within the 500 range, false positive rate is kept to a minimum. This is evidenced by the fact that not a single false positive was encountered during the course of the evaluation.

6.2 Limitations

The implemented solution has several limitations. First, the fuzzing results heavily depend on the quality of the OpenAPI specification document, as well as on projects abiding to HTTP and REST semantics. Especially the correct use of response codes is required, otherwise many faults could be missed. For example, if a project catches all internal server errors and then returns them as errors in the 400 range, this will not be detected by the PoC in its current configuration.

An additional problem is posed by outdated OpenAPI specifications. If the specification is not automatically created, missing or wrong endpoints and parameters are not known to the fuzzers and, thus, cannot be tested.

In terms of code coverage, fuzzing can hardly achieve the same level as carefully written unit and integration tests and fuzzers are not able to automatically pick up the semantics of certain parameters. In the provided case study, fuzzers were not able to reach deep into the code, instead, most requests did not make it past the input parameter validation. It is, therefore, not an alternative for sophisticated unit tests. Nevertheless, it proved to be an additional measure to improve software robustness and security.

Towards generalizing the statements made in this thesis it must be emphasized that the conducted case study has intrinsic limitations, as it only covers one concrete example project. Although the provided solution was developed with the goal to make it easily integrable with other projects, especially the steps in the build and test stage, as well as the white-box fuzzing approach, may not be representative for other projects using different programming languages or frameworks.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 7

Conclusion and Further Work

This final chapter concludes the main points of the thesis and offers prospects on further improvements of the implemented continuous fuzzing solution, as well as on additional research opportunities.

7.1 Conclusion

In this thesis, a case study was conducted to explore the integration of fuzzing into a continuous development environment using an example project in the domain of REST APIs. Through a thorough literature research, the requirements for a continuous fuzzing solution were identified, and a final design was developed.

Readily available fuzzing tools were assessed based on their code coverage and bug finding capabilities on the selected test target. Given that the usability of a continuous fuzzing solution was determined as a key requirement, particular emphasis was placed on ensuring that the tools remained user-friendly when integrated into a CI/CD pipeline. After selecting suitable fuzzers, a PoC implementing the proposed design was created.

The implemented solution utilizes the two black-box fuzzing tools EvoMaster and Schemathesis to fuzz each commit for 10 minutes to provide rapid feedback to developers. A longer fuzzing campaign of 50 minutes that employs EvoMaster's white-box mode runs on every merge request to fuzz the target before faults are merged back into the main branch. By making the resulting fuzzing report available in GitLab's merge request screen, any uncovered faults are brought to the attention of the project maintainers.

Several optimizations to speed up the stages in the pipeline were identified. To that end, a cache was set up that prevents the download of common dependencies across the pipeline stages. In addition, to minimize time spent setting up a job for the fuzzing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

process to start, a Docker image was created that encapsulates these common installation steps.

By uncovering 51 faults (13 of which were unique) in 22 historical commits during a simulated development workflow in an example project, the implemented PoC continuous fuzzing solution proved that bringing fuzzing in early into the development cycle has major benefits. Through a conservative configuration that only detects server errors that have a status code in the 500 range, it was possible to achieve a false positive rate of 0%, thus, the implemented solution is minimally invasive to the development process. However, the white-box fuzzer applied in the merge request pipeline was only able to detect an additional fault in one out of the 22 commits. The question arises whether the implementation effort of the driver required for white-box fuzzing, and the additional computing resources used by the longer-running fuzzing campaigns, are justified.

Although most of the detected faults are relatively simple issues where, e.g., input values could not be mapped to a more specific datatype, one of the uncovered faults is not only an issue of robustness but even led to the uncovering of an open redirect vulnerability, which was responsibly disclosed to the developers of gematik who confirmed the issue. This demonstrates the link between software robustness and security once more.

7.2 Further Work

With the groundwork being laid by the implementation of a technical PoC which was evaluated in a simulated development environment, the integration of the solution in a real software project's development workflow should now be studied.

In this further research, the human factor should be taken into account by evaluating the developer's experiences and satisfaction with setting up the solution into their CI environment. In addition, it should be evaluated how developers are able to reproduce the issues given the fuzzing reports and whether raised issues are false positives or not. Since the white-box fuzzing approach did only lead to slight improvements compared to the results of the shorter fuzzing campaigns in the selected project, this stage should also be evaluated in other projects to assess whether the extra computational resources and the increased effort needed to set up this second stage are justified.

In terms of possible improvements to the continuous fuzzing solution, several directions are possible. One could, for example, add an extra stage that utilizes traditional fuzzers that target single, specific methods that are detrimental to a project's business value. Furthermore, the implementation currently does not make use of the fuzzer's abilities to add authentication headers to the requests as this would require some manual test data generation before starting the fuzzing process, resulting in three endpoints not being fuzzed past the authentication steps. In a real-world integration of the provided solution, such optimizations tailored to a single project could also be considered to further improve the fuzzer's code coverage, although, with the drawback of increasing the configuration overhead needed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The de-duplication of faults is another issue one could solve. Currently, the fuzzing report in the basic pipeline can contain the same fault twice if both, EvoMaster and Schemathesis detect the same fault. Although the different output of the two tools could provide further information to reproduce a fault, the manual de-duplication of faults is valuable time one could save by automating this process.

Moreover, the fuzzing reports currently only provide the found issues via a simple text file. Other approaches to feed the found issues back into the development process could be explored, for example, by automatically creating GitLab issues of faults that are detected. Given the fact that no false positive was encountered during the course of the evaluation, such an integration into the development process would further improve the visibility of detected faults. To not flood developers with faults detected in various branches by the basic pipeline, this approach could also be limited to the merge request pipeline.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

2.1 Cohn's Test Pyramid Illustrated by Mukhin et al. [23] 9
2.2 Example Program and Corresponding CFG 11
2.3 GitLab CI/CD Workflow [W6] 17
2.4 OpenAPI Specification Document [W7] 20
2.5 Intended Versus Implemented Software Behavior in Applications by Thompson [61] 25
2.6 Fuzzing Algorithm Visualization [65] 30
2.7 Classification of Fuzzers by Beaman et al. [65] 31

3.1 Code Coverage of REST-based Fuzzers in Empirical Study by Kim et al. [6] 49
3.2 Line Coverage of REST-based Fuzzers in Comparison by Zhang and Arcuri [8] 50

4.1 Code Coverage of idp-server Module Achieved by Unit Tests 64
4.2 Code Coverage of idp-server Module Achieved by EvoMaster with OpenAPI Specification Created by springdoc 68
4.3 Schemathesis Report for Found Defects 70

5.1 Continuous Fuzzing Design Based on Klooster et al. [10] 76
5.2 Continuous Fuzzing Basic Pipeline Jobs and Their Dependencies 78
5.3 Continuous Fuzzing Merge Request Pipeline Jobs and Their Dependencies 79
5.4 Exposed Artifact on Merge Request Approval Screen 80
5.5 Total Detected Faults for Each Commit 86
5.6 Cumulative Unique Detected Faults for Each Commit 87

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

2.1 Examplary Equivalence Partitions by Graham et al. [20] 10
2.2 Detectable Vulnerabilities per Bug Oracle 35

3.1 Comparison of Traditional Fuzzers 41
3.2 Comparison of REST-based Fuzzers Based on Kim et al. [6] 51
3.3 Comparison of Existing Continuous Fuzzing Solutions 60

4.1 Comparison of Code Analysis vs. Traffic Capture Approach for OpenAPI Specification Document Generation 67
4.2 Fuzzer Results After 10 Minutes with Different OpenAPI Specification Generation Approaches 73

5.1 Detected Faults by the Fuzzers for Each Commit 88

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

2.1 Fuzz Testing Algorithm by Manès et al. [67] 28

List of Listings

3.1 Jazzer Fuzz Target [W26] 42
3.2 Javafuzz Fuzz Target [W16] 45
3.3 JQF Fuzz Target [38] 46
4.1 JUnit Test Generated by EvoMaster 69
5.1 NullPointerException in POST /extauth endpoint in ref-idp-server . . 84
5.2 Request Parameters for GET /extauth endpoint in ref-idp-server . . . 85
5.3 Unsafe Use of redirect_uri Parameter in ref-idp-server 85
5.4 Open Redirect Exploit for GET /extauth endpoint in ref-idp-server . . 86

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

AFL American Fuzzy Lop. 38, 40, 41, 43, 44, 46, 60, 61

API Application Programming Interface. ix, xi, 2, 18, 19, 39, 40, 43, 45, 47–49, 51–56, 58–62, 64–67, 93

bBOXRT black BOX tool for Robustness Testing. 47–51, 56, 57

BFS Breadth-First Search. 55

CFG Control Flow Graph. 10–12, 97

CI Continuous Integration. ix, xi, 1, 5, 13, 15–18, 37, 38, 60, 63, 75, 77, 94

CI/CD Continuous Integration/Continuous Deployment. 2, 3, 17, 18, 22, 37, 38, 61, 67, 75–77, 81, 89–91, 93, 97

CLI Command Line Interface. 64

CT Combinatorial Testing. 14, 58, 59

CVE Common Vulnerabilities and Exposures. 23

DOM Document Object Model. 22, 34

IDP Identity Provider. 63

ISTQB International Software Testing Qualifications Board. 8

JVM Java Virtual Machine. 40, 41, 50, 52, 60, 61

JWT JSON Web Token. 69, 90

MIO Many Independent Objective. 13, 14, 52

ODG Operation Dependency Graph. 55, 56

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

OWASP Open Web Application Security Project. 21–23

PoC Proof of Concept. 2, 3, 76, 89, 90, 92–94

PUT Program Under Test. 27–29, 31–33, 35, 38, 40, 42, 43, 45, 47, 50, 52–54, 56–58, 62, 63, 65–69, 75, 77–80, 90, 91

REST Representational State Transfer. 2, 18, 19, 35, 38–40, 47–50, 52–62, 64–66, 76, 89, 91–93

RQ Research Question. 3, 89–92

SaaS Software as a Service. 76

SDLC Software Development Lifecycle. ix, xi, 5, 24, 26, 90

SQL Structured Query Language. 22, 34, 35, 52, 53, 57, 87

SQLi SQL Injection. 22, 35, 38, 87

TDD Test-Driven Development. 12, 13, 17, 90

UI User Interface. 9, 19, 62, 92

URI Uniform Resource Identifier. 18, 70, 85

URL Uniform Resource Locator. 18, 22, 62, 66, 71, 82, 83, 87, 88

VM Virtual Machine. 61, 76

WTS Whole Test Suite. 13

XP Extreme Programming. 12, 15

XSS Cross-Site-Scripting. 22, 23, 34, 35, 38, 57

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[1] B. Fitzgerald and K.-J. Stol, „Continuous software engineering and beyond: Trends and challenges“, in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, 2014, pp. 1–9. DOI: 10.1145/2593812.2593813.

[2] B. P. Miller, L. Fredriksen, and B. So, „An empirical study of the reliability of unix utilities“, *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990. DOI: 10.1145/96267.96279.

[3] J. Li, B. Zhao, and C. Zhang, „Fuzzing: A survey“, *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018. DOI: 10.1186/s42400-018-0002-y.

[4] G. Tassey, „The economic impacts of inadequate infrastructure for software testing“, *National Institute of Standards and Technology*, vol. RTI Project Number 7007.011, 2002.

[5] K. Rindell, J. Ruohonen, J. Holvitie, S. Hyrynsalmi, and V. Leppänen, „Security in agile software development: A practitioner survey“, *Information and Software Technology*, vol. 131, p. 106 488, 2021. DOI: 10.1016/j.infsof.2020.106488.

[6] M. Kim, Q. Xin, S. Sinha, and A. Orso, „Automated test generation for rest apis: No time to rest yet“, in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual, South Korea: Association for Computing Machinery, 2022, pp. 289–301. DOI: 10.1145/3533767.3534401.

[7] M. Wang, J. Liang, C. Zhou, Y. Chen, Z. Wu, and Y. Jiang, „Industrial oriented evaluation of fuzzing techniques“, in *14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 306–317. DOI: 10.1109/ICST49551.2021.00043.

[8] M. Zhang and A. Arcuri, „Open problems in fuzzing restful apis: A comparison of tools“, 2022. DOI: 10.48550/ARXIV.2205.05325.

[9] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, „Fuzzing: State of the art“, *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018. DOI: 10.1109/TR.2018.2834476.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[10] T. Klooster, F. Turkmen, G. Broenink, R. T. Hove, and M. Böhme, „Effectiveness and scalability of fuzzing techniques in CI/CD pipelines“, 2022. DOI: 10.48550/ARXIV.2205.14964.

[11] Y. B. Leau, W. K. Loo, W. Y. Tham, and S. F. Tan, „Software development life cycle agile vs traditional approaches“, in *International Conference on Information and Network Technology*, vol. 37, 2012, pp. 162–167.

[12] M. Stoica, M. Mircea, and B. Ghilic-Micu, „Software development: Agile vs. traditional.“, *Informatica Economica*, vol. 17, no. 4, 2013. DOI: 10.12948/issn14531305/17.4.2013.06.

[13] A. Moniruzzaman and D. S. A. Hossain, „Comparative study on agile software development methodologies“, 2013. DOI: 10.48550/arXiv.1307.3356.

[14] S. Al-Saqqa, S. Sawalha, and H. Abdelnabi, „Agile software development: Methodologies and trends“, *International Journal of Interactive Mobile Technologies*, vol. 14, no. 11, 2020. DOI: 10.3991/ijim.v14i11.13269.

[15] R. Hoda, N. Salleh, and J. Grundy, „The rise and evolution of agile software development“, *IEEE Software*, vol. 35, no. 5, pp. 58–63, 2018. DOI: 10.1109/MS.2018.290111318.

[16] IEEE, „Standard glossary of software engineering terminology“, *IEEE Standard 610.12-1990*, pp. 1–84, 1990. DOI: 10.1109/IEEESTD.1990.101064.

[17] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, „The oracle problem in software testing: A survey“, *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015. DOI: 10.1109/TSE.2014.2372785.

[18] L. Rosenberg, T. Hammer, and J. Shaw, „Software metrics and reliability“, in *9th International Symposium on Software Reliability Engineering*, 1998.

[19] P. Bourque, R. Dupuis, A. Abran, J. Moore, and L. Tripp, „The guide to the software engineering body of knowledge“, *IEEE Software*, vol. 16, no. 6, pp. 35–44, 1999. DOI: 10.1109/52.805471.

[20] D. Graham, E. V. Veenendaal, I. Evans, and R. Black, *Foundations of Software Testing: ISTQB Certification*. Cengage Learning Business Press, 2006, ISBN: 1844803554.

[21] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2018.

[22] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*, 1st. Addison-Wesley Professional, 2009, ISBN: 0321579364.

[23] V. Mukhin, Y. Kornaga, Y. Bazaka, *et al.*, „The testing mechanism for software and services based on mike cohn's testing pyramid modification“, in *11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, 2021, pp. 589–595. DOI: 10.1109/IDAACS53288.2021.9660999.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[24] D. Spinellis, „State-of-the-art software testing“, *IEEE Software*, vol. 34, no. 5, pp. 4–6, 2017. DOI: 10.1109/MS.2017.3571564.

[25] N. Oh, P. Shirvani, and E. J. McCluskey, „Control-flow checking by software signatures“, *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002. DOI: 10.1109/24.994926.

[26] M. D. Weiser, J. D. Gannon, and P. R. McMullin, „Comparison of structural test coverage metrics“, *IEEE Software*, vol. 2, no. 2, pp. 80–85, 1985. DOI: 10.1109/MS.1985.230356.

[27] M. M. Tikir and J. K. Hollingsworth, „Efficient instrumentation for code coverage testing“, *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 86–96, 2002. DOI: 10.1145/566171.566186.

[28] V. Ramasamy and R. Hundt, „Dynamic binary instrumentation on ia-64“, in *Proceedings of the First EPIC Workshop*, 2001.

[29] J. C. King, „Symbolic execution and program testing“, *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976. DOI: 10.1145/360248.360252.

[30] V. Kettunen, J. Kasurinen, O. Taipale, and K. Smolander, „A study on agility and testing processes in software organizations“, in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010, pp. 231–240. DOI: 10.1145/1831708.1831737.

[31] I. Karac and B. Turhan, „What do we (really) know about test-driven development?“, *IEEE Software*, vol. 35, pp. 81–85, 2018. DOI: 10.1109/MS.2018.2801554.

[32] S. Stolberg, „Enabling agile testing through continuous integration“, in *Agile Conference*, IEEE, 2009, pp. 369–374. DOI: 10.1109/AGILE.2009.16.

[33] M. Polo, P. Reales, M. Piattini, and C. Ebert, „Test automation“, *IEEE Software*, vol. 30, no. 1, pp. 84–89, 2013. DOI: 10.1109/MS.2013.15.

[34] A. Arcuri, „Many independent objective (MIO) algorithm for test suite generation“, in *Search Based Software Engineering*, Springer International Publishing, 2017, pp. 3–17. DOI: 10.1007/978-3-319-66299-2_1.

[35] G. Fraser and A. Arcuri, „Whole test suite generation“, *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012. DOI: 10.1109/TSE.2012.14.

[36] R. Kuhn, D. Wallace, and A. M. Gallo, „Software fault interactions and implications for software testing“, *IEEE Transactions on Software Engineering*, vol. 30, pp. 418–421, 2004. DOI: 10.1109/TSE.2004.24.

[37] H. Wu, L. Xu, X. Niu, and C. Nie, „Combinatorial testing of restful apis“, *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 426–437, 2022. DOI: 10.1145/3510003.3510151.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[38] R. Padhye, C. Lemieux, and K. Sen, „Jqf: Coverage-guided property-based testing in java“, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 398–401. DOI: 10.1145/3293882.3339002.

[39] J. Malburg and G. Fraser, „Combining search-based and constraint-based testing“, in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 436–439. DOI: 10.1109/ASE.2011.6100092.

[40] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, „A survey on adaptive random testing“, *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2052–2083, 2021. DOI: 10.1109/TSE.2019.2942921.

[41] G. Booch, *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., 1990.

[42] K. Beck, „Embracing change with extreme programming“, *Computer*, vol. 32, no. 10, pp. 70–77, 1999. DOI: 10.1109/2.796139.

[43] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, „The impact of continuous integration on other software development practices: A large-scale empirical study“, in *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 60–71. DOI: 10.1109/ASE.2017.8115619.

[44] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, „Usage, costs, and benefits of continuous integration in open-source projects“, in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2016, pp. 426–437. DOI: 10.1145/2970276.2970358.

[45] Y. Yu, G. Yin, T. Wang, C. Yang, and H. Wang, „Determinants of pull-based development in the context of continuous integration“, *Science China Information Sciences*, vol. 59, 2016. DOI: 10.1007/s11432-016-5595-8.

[46] R. T. Fielding and R. N. Taylor, „Architectural styles and the design of network-based software architectures“, AAI9980887, Ph.D. dissertation, 2000, ISBN: 0599871180.

[47] C. Rodríguez, M. Baez, F. Daniel, *et al.*, „Rest apis: A large-scale analysis of compliance with principles and best practices“, in *Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings 16*, Springer, 2016, pp. 21–39. DOI: 10.1007/978-3-319-38791-8_2.

[48] Z. Hatfield-Dodds and D. Dygalo, „Deriving semantics-aware fuzzers from web api schemas“, in *IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 345–346. DOI: 10.1145/3510454.3528637.

[49] G. McGraw, „Software security“, *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004. DOI: 10.1109/MSECP.2004.1281254.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[50] R. A. Khan, S. U. Khan, H. U. Khan, and M. Ilyas, „Systematic mapping study on security approaches in secure software engineering“, *IEEE Access*, vol. 9, pp. 19 139–19 160, 2021. DOI: 10.1109/ACCESS.2021.3052311.

[51] N. Laranjeiro, J. Agnelo, and J. Bernardino, „A black box tool for robustness testing of rest services“, *IEEE Access*, vol. 9, pp. 24 738–24 754, 2021. DOI: 10.1109/ACCESS.2021.3056505.

[52] S. Samonas and D. Coss, „The cia strikes back: Redefining confidentiality, integrity and availability in security“, *Journal of Information System Security*, vol. 10, no. 3, 2014.

[53] H. H. AlBreiki and Q. H. Mahmoud, „Evaluation of static analysis tools for software security“, in *2014 10th International Conference on Innovations in Information Technology (IIT)*, 2014, pp. 93–98. DOI: 10.1109/INNOVATIONS.2014.6987569.

[54] O. B. Fredj, O. Cheikhrouhou, M. Krichen, H. Hamam, and A. Derhab, „An owasp top ten driven survey on web application protection methods“, in *International Conference on Risks and Security of Internet and Systems*, Springer, 2020, pp. 235–252. DOI: 10.1007/978-3-030-68887-5_14.

[55] M. Böhme, C. Cadar, and A. Roychoudhury, „Fuzzing: Challenges and reflections“, *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021. DOI: 10.1109/MS.2020.3016773.

[56] A. Doupé, M. Cova, and G. Vigna, „Why johnny can't pentest: An analysis of black-box web vulnerability scanners“, in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Kreibich and M. Jahnke, Eds., Springer Berlin Heidelberg, 2010, pp. 111–131. DOI: 10.1007/978-3-642-14215-4_7.

[57] M. A. Butt, Z. Ajmal, Z. I. Khan, M. Idrees, and Y. Javed, „An in-depth survey of bypassing buffer overflow mitigation techniques“, *Applied Sciences*, vol. 12, p. 6702, Jul. 2022. DOI: 10.3390/app12136702.

[58] P. Akritidis, „Cling: A memory allocator to mitigate dangling pointers.“, in *USENIX security symposium*, Washington DC, 2010, pp. 177–192.

[59] W. Dietz, P. Li, J. Regehr, and V. Adve, „Understanding integer overflow in c/c++“, in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 760–770. DOI: 10.1109/ICSE.2012.6227142.

[60] A. J. Jafari and A. Rasoolzadegan, „Security patterns: A systematic mapping study“, *Journal of Computer Languages*, vol. 56, p. 100 938, 2020. DOI: 10.1016/j.cola.2019.100938.

[61] H. Thompson, „Why security testing is hard“, *IEEE Security & Privacy*, vol. 1, no. 4, pp. 83–86, 2003. DOI: 10.1109/MSECP.2003.1219078.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[62] M. Kelly, C. Treude, and A. Murray, „A case study on automated fuzz target generation for large codebases“, in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–6. DOI: 10.1109/ESEM.2019.8870150.

[63] J. W. Duran and S. C. Ntafos, „An evaluation of random testing“, *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 438–444, 1984. DOI: 10.1109/TSE.1984.5010257.

[64] B. P. Miller, D. Koski, C. P. Lee, *et al.*, „Fuzz revisited: A re-examination of the reliability of unix utilities and services“, University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1995.

[65] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, „Fuzzing vulnerability discovery techniques: Survey, challenges and future directions“, *Computers and Security*, vol. 120, no. C, 2022. DOI: 10.1016/j.cose.2022.102813.

[66] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, „Finding software vulnerabilities by smart fuzzing“, in *Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 427–430. DOI: 10.1109/ICST.2011.48.

[67] V. J. M. Manes, H. Han, C. Han, *et al.*, „The art, science, and engineering of fuzzing: A survey“, *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021. DOI: 10.1109/TSE.2019.2946563.

[68] B. P. Miller, M. Zhang, and E. R. Heymann, „The relevance of classic fuzz testing: Have we solved this one?“, *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2028–2039, 2022. DOI: 10.1109/TSE.2020.3047766.

[69] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, „Kameleonfuzz: Evolutionary fuzzing for black-box xss detection“, in *Proceedings of the 4th ACM conference on Data and application security and privacy*, 2014, pp. 37–48. DOI: 10.1145/2557547.2557550.

[70] O. v. Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, „Webfuzz: Grey-box fuzzing for web applications“, in *European Symposium on Research in Computer Security*, Springer, 2021, pp. 152–172. DOI: 10.1007/978-3-030-88418-5_8.

[71] V. Atlidakis, P. Godefroid, and M. Polishchuk, „Restler: Stateful rest api fuzzing“, in *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 748–758. DOI: 10.1109/ICSE.2019.00083.

[72] A. Arcuri, J. P. Galeotti, B. Marculescu, and M. Zhang, „Evomaster: A search-based system test generation tool“, *Journal of Open Source Software*, vol. 6, no. 57, p. 2153, 2021. DOI: 10.21105/joss.02153.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[73] M. Böhme and B. Falk, „Fuzzing: On the exponential cost of vulnerability discovery“, in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 713–724. DOI: 10.1145/3368089.3409729.

[74] X. Zhu and M. Böhme, „Regression greybox fuzzing“, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2169–2182. DOI: 10.1145/3460120.3484596.

[75] A. Hazimeh, A. Herrera, and M. Payer, „Magma: A ground-truth fuzzing benchmark“, *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, 2021. DOI: 10.1145/3428334.

[76] D. Asprone, J. Metzman, A. Arya, G. Guizzo, and F. Sarro, „Comparing fuzzers on a level playing field with linebreak fuzzbench“, in *IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 302–311. DOI: 10.1109/ICST53961.2022.00039.

[77] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, „FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing“, in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3699–3715, ISBN: 978-1-939133-31-1.

[78] R. Kersten, K. Luckow, and C. S. Păsăreanu, „Poster: Afl-based fuzzing for java with kelinci“, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2511–2513. DOI: 10.1145/3133956.3138820.

[79] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, „AFL++: Combining incremental steps of fuzzing research“, in *14th USENIX Workshop on Offensive Technologies*, USENIX Association, 2020.

[80] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, „Empirical comparison of black-box test case generation tools for restful apis“, 2021, pp. 226–236. DOI: 10.1109/SCAM52516.2021.00035.

[81] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, „Test coverage criteria for restful web apis“, in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 15–21. DOI: 10.1145/3340433.3342822.

[82] E. Viglianisi, M. Dallago, and M. Ceccato, „Resttestgen: Automated black-box testing of restful apis“, in *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 142–152. DOI: 10.1109/ICST46399.2020.00024.

[83] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, „Restest: Black-box constraint-based testing of restful web apis“, in *International Conference on Service-Oriented Computing*, Springer, 2020, pp. 459–475. DOI: 10.1007/978-3-030-65310-1_33.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[84] J. Lin, T. Li, Y. Chen, *et al.*, „Forest: A tree-based approach for fuzzing restful apis“, 2022. DOI: 10.48550/ARXIV.2203.02906.

[85] A. Arcuri, „Evomaster: Evolutionary multi-context automated system test generation“, in *IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 394–397. DOI: 10.1109/ICST.2018.00046.

[86] A. Arcuri and J. P. Galeotti, „Handling sql databases in automated system test generation“, *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 4, 2020. DOI: 10.1145/3391533.

[87] D. C. Kozen, „Depth-first and breadth-first search“, in *The Design and Analysis of Algorithms*. New York, NY: Springer New York, 1992, pp. 19–24, ISBN: 978-1-4612-4400-4. DOI: 10.1007/978-1-4612-4400-4_4.

[88] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, „RESTest: Automated Black-Box Testing of RESTful Web APIs“, in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, 2021. DOI: 10.1145/3460319.3469082.

[89] K. Serebryany, „OSS-Fuzz - google's continuous fuzzing service for open source software“, USENIX Association, 2017.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Web Resources

[W1] Google, *OSS-Fuzz: Continuous Fuzzing for Open Source Software*, https://github.com/google/oss-fuzz, Accessed: 2023-06-29.

[W2] *Recommended minimum standard for vendor or developer verification of code*, https://www.nist.gov/itl/executive-order-improving-nations-cybersecurity/recommended-minimum-standard-vendor-or-developer, Accessed: 2023-06-29.

[W3] *Manifesto for agile software development*, http://www.agilemanifesto.org/, Accessed: 2023-06-29.

[W4] Digital.ai, *15th annual State of Agile Report*, https://digital.ai/resource-center/analyst-reports/state-of-agile-report, Accessed: 2023-06-29.

[W5] M. Fowler, *Continuous Integration*, https://martinfowler.com/articles/continuousIntegration.html, Accessed: 2023-06-29.

[W6] GitLab, *CI/CD concepts*, https://docs.gitlab.com/ee/ci/introduction/, Accessed: 2023-06-29.

[W7] *Swagger petstore*, https://petstore3.swagger.io/api/v3/openapi.yaml, Accessed: 2023-06-29.

[W8] *OpenAPI Specification*, https://swagger.io/specification/, Accessed: 2023-06-29.

[W9] *Swagger tools*, https://swagger.io/tools/open-source/, Accessed: 2023-06-29.

[W10] OWASP, *OWASP Top 10*, https://owasp.org/Top10/, Accessed: 2023-06-29.

[W11] *Cwe-601: Url redirection to untrusted site ('open redirect')*, https://cwe.mitre.org/data/definitions/601.html, Accessed: 2023-06-29.

[W12] *Buffer overflow vulnerability*, https://owasp.org/www-community/vulnerabilities/Buffer_Overflow, Accessed: 2023-06-29.

[W13] *Cve-2002-0639*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0639, Accessed: 2023-06-29.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[W14] *AddressSanitizerAlgorithm*, https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm, Accessed: 2023-06-29.

[W15] F. Meumertzheim, *Java Fuzzing With Jazzer*, https://www.code-intelligence.com/blog/engineering-jazzer, Accessed: 2023-06-29.

[W16] *Javafuzz: coverage-guided fuzz testing for Java*, https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/javafuzz, Accessed: 2023-06-29.

[W17] *libFuzzer - a library for coverage-guided fuzz testing*, https://llvm.org/docs/LibFuzzer.html, Accessed: 2023-06-29.

[W18] *libFuzzer - a library for coverage-guided fuzz testing*, https://github.com/llvm-mirror/llvm/blob/master/docs/LibFuzzer.rst, Accessed: 2023-06-29.

[W19] *Jazzer*, https://github.com/CodeIntelligenceTesting/jazzer, Accessed: 2023-06-29.

[W20] M. Zalewski, *American Fuzzy Lop*, https://lcamtuf.coredump.cx/afl/, Accessed: 2023-06-29.

[W21] M. Zalewski, *Technical "whitepaper" for afl-fuzz*, https://lcamtuf.coredump.cx/afl/technical_details.txt, Accessed: 2023-06-29.

[W22] *American Fuzzy Lop*, https://github.com/google/AFL, Accessed: 2023-06-29.

[W23] *AFL-based fuzzing for Java*, https://github.com/isstac/kelinci, Accessed: 2023-06-29.

[W24] *Javafuzz: coverage-guided fuzz testing for Java*, https://github.com/fuzzitdev/javafuzz, Accessed: 2023-06-29.

[W25] *JQF + Zest: Semantic Fuzzing for Java*, https://github.com/rohanpadhye/JQF, Accessed: 2023-06-29.

[W26] *Jazzer path traversal sanitizer example*, https://github.com/CodeIntelligenceTesting/jazzer/blob/main/examples/src/main/java/com/example/ExamplePathTraversalFuzzerHooks.java, Accessed: 2023-06-29.

[W27] *GitLab Acquires Peach Tech and Fuzzit to Expand its DevSecOps Offering*, https://about.gitlab.com/press/releases/2020-06-11-gitlab-acquires-peach-tech-and-fuzzit-to-expand-devsecops-offering.html, Accessed: 2023-06-29.

[W28] *Openapi specification*, https://oai.github.io/Documentation/introduction.html, Accessed: 2023-06-29.

[W29] *Oss-fuzz documentation*, https://google.github.io/oss-fuzz/, Accessed: 2023-06-29.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[W30] Google, *ClusterFuzz*, https://github.com/google/clusterfuzz, Accessed: 2023-06-29.

[W31] Google, *ClusterFuzzLite*, https://github.com/google/clusterfuzzlite, Accessed: 2023-06-29.

[W32] Microsoft, *OneFuzz: A self-hosted Fuzzing-As-A-Service platform*, https://github.com/microsoft/onefuzz, Accessed: 2023-06-29.

[W33] GitLab, *Coverage-guided fuzz testing*, https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/, Accessed: 2023-06-29.

[W34] GitLab, *Api fuzzing*, https://docs.gitlab.com/ee/user/application_security/api_fuzzing/, Accessed: 2023-06-29.

[W35] gematik, *Idp server*, https://github.com/gematik/ref-idp-server, Accessed: 2023-06-29.

[W36] *Openid connect*, https://openid.net/connect/, Accessed: 2023-06-29.

[W37] Apache, *Maven*, https://maven.apache.org/, Accessed: 2023-06-29.

[W38] *Maven plugin for building docker images*, https://github.com/fabric8io/docker-maven-plugin, Accessed: 2023-06-29.

[W39] *Spring boot reference documentation*, https://docs.spring.io/spring-boot/docs/current/reference/html/, Accessed: 2023-06-29.

[W40] *Springdoc-openapi*, https://springdoc.org/v2/, Accessed: 2023-06-29.

[W41] *Pcap2har-go*, https://github.com/colinnewell/pcap2har-go, Accessed: 2023-06-29.

[W42] *Har2openapi*, https://github.com/dcarr178/har2openapi, Accessed: 2023-06-29.

[W43] *EvoMaster*, https://github.com/EMResearch/EvoMaster, Accessed: 2023-06-29.

[W44] *Schemathesis*, https://github.com/schemathesis/schemathesis, Accessed: 2023-06-29.

[W45] Microsoft, *RESTler*, https://github.com/microsoft/restler-fuzzer, Accessed: 2023-06-29.

[W46] *Restler commit fixing nullreferenceexception*, https://github.com/microsoft/restler-fuzzer/commit/cc20a5ae95d89e8b81fde9cad101d6eb2e6e4077, Accessed: 2023-06-29.

[W47] *Restler issue: Unexpected response without prior request*, https://github.com/microsoft/restler-fuzzer/issues/231, Accessed: 2023-06-29.

[W48] *RestTestGen*, https://github.com/SeUniVr/RestTestGen, Accessed: 2023-06-29.

[W49] *Evorefuzz*, https://git.dei.uc.pt/cnl/bBOXRT/tree/master/EvoReFuzz, Accessed: 2023-06-29.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[W50] *RESTest*, https://github.com/isa-group/RESTest, Accessed: 2023-06-29.

[W51] *RestCT*, https://github.com/GIST-NJU/RestCT, Accessed: 2023-06-29.

[W52] *Api-spec-converter*, https://github.com/LucyBot-Inc/api-spec-converter, Accessed: 2023-06-29.

[W53] GitLab, *Runner*, https://docs.gitlab.com/runner/, Accessed: 2023-06-29.

[W54] GitLab, *Pricing*, https://about.gitlab.com/pricing/, Accessed: 2023-06-29.

[W55] GitLab, *Installation*, https://docs.gitlab.com/ee/install/docker.html#install-gitlab-using-docker-compose, Accessed: 2023-06-29.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Appendix

Listing 1 shows the implemented EvoMasterController which extends EvoMaster's EmbeddedSutController and implements the required functionality. It is used for white-box fuzzing the test target, i.e., the ref-idp-server introduced in Chapter 4.

```
1  package de.gematik.idp.server;
2
3  import org.evomaster.client.java.controller.EmbeddedSutController;
4  import org.evomaster.client.java.controller.InstrumentedSutStarter;
5  import org.evomaster.client.java.controller.api.dto.AuthenticationDto;
6  import org.evomaster.client.java.controller.api.dto.SutInfoDto;
7  import org.evomaster.client.java.controller.api.dto.database.schema.DatabaseType;
8  import org.evomaster.client.java.controller.internal.SutController;
9  import org.evomaster.client.java.controller.internal.db.DbSpecification;
10 import org.evomaster.client.java.controller.problem.ProblemInfo;
11 import org.evomaster.client.java.controller.problem.RestProblem;
12 import org.springframework.boot.SpringApplication;
13 import org.springframework.context.ConfigurableApplicationContext;
14 import org.springframework.jdbc.CannotGetJdbcConnectionException;
15 import org.springframework.jdbc.core.JdbcTemplate;
16
17 import java.sql.Connection;
18 import java.sql.SQLException;
19 import java.util.Arrays;
20 import java.util.List;
21
22 public class EvoMasterController extends EmbeddedSutController {
23     private ConfigurableApplicationContext ctx;
24     private Connection sqlConnection;
25
26     public static void main(String[] args) {
27         SutController controller = new EvoMasterController();
28         InstrumentedSutStarter starter = new InstrumentedSutStarter(controller);
29         starter.start();
30     }
31
32     @Override
33     public boolean isSutRunning() {
34         return ctx != null && ctx.isRunning();
35     }
36
37     @Override
38     public String getPackagePrefixesToCover() {
39         return "de.gematik.idp.server";
40     }
41
42     @Override
43     public List<AuthenticationDto> getInfoForAuthentication() {
44         return null;
45     }
46
47     @Override
48     public ProblemInfo getProblemInfo() {
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
49          return new RestProblem(System.getenv("CI_PROJECT_DIR") + "/idp-server/target/idp-server.
                yaml", null);
50      }
51
52      @Override
53      public SutInfoDto.OutputFormat getPreferredOutputFormat() {
54          return SutInfoDto.OutputFormat.JAVA_JUNIT_5;
55      }
56
57      @Override
58      public String startSut() {
59          ctx = SpringApplication.run(IdpServer.class);
60          JdbcTemplate jdbc = ctx.getBean(JdbcTemplate.class);
61          try {
62              sqlConnection = jdbc.getDataSource().getConnection();
63          } catch (SQLException e) {
64              throw new RuntimeException(e);
65          }
66          return "http://localhost:8080";
67      }
68
69      @Override
70      public void stopSut() {
71          ctx.stop();
72      }
73
74      @Override
75      public void resetStateOfSUT() {
76      }
77
78      @Override
79      public List<DbSpecification> getDbSpecifications() {
80          try {
81              return Arrays.asList(new DbSpecification(DatabaseType.H2, sqlConnection));
82          } catch (CannotGetJdbcConnectionException e) {
83              System.out.println("SQL spec error");
84              System.out.println(e.getMessage());
85              return null;
86          }
87      }
88  }
```

Listing 1: Implemented EvoMasterController for ref-idp-server

In Listing 2, the implemented continuous fuzzing solution is given. It describes the jobs and stages used in GitLab's basic pipeline and merge request pipeline. A detailed description of all the jobs and their steps is presented in Section 5.3.

```
1   image: maven:3.8.7-openjdk-18
2
3   variables:
4     TIME_BUDGET: 500s
5     TIME_BUDGET_MERGE: 3000s
6     SCHEMATHESIS_MAX_EXAMPLES: 1500
7     SCHEMATHESIS_OUTPUT_FILE: $CI_COMMIT_SHORT_SHA/schemathesis.log
8     EVOMASTER_OUTPUT_PATH: $CI_COMMIT_SHORT_SHA/evomaster
9     GIT_CLEAN_FLAGS: none
10    MAVEN_OPTS: >-
11      -Dmaven.repo.local=$CI_PROJECT_PATH/m2/repository
12      -Djava.awt.headless=true
13
14  cache:
15    key: "$CI_COMMIT_REF_SLUG"
16    paths:
17      - $CI_PROJECT_PATH/m2/repository
18
19  stages:
20    - build
21    - test
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
22    - fuzz
23    - report
24
25  build:
26    stage: build
27    script:
28      - mvn clean install -pl idp-server -am $MAVEN_CLI_OPTS -Dskip.unittests -Dskip.inttests
29
30  test:
31    stage: test
32    script:
33      - mvn verify -pl idp-server -am $MAVEN_CLI_OPTS -Dgenerate-springdoc -Dskip.dockerbuild
34    artifacts:
35      paths:
36        - idp-server/target/idp-server.yaml
37
38  fuzz-schemathesis:
39    stage: fuzz
40    image: $CI_REGISTRY_IMAGE
41    except:
42      - merge_requests
43    script:
44      - docker run --rm -d --network gitlab_instance_default --name idp-server-schemathesis ref-idp-
            server-cifuzz:21.0.22
45      - sleep 10
46      - schemathesis run idp-server/target/idp-server.yaml --base-url http://idp-server-schemathesis
            :8080 --max-response-time 1000 --request-timeout 1000 --hypothesis-deadline 1000 --
            hypothesis-max-examples $SCHEMATHESIS_MAX_EXAMPLES 2>&1 | tee $SCHEMATHESIS_OUTPUT_FILE ||
             true
47    after_script:
48      - docker stop idp-server-schemathesis
49    artifacts:
50      when: always
51      paths:
52        - $SCHEMATHESIS_OUTPUT_FILE
53    needs:
54      - job: test
55        artifacts: true
56
57  fuzz-evomaster:
58    stage: fuzz
59    image: $CI_REGISTRY_IMAGE
60    except:
61      - merge_requests
62    script:
63      - docker run --rm -d --network gitlab_instance_default --name idp-server-evomaster ref-idp-
            server-cifuzz:21.0.22
64      - sleep 10
65      - evomaster --blackBox true --bbSwaggerUrl file://$CI_PROJECT_DIR/idp-server/target/idp-server.
            yaml --bbTargetUrl http://idp-server-evomaster:8080 --maxTime $TIME_BUDGET --outputFolder
            $EVOMASTER_OUTPUT_PATH --outputFormat JAVA_JUNIT_5 --problemType REST
66    after_script:
67      - docker stop idp-server-evomaster
68    artifacts:
69      paths:
70        - $EVOMASTER_OUTPUT_PATH
71    needs:
72      - job: test
73        artifacts: true
74
75  fuzz-whitebox:
76    stage: fuzz
77    image: $CI_REGISTRY_IMAGE
78    only:
79      - merge_requests
80    script:
81      - mvn test-compile -pl idp-server -am $MAVEN_CLI_OPTS
82      - mvn dependency:build-classpath -Dmdep.outputFile=mvn_cp.txt -pl idp-server -am
            $MAVEN_CLI_OPTS
83      - cd idp-server
84      - echo -n "-cp $PWD/target/classes/:$PWD/target/test-classes/:" > cp.txt
85      - cat mvn_cp.txt >> cp.txt
86      - java -Djdk.attach.allowAttachSelf=true --add-opens java.base/java.util.regex=ALL-UNNAMED --
```

```
          add-opens java.base/java.net=ALL-UNNAMED --add-opens java.base/java.lang=ALL-UNNAMED -
          Dfile.encoding=UTF-8 @cp.txt de.gematik.idp.server.EvoMasterController > server.log &
87      - sleep 5
88      - java -Djdk.attach.allowAttachSelf=true --add-opens java.base/java.net=ALL-UNNAMED --add-opens
           java.base/java.util=ALL-UNNAMED -jar /evomaster.jar --maxTime $TIME_BUDGET_MERGE --
          outputFolder ../$EVOMASTER_OUTPUT_PATH
89    artifacts:
90      paths:
91        - $EVOMASTER_OUTPUT_PATH
92    needs:
93      - job: test
94        artifacts: true
95
96  fuzz-report:
97    stage: report
98    image: $CI_REGISTRY_IMAGE
99    script:
100     - report
101   artifacts:
102     paths:
103       - report.txt
104     expose_as: 'Fuzzing Report'
105     when: always
106   needs:
107     - job: fuzz-schemathesis
108       artifacts: true
109       optional: true
110     - job: fuzz-evomaster
111       artifacts: true
112       optional: true
113     - job: fuzz-whitebox
114       artifacts: true
115       optional: true
```

Listing 2: gitlab-ci.yml for Continuous Fuzzing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub