FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Automated Repair of Security Vulnerabilities using Coverage-guided Fuzzing

**João Fernando da Costa Meireles Barbosa**

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira, Assistant Professor

Second Supervisor: André Restivo, Assistant Professor

July 25, 2021

# Automated Repair of Security Vulnerabilities using Coverage-guided Fuzzing

**João Fernando da Costa Meireles Barbosa**

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Nuno Macedo
External Examiner: Prof. João Barraca
Supervisor: Prof. Hugo Sereno Ferreira

July 25, 2021

# Abstract

Software development is an increasingly large and complex task, as it is applied to a broader range of domains. Moreover, as software assumes more business and safety-critical applications, its development requires more intricate testing and quality assurance processes, and lower maintenance and repair times. Thus, the concept of Software Development Life Cycle (SDLC) appeared as a process to develop and maintain software. Each cycle comprises multiple stages, one of which is software testing, a key process in avoiding software regression and detecting new faults. Fuzzing is a software testing method proven effective at discovering security vulnerabilities by generating malformed inputs and monitoring the faulty program.

Due to software's increasing responsibility, faults have a more significant negative impact. While techniques have been developed to improve and increase fault detection, debugging and repairing faults remains a time-consuming manual process. Automated Program Repair (APR) is a research field focused on localizing and repairing software faults automatically, using a set of tests as program specification. While it has the potential to reduce debugging costs, most current APR research is focused on benchmarks and synthetic experiments rather than developing and evaluating APR tools in practical environments. Fuzzing techniques provide a source of faults and generated test cases to feed APR. Furthermore, the introduction of continuous fuzzing workflows increases the inflow of bug reports, providing a use case for APR integrated with the SDLC.

Since APR techniques are not evaluated in realistic environments and there is a lack of adoption of APR in industrial software, there is little scientific evidence supporting that APR can correctly patch real-life software projects. We believe APR integration with continuous fuzzing workflows is promising, given the wealth of information fuzzing processes can provide to enhance APR techniques and the high demand for fault repair caused by the adoption of continuous fuzzing workflows.

However, preliminary experiments with current APR techniques suggest that they are not suitable for such an integration. Consequently, we develop a study on the ability of current APR techniques to generate correct patches using coverage-guided fuzzing tools to identify faults. We apply this process to five simple synthetic scenarios and four real-life vulnerabilities. None of the APR techniques generated a correct patch for any of the faults, which is concerning, especially since *Angelix* could not patch *Heartbleed*, as otherwise stated by their authors [42]. In light of this result, we analyze the original *Angelix* experiments and discover that *Angelix* does not patch *Heartbleed* by itself, compromising the credibility of APR research. Regardless, we are confident in the potential of APR and suggest shifting research towards the development of practical tools, listing some recommendations for developing such tools, namely ones leveraging fuzzing.

**Keywords**: Automated Program Repair. Software Development Process. Software Development Life Cycle. Fuzzing. Greybox Fuzzing. Coverage-guided Fuzzing. Continuous Fuzzing.

# Resumo

O desenvolvimento de *software* é uma tarefa cada vez maior e mais complexa, sendo aplicada a uma gama de domínios cada vez maior. Além disso, conforme o *software* assume aplicações cada vez mais críticas para negócios e para a segurança, o seu desenvolvimento requer processos de testagem e garantia de qualidade mais complexos e tempos de reparação e manutenção menores. Assim, o conceito de ciclo de vida de desenvolvimento de *software* (SDLC) surge como um processo para o desenvolvimento e manutenção de *software*. Cada ciclo é composto por múltiplas etapas, uma delas a de testes, que consiste na deteção de novas falhas e de falhas recorrentes. *Fuzzing* é um método de testes eficaz na descoberta de vulnerabilidades de segurança através da geração automática de *inputs* malformados e da monitorização do programa defeituoso.

Devido à responsabilidade crescente assumida pelo *software*, as suas falhas têm um impacto negativo cada vez mais significativo. Enquanto foram desenvolvidas técnicas para melhorar e aumentar a deteção de falhas, os processos de depuração e de reparação de falhas continuam a ser demorados e manuais. Reparação automática de programas (APR) é um campo de pesquisa focado na deteção e localização automática de falhas, usando um conjunto de testes como especificação do programa. Apesar de APR ter o potencial de reduzir os custos de depuração, grande parte da pesquisa atual está focada na utilização de modelos de referência e de experiências sintetizadas do que em desenvolver e avaliar ferramentas APR em ambientes reais. As técnicas de *fuzzing* servem como uma fonte de falhas e de casos de testes gerados para alimentar ferramentas APR. Além do mais, o aparecimento de fluxos contínuos de *fuzzing* causa um aumento do influxo de relatórios de erros, tornando-se num caso de uso para APR integrado no SDLC.

Uma vez que muitas das técnicas de APR existentes não são avaliadas em ambientes reais e existe uma falta de adoção de APR em *software* industrial, existe pouca evidência científica de que APR consegue corrigir corretamente falhas em projetos reais. Nós acreditamos que existe potencial para a integração de APR com *fuzzing* contínuo, dado o manancial de informação que os processos de *fuzzing* disponibilizam para melhorar as técnicas de APR e a demanda alta para reparar falhas originadas pela adoção de fluxos contínuos de *fuzzing*.

No entanto, as nossas experiências iniciais com técnicas de APR sugerem que elas não estão adequadas para uma integração deste género. Consequentemente, estudamos a habilidade de estas técnicas de APR corrigirem erros a partir de falhas identificadas por ferramentas de *fuzzing* baseadas em cobertura. Aplicamos este processo a cinco cenários sintetizados simples e a quatro vulnerabilidades reais. Nenhuma das técnicas de APR usadas corrigiram alguma das falhas, um aspeto preocupante, sobretudo porque o *Angelix* foi incapaz de corrigir o *Heartbleed*, ao contrário do que foi indicado pelos seus autores [42]. Face a estes resultados, analisamos as experiências originais do *Angelix* e descobrimos que este não corrige o *Heartbleed* por si próprio, comprometendo a credibilidade da investigação sobre APR. De qualquer forma, estamos confiantes no potencial de APR, sugerimos mudar o foco de pesquisa para o desenvolvimento de ferramentas práticas, e listamos recomendações para o desenvolvimento de ferramentas APR, nomeadamente baseadas em *fuzzing*.

# Acknowledgements

Throughout my *Master's Degree* and the development of this dissertation, I have received a lot of support and assistance from professors, colleagues, friends, and family, to whom I'd like to extend my sincerest gratitude.

First, I would like to thank my supervisors, Hugo Sereno Ferreira and André Restivo, for the opportunity to work on this dissertation. Their guidance and invaluable expertise helped in pushing my effort to a higher level.

To all my friendships made and strengthened during my life and academic path in FEUP, providing me with happy memories and helping me grow. To the nights well spent on Discord, which helped me keep clear-minded throughout this pandemic. My thanks to César Pinho and Rui Guedes for all the laughs, knowledge, and insightful discussions.

My warmest thanks go to my parents, who provided for me all my life and made me who I am today. I am eternally grateful for their emotional presence and support.

João Barbosa

*"The test of the machine is the satisfaction it gives you.*
*There isn't any other test. If the machine produces tranquility it's right.*
*If it disturbs you it's wrong until either the machine or your mind is changed."*

Robert M. Pirsig

# Contents


**1 Introduction** **1**
1.1 Context . . . 1
1.2 Motivation . . . 2
1.3 Problem Definition . . . 2
1.4 Goals . . . 3
1.5 Document Structure . . . 3

**2 Background** **5**
2.1 Automated Program Repair . . . 5
2.1.1 Fault Localization . . . 5
2.1.2 Heuristic Repair . . . 6
2.1.3 Constraint-based Repair . . . 6
2.1.4 Learning-based Repair . . . 6
2.2 APR Benchmarking . . . 7
2.3 Fuzzing . . . 7
2.4 Summary . . . 8

**3 State of the Art** **9**
3.1 Survey Research Questions . . . 9
3.2 Procedure . . . 10
3.3 Results . . . 11
3.3.1 APR Techniques . . . 11
3.3.2 Practical APR Solutions . . . 13
3.3.3 Categorization . . . 14
3.3.4 Analysis . . . 16
3.4 Summary . . . 19

**4 Problem Statement** **21**
4.1 Research Gaps . . . 21
4.2 Hypothesis . . . 22
4.3 Scope . . . 23
4.4 Research Questions . . . 23
4.5 Summary . . . 24

**5 Preliminary Work** **25**
5.1 Patch Suggestion through a GitHub Bot . . . 26
5.1.1 Objectives . . . 26
5.1.2 Initial Implementation . . . 27

5.2 Early Experiments with *Angelix* . . . 27
5.3 Early Experiments with Coverage-guided Fuzzing . . . 30
5.3.1 Fuzzing with AFL++ . . . 30
5.3.2 Automating the Fuzzing Process . . . 31
5.4 Overview . . . 33

**6 Empirical Study** **35**
6.1 Objectives . . . 35
6.2 Planning . . . 36
6.2.1 Joint Fuzzing–APR Process . . . 36
6.2.2 Project and Fault Selection . . . 38
6.2.3 APR Technique Selection . . . 39
6.2.4 Fuzzing Configuration . . . 41
6.2.5 APR Configuration . . . 43
6.3 Results . . . 46
6.3.1 Synthetic Programs . . . 46
6.3.2 Real-life Projects . . . 47
6.3.3 Analysis . . . 47
6.4 Replication of Original APR Experiments . . . 49
6.5 Threats to Validity . . . 50
6.5.1 Internal Validity . . . 50
6.5.2 External Validity . . . 52
6.6 Discussion . . . 53
6.7 Summary . . . 55

**7 Conclusions** **57**
7.1 Summary . . . 57
7.2 Contributions . . . 58
7.3 Future Work . . . 58

**A Synthetic Programs** **61**
A.1 CWE-121 (Stack Overflow) - Restricted Access . . . 61
A.2 CWE-122 (Heap Overflow) - Brainfuck Interpreter . . . 63
A.3 CWE-369 (Divide by Zero) - Division Operation . . . 67
A.4 CWE-134 (Format String) - String Customizer . . . 69
A.5 CWE-190 (Integer Overflow) - Useless Computation . . . 71

**References** **73**

# List of Figures


2.1 Overview of the different APR methods . . . . . . . . . . . . . . . . . . . . . . 6
2.2 Architectural diagram of a fuzzing engine . . . . . . . . . . . . . . . . . . . . . 8

3.1 Evolution of Target Languages used by APR techniques . . . . . . . . . . . . . . 18

5.1 Activity diagram of a pull request workflow from the bot's perspective . . . . . . 27
5.2 Patch diff file for the small program with a stack overflow vulnerability . . . . . 28
5.3 Diff file between the small program with a stack overflow vulnerability and its instrumented version . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.4 Test oracle script with two positive tests and one negative test . . . . . . . . . . 29
5.5 Example of the AFL++ status screen . . . . . . . . . . . . . . . . . . . . . . . 30
5.6 Example of a message to start a new fuzzing session . . . . . . . . . . . . . . . 32
5.7 Example of a message to stop an ongoing fuzzing session . . . . . . . . . . . . 32
5.8 Class diagram of the fuzzing service . . . . . . . . . . . . . . . . . . . . . . . 32

6.1 High-level architecture of the fuzzing–APR process . . . . . . . . . . . . . . . . 37
6.2 Official Heartbleed patch diff for the TLS implementation of OpenSSL . . . . . . 40
6.3 *Angelix*'s Heartbleed Patch diff for the TLS implementation of OpenSSL . . . . . 40
6.4 Example of a fuzz target in *libFuzzer* . . . . . . . . . . . . . . . . . . . . . . . 42
6.5 *read_line* function that reads the contents and size of a file . . . . . . . . . . . . 43
6.6 Generated Heartbleed patch diff file using our process . . . . . . . . . . . . . . . 48
6.7 Dockerfile for reproducing *Angelix*'s original experiments . . . . . . . . . . . . . 49
6.8 OpenSSL source diff applied before running *Angelix* . . . . . . . . . . . . . . . 50
6.9 Standard output of a *Prophet* execution . . . . . . . . . . . . . . . . . . . . . . 52

# List of Tables


3.1 Categorization of APR techniques . . . . . 15
3.2 Categorization of practical APR solutions within the SDLC software integration step 16

6.1 List of synthetic programs based on their vulnerability types . . . . . 38
6.2 List of real-life vulnerabilities and respective projects . . . . . 38
6.3 List of selected APR techniques, and their primary attributes . . . . . 41
6.4 Fuzzing engine environment information . . . . . 41
6.5 APR technique environment information . . . . . 44
6.6 APR results with synthetic programs identified by their weakness identifier . . . 46
6.7 APR results with real-life projects identified by their vulnerability identifier . . . 47

# Abbreviations

| | |
|---|---|
| ADT | Abstract Data Type |
| AFL | American Fuzzy Lop |
| ANDF | Architecture-Neutral Distribution Format |
| API | Application Programming Interface |
| APR | Automated Program Repair |
| AST | Abstract Syntax Tree |
| CAD | Computer-Aided Design |
| CASE | Computer-Aided Software Engineering |
| CBRS | Component-Based Repair Synthesis |
| CORBA | Common Object Request Broker Architecture |
| CI | Continuous Integration |
| CIL | C Intermediate Language |
| CVE | Common Vulnerability Enumeration |
| CVSS | Common Vulnerability Scoring System |
| CWE | Common Weakness Enumeration |
| DTLS | Datagram Transport Layer Security |
| FL | Fault Localization |
| IR | Information Retrieval |
| LOC | Lines of Code |
| LTO | Link Time Optimization |
| NVD | National Vulnerability Database |
| RFC | Request For Comments |
| RQ | Research Question |
| SDLC | Software Development Life Cycle |
| TLS | Transport Layer Security |
| UNCOL | UNiversal COmpiler-oriented Language |
| URL | Uniform Resource Locator |
| WWW | *World Wide Web* |

# Chapter 1

# Introduction


1.1 Context . . . . . . . . . . 1
1.2 Motivation . . . . . . . . . . 2
1.3 Problem Definition . . . . . . . . . . 2
1.4 Goals . . . . . . . . . . 3
1.5 Document Structure . . . . . . . . . . 3


This chapter introduces the motivation, problem, and scope of this dissertation. Section 1.1 details the context of the software development process, fuzzing, and Automated Program Repair (APR). Section 1.2 explains the benefits of APR and its use-cases within the Software Development Life Cycle (SDLC). Then, Section 1.3 explains the issues with the lack of adoption of APR in software development and with the inflow of faults from continuous fuzzing workflows. Section 1.4 describes the objectives of this dissertation. Finally, Section 1.5 describes the structure and contents of this document.

## 1.1 Context

The development of software is a complex task that is applied to an increasing number of domains. During the history of software engineering, the concept of a software development process, also known as the SDLC, was introduced to allow for the development and management of large-scale software. The SDLC is currently a crucial process in the software industry and represents how software developers develop, maintain, and integrate software.

With development, there is the appearance of bugs, errors that cause a program not to behave as intended. They may cause crashes, security concerns, and more severe effects like the loss of human lives [32]. To combat this, the SDLC includes debugging and testing processes to find and resolve bugs, often aided by code analysis tools. One such testing process is fuzzing, which monitors program behavior using automatically generated test cases to discover bugs and security vulnerabilities.

While testing techniques have evolved to improve and increase fault detection, debugging and repairing faults remains a time-consuming manual process. APR is an emerging research field aimed at automating the repair of software errors and vulnerabilities, taking as input a faulty program and a correctness specification that the program should meet (most research techniques use test suites). There is a variety of APR techniques that detect locations in the source code triggering errors (a process known as Fault Localization [52]) and synthesize patches that repair faults [26].

## 1.2 Motivation

Ensuring quality and debugging is a time-consuming yet necessary process in software development. APR has the potential of reducing the cost and effort of debugging and building quality software, allowing developers to be more productive [26], and reducing the negative impact of bugs in production. Unfortunately, current APR research focuses primarily on experimenting with new techniques in synthetic environments and benchmarks rather than developing APR solutions for practical use cases. Regardless, there has been some effort in researching APR's practical applications lately, namely integrating APR with the SDLC. *PAPRika* is a Visual Studio Code extension leveraging test-based APR to provide live feedback to the developer about issues in the code, localizing faults and suggesting patches for it. Developers accepted and felt comfortable using this extension, improving code quality and accelerating repair time [13]. In the software integration step of the SDLC, a few practical APR solutions have been developed, and their research shows that developers are receptive towards being suggested patches by bots [50, 39].

Continuous fuzzing workflows, which automatically fuzz newer software builds and identify faults, also provide use-cases for APR within the SDLC. Combining fuzzing with APR broadens the spectrum of faults benefited by APR. Moreover, it provides a fault-triggering test case and a set of generated test cases to enrich our test-based specification automatically.

## 1.3 Problem Definition

APR is not widely available and adopted in the software industry, despite the potential benefits it has. An explanation for the lack of adoption is that APR research has been focused on expanding and demonstrating its capabilities through benchmarks rather than exploring practical applications and integrations with the SDLC. While some practical solutions exist, they do not attract developers enough to be applied in industrial and large-scale software. As a result, the ability of APR to generate correct patches for faults in practical software projects is barely known.

Continuous fuzzing solutions such as OSS-Fuzz [3] increase the rate at which faults are discovered by fuzzing methods. As a result, developers cannot fully keep up with generated issues, leading to slower repair times and an increasing overload of lower priority bug reports. Aggravating this issue is the concept of fuzz blockers, faults that block the fuzzing process and prevent

further exploration until fixed, leading to a cascading effect where repairing a fault triggers more faults.

Chapter 4 expands on this problem definition, describing research gaps based on our review of the state of the art, and defining our research questions and scope.

## 1.4 Goals

The primary goal of this dissertation is to evaluate the effectiveness of APR when combined with fuzzing and applied in practical environments. We characterize APR effectiveness as the ability to generate correct patches for software faults. To that end, we implement a process leveraging coverage-guided fuzzing to identify faults and generate a set of test cases (one of which triggers the fault) that are then fed to APR techniques to generate patches. To integrate these patches with the faulty software, we propose implementing a GitHub Bot capable of creating and managing pull requests to suggest patches, apply code review suggestions, and interact with developers using textual commands.

Additionally, we intend to identify which types of faults APR is more effective at repairing, when applied with fuzzing, and identify how APR research and the development of new APR tools can be adapted to perform better with fuzzing and in practical applications.

## 1.5 Document Structure

This document is followed by six more chapters, structured as follows:

- Chapter 2, **Background**, introduces background information and nomenclature required for the full understanding of this document.
- Chapter 3, **State of the Art**, describes the methodology and results of a literature review on the current state of the art of open-source APR techniques and of APR solutions for practical use-cases.
- Chapter 4, **Problem Statement**, identifies research gaps and presents the problem we aim to solve, our objectives, and our planned approach to solve it.
- Chapter 5, **Preliminary Work**, provides an overview of our initial experiments with coverage-guided fuzzing and APR, as well as describes our initial development of solutions automating fuzzing and patch suggestion.
- Chapter 6, **Empirical Study**, describes the development of a join fuzzing–APR process and analyzes the results obtained from applying this process to a set of faulty programs with various APR techniques.
- Chapter 7, **Conclusions** summarizes our developed work, concludes this dissertation, and provides insights for future development and more thorough studies.

# Chapter 2

# Background


2.1 Automated Program Repair . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 APR Benchmarking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 Fuzzing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8


This chapter describes some fundamental concepts on APR and fuzzing, essential for the further understanding of this dissertation. Section 2.1 introduces the definition and goals of APR and describes current approaches to APR at a high-level. Section 2.2 mentions some examples of benchmarks used to evaluate APR techniques and how they may be misleading in practical use-cases. Section 2.3 introduces the concept of fuzzing, and how it has evolved.

## 2.1 Automated Program Repair

APR is an emerging research field in the automated generation of patches to solve faults, taking as input a faulty program and a correctness specification that the program should meet (most research techniques use test suites). A **fault** is the cause of an error, which is a condition that may lead to a failure, a deviation of a service from its correct behavior [7]. Most APR approaches start with a **Fault Localization** (2.1.1) stage and can be categorized on a high level between two high-level methods, based on how they generate and validate patches: (i) **Heuristic Repair** (2.1.2), also known as generate-and-test; and (ii) **Constraint-based repair** (2.1.3). Both of these methods can be complemented with **Learning-based repair** (2.1.4) techniques [26].

### 2.1.1 Fault Localization

Fault localization (FL) is a dedicated research field on its own, focused on the identification of the locations of faults (i.e. identifying code statements responsible for a fault). Wong et al. [52] conducted a survey and classified FL techniques into eight categories: slice-based, spectrum-based,

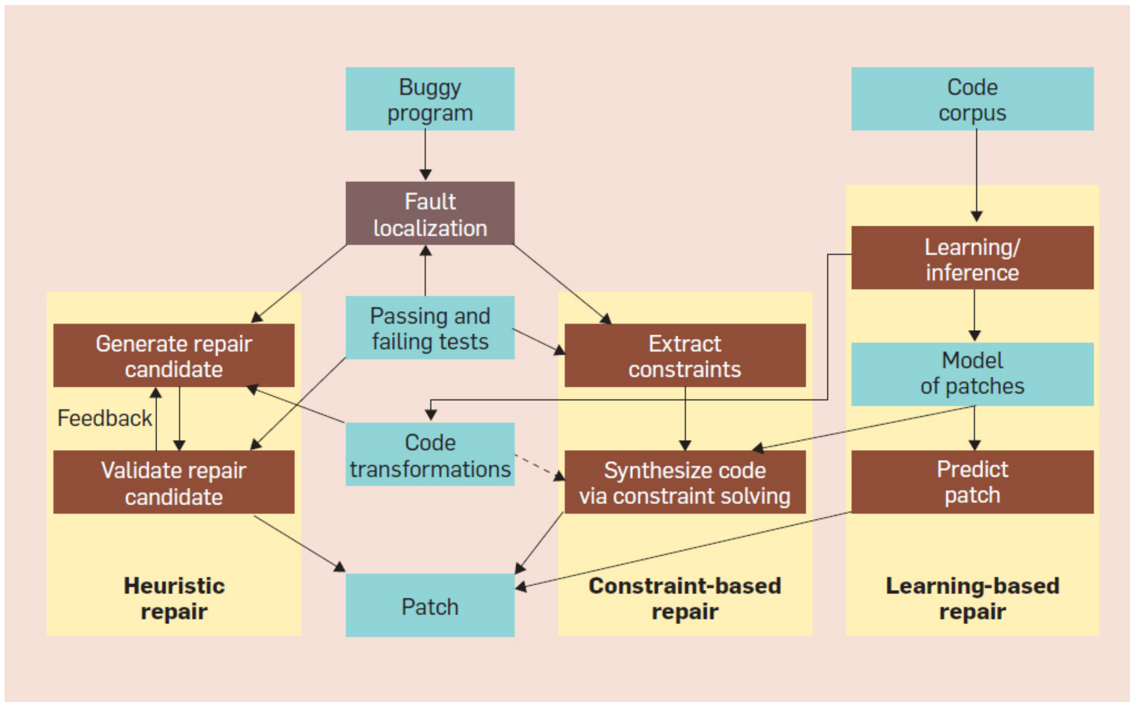


**Figure 2.1:** Overview of the different APR methods (Reprinted from [26])

statistics-based, program state-based, machine learning-based, data mining-based, model-based and miscellaneous techniques. Spectrum-based FL, which is inspired by statistical and probabilistic models, can be used to track program behavior and is typically used by APR techniques.

### 2.1.2 Heuristic Repair

Heuristic repair methods consist of creating a search space of candidate patches and iterating over them for validation. Patches are typically generated by transforming the program's abstract syntax tree (AST) using mutation operators [45, 6] (mutation-based APR) or templates, also known as fix patterns (template-based APR). Traversal over the search space can be done with meta-heuristics such as genetic programming [25], random sampling, or guided by heuristics [26].

### 2.1.3 Constraint-based Repair

Constraint-based methods consist of creating a set of constraints from a test suite that a correct patch should satisfy. A popular technique to extract properties that constitute a constraint is symbolic execution to find *angelic values* [15]. These constraints are solved to synthesize a patch, usually formulated as a Satisfiability Modulo Theory (SMT) problem [26].

### 2.1.4 Learning-based Repair

Recent advancements in machine learning, combined with the availability of a large number of patches as training data, enable the application of learning-based repair methods to enhance the

aforementioned APR methods [26]. Current applications include learning a model to rank a set of candidate patches [35], and inferring AST transformation templates from successful patches [33]. Some techniques learn models for end-to-end repair, bypassing the need for a correctness specification to validate patches [27, 49].

## 2.2 APR Benchmarking

Comparing and reproducing empirical research in APR is enabled with benchmarks comprised of datasets of faults extracted from practical programs. Such datasets include ManyBugs, IntroClass [24], and CodeFlaws [48] for the C language, as well as Defects4J for Java [30].

While these benchmarks allow for comparing different APR techniques, they do not necessarily represent the practical environment where APR tools operate. Just *et al.* [29] compared user-provided tests (i.e., tests found in bug reports, pre-fix) with developer-provided tests (i.e., tests committed alongside the fixes, post-fix). Their analysis suggests that developer-provided tests are more defect targeted (less code coverage) and have stronger assertions (more resilient to mutations). They measured the effect of each type of test in FL and APR techniques. With user-provided tests, FL techniques performed consistently worse than with developer-provided tests. With similar results, APR techniques using user-provided tests resulted in fewer generated patches, fewer correct patches, and increased repair times.

These results suggest that FL and APR benchmarks based on post-fix information overestimate their performance in practical use-cases where such information is unavailable. Furthermore, in some cases, the existing test-suite contradicts the defect-triggering test and needs fixing as well (is testing the implementation, not the specification). APR is infeasible for these defects unless techniques can identify invalid tests. The authors also inquired about the effect of test separation (creating a new small defect-triggering test vs. extending an existing test). They concluded that FL techniques performed better when triggering tests are separated.

## 2.3 Fuzzing

Fuzzing is a software testing technique effective at finding security vulnerabilities by monitoring program behavior when input with automatically generated payloads. Originally, a **fuzzing engine** consisted of feeding a target program with test cases created through random mutations. When the target program crashes or reports an error, the fuzzing engine collects and analyzes these issues to detect bugs and potential vulnerabilities [16]. This structure is presented in the blue-dashed frame of Figure 2.2.

Fuzzing engines have since evolved, using more advanced test case generation operations and introducing new techniques to improve fuzzing efficiency by feeding the test case generator with static and runtime information. Such techniques include symbolic execution [12, 14, 17, 23], coverage feedback [11, 10, 20, 2], and grammar representation [9]. The red-dashed frame of Figure 2.2 represents the current structure of fuzzing engines [16].

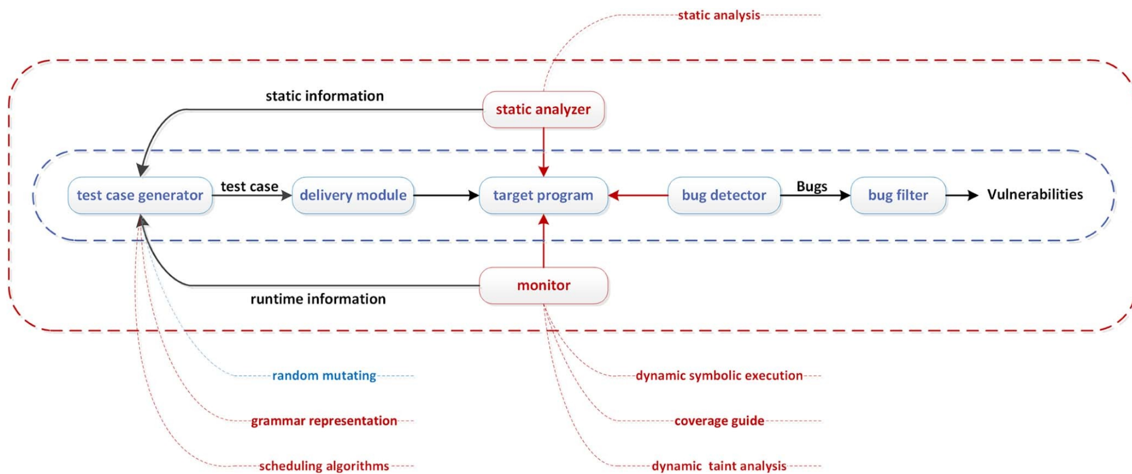


**Figure 2.2:** Architectural diagram of a fuzzing engine (Reprinted from [16])

Coverage-guided fuzzing tools such as AFL(American Fuzzy Lop)++ [20], and *LibFuzzer* [2] are currently being used to find vulnerabilities in key software libraries handling file formats, cryptography, and other operating system utilities. In addition, projects such as OSS-Fuzz [3] provide continuous fuzzing services for open-source software, providing a potential use-case for APR [26].

## 2.4 Summary

This chapter describes some fundamental concepts on APR and Fuzzing, essential for the further understanding of this dissertation. Section 2.1 introduces the definition and goals of APR and the concept of fault localization. Both the generate-and-test and constraint-based program repair approaches are explained, as well as how machine learning is being applied to augment these methods. Section 2.2 introduces the concept of benchmarks in APR, as well as some examples of benchmarks used to evaluate APR techniques, and explains how they may be misleading in practical use-cases. Finally, Section 2.3 introduces the concept of Fuzzing and mentions a potential use-case for APR.

# Chapter 3

# State of the Art


3.1 Survey Research Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.2 Procedure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19


This literature review aims to gather information on existing APR techniques and practical solutions that integrate APR with the SDLC during the software testing and integration step. We follow a structured methodology to ensure better integrity and results from this survey. This methodology consists of survey research questions to focus our analysis, defined in Section 3.1, and a procedure for the source, search, filtering and selection of publications, described in Section 3.2. In Section 3.3 we describe the results, followed by their categorization in Section 3.3.3, and their analysis in Section 3.3.4. Finally, Section 3.4 contains a summary of the literature review and final conclusions.

## 3.1 Survey Research Questions

In order to explore the current practices, research, and studies related to APR and its practical application, and identify recurring practices and challenges, we outline the following survey research questions (SRQs):

- **SRQ1** *What current, relevant open-source APR techniques exist?* APR is a topic with over a decade of research, with multiple proposals of different or enhanced techniques appearing over time. We are interested in the identification of up-to-date APR techniques with publically available source-code of the implementation.

- **SRQ2** *Which target languages are preferred by the techniques found in **SRQ1**?* As the APR implementation process is typically coupled with the language specification and compiler libraries, techniques can only support a subset of programming languages. Knowing

which languages were preferred by the techniques' developers is crucial for deciding how to integrate them with the SDLC.

- **SRQ3** *How fitting are the techniques found in* ***SRQ1*** *to be applied in practical large-scale projects?* Since APR research has only recently started focusing on practical applications, proposed APR techniques may not fit the requirements necessary to apply them in practical projects. Integration with the SDLC has certain performance and scalability requirements that APR techniques have to meet.

- **SRQ4** *What solutions exploring the application of APR in the software integration step of the SDLC exist?* To better understand how APR can be applied in practical domains and integrated with the SDLC during software integration, we need to identify current proposed approaches and attempts.

## 3.2 Procedure

1. Search for documents on the Scopus electronic database with *automated*, *program* and *repair* in their title, abstract or keywords.

2. Filter for Articles, Reviews, Conference Papers, and Conference Reviews, with a publication year between 2016 and 2021 inclusive.

3. Prioritize documents based on their relevance[1] and descending number of citations.

4. Analyze publications that describe at least one APR technique, using the following criteria to select relevant APR techniques:

   (a) Include reimplementations of a previous APR technique for a different target language (e.g., *jGenProg* [40], which is an implementation of *Genprog* [25] in Java).

   (b) Exclude APR techniques without a corresponding implementation.

   (c) Exclude APR techniques without publicly accessible source code of the implementation, or not used in a practical APR solution (e.g., *BovInspector* [21], a tool framework for the automated repair of buffer overflow vulnerabilities whose implementation is not available due to a broken link).

5. Analyze and select publications that explore the practical use of APR on the software integration step of the SDLC.

6. Use snowballing and reverse snowballing on analyzed publications to find more relevant publications to analyze and select (Through snowballing, it is possible to include relevant publications previous to 2016).

[1]Ranking based on Scopus' relevance ranking algorithm.

## 3.3 Results

Using the aforementioned methodology, we identify 14 APR techniques and 4 practical solutions for the integration of APR with the SDLC, described in Section 3.3.1 and Section 3.3.2 respectively. Section 3.3.3 categorizes our identified techniques and practical solutions so that in Section 3.3.4 we can analyze the results and present answers to the survey research questions.

### 3.3.1 APR Techniques

***GenProg* [25]** uses Genetic Programming, at the statement level of the AST, to find a program variant that does not suffer from a given defect while preserving functionality. *GenProg* uses statements from the program itself to produce a patched variant to reduce the search space and is therefore incapable of generating new code.

***SemFix* [44]** is an automated repair method based on symbolic execution, constraint solving, and program synthesis. It iterates each suspicious statement, in descending order, and attempts to generate a repair candidate by creating constraints out of the input tests and generating a repair constraint through symbolic execution.

***Angelix* [42]** is a scalable semantics-based APR tool capable of multi-location bug fixing, using the Jaccard [4, 5] formula for spectrum-based FL. It first extracts a specification in the form of an *Angelic Forest*, a set of *Angelic Paths* which encode constraints for a test-case based on the concept of *Angelic Value* [15]. Each *Angelic Path* is extracted using a custom symbolic execution algorithm that installs symbols in suspicious expressions. A patch is then synthesized based on this specification using Component-Based Repair Synthesis (CBRS) [28] and constraint solving. *SemFix*'s [44] algorithm was incorporated into *Angelix* and the authors claim to have successfully reproduced the experiments of the original *SemFix*.

***AllRepair* [47]** is a mutation-based APR tool to repair C programs, using a set of assertions as a formal bounded correctness specification rather than a set of tests. The tool ensures that every generated patch is minimal and bounded correct. Since it is based on formal methods and guarantees completeness, it does not use FL.

***Prophet* [36]** is a patch generation system using a probabilistic, application-independent model of correct code. With a set of successful human-made patches obtained from open-source projects, *Prophet* learns a probabilistic model to rank and identify correct patches within a generated search space of candidate patches.

***jGenProg* [40]** is one of the repair methodologies in Astor [40], an APR framework, and a Java adaptation of *GenProg* [25]. *JGenProg* uses Ochiai [4, 5] for FL, includes a single-change

search space navigation mode (without crossover), and offer three strategies for the scope of repair ingredients used in mutations (*Application*, *Package*, and *File* level).

***jKali* [40]** is one of the repair methodologies in Astor [40] and an implementation of Kali [46] in Java. Kali is a search-based APR technique focused on generating patches that remove functionality. It performs an exhaustive search over the repair space, navigating suspicious statements obtained from FL in descending order and applying to each the following code transformations: (i) remove statement; (ii) replace if condition with *true*; (iii) replace if condition with *false*; and (iv) insert return statement.

***jMutRepair* [40]** is one of the repair methodologies in Astor [40]. It is a Java implementation of a mutation-based APR technique presented by Debroy and Wong [18], applied on suspicious if statements. There are three classes of mutation operators: (i) relational, with six interchangeable equality and relational operators (i.e., replacement of an operator with one of the others); (ii) logical, with a swap of the *OR* and *AND* operators; and (iii) unary, with the addition or removal of the negation operator.

***Genesis* [34]** is a patch generation system that processes human-made patches to infer patch generation transforms automatically. Subsequently, *Genesis* infers the candidate patch search space generation driving the generate-and-test feedback loop based on these transforms.

***NPEFix* [19]** is an APR technique focused on null pointer exception fault repairs based on dynamic analysis and meta-programming. It generates a meta-program from the faulty program, by injecting hooks that detect *null* objects and apply one of nine strategies, allowing for patch search-space exploration at run-time. The authors also created a naive template-based implementation (exhaustive search of the patch space) with the same strategies called *TemplateNPE*, as a benchmark baseline for *NPEFix*. *NPEFix* produces more valid patches than *TemplateNPE* but with less performance. The authors reflect on this naive template-based implementation, and state that it could be a better solution than *NPEFix*, as it is much simpler to implement and maintain, while being faster and still offering good results.

***Nopol* [53]** is an APR tool for faulty conditional statements using dynamic analysis and constraint-based patch synthesis. It processes suspicious statements (if the statement is not conditional, then it assumes it is missing a precondition) in three stages: (i) angelic fix localization, which finds and localizes *angelic values*; (ii) runtime trace collection, which collects the values and variables available at each location during program execution; and (iii) patch synthesis, based on CBRS [28] and constraint-solving.

***Cardumen* [41]** is one of the repair methodologies in Astor [40]. It is a template-based APR technique with spectrum-based FL focused on maximizing the repair search space. After identifying potential modification points in suspicious statements, it parses the AST of the program under repair to extract templates. Once it creates a template pool, it uses probabilistic search methods to select templates to generate patches. *Cardumen* generated 8935 test-suite adequate patches over 77 bugs from Defects4J [30]. According to the authors, it is the largest number of automatically synthesized patches ever reported to this data-set, up until the publication date.

***DeepRepair* [51]** is one of the repair methodologies in Astor [40]. It is an extension of ***jGenProg*** using machine learning to prioritize and transform fix ingredients. It is comprised of three stages: (i) language recognition, which creates and normalizes code corpora at different levels of granularity (e.g., files, classes, methods, identifiers) based on the program AST; (ii) machine learning, which involves training a neural network language model, encoders of different granularity levels, and clustering identifiers' embeddings; and (iii) program repair, which consists of the *jGenProg* generate-and-test repair loop, using the trained models and clustering to infer code similarities that dictate how the patch search space is traversed according to one of five strategies for sorting and transforming fix ingredients.

***GetaFix* [8]** is a template-based APR tool that learns recurring fix patterns from static analysis warnings to quickly generate human-like fixes (similar performance as a static analysis tool). Given the imposed performance constraints, it only presents and validates a few configurable number of top-ranked fixes and uses a static analyzer instead of a test-suite for validation. A set of pairs of bugs and corresponding fixes are fed to *GetaFix* as training data. It uses a tree differencer to generate the change between before and after the fix and organizes fix patterns hierarchically using a hierarchical clustering algorithm. The tool then applies tree patterns to obtain a set of fix candidates, ranked using a statistical algorithm.

### 3.3.2 Practical APR Solutions

***Repairnator* [50]** is an end-to-end program repair bot. It consists of a three-stage pipeline: (i) Continuous Integration (CI) Build Analysis, which collects a set of suitable builds to repair; (ii) bug reproduction, which attempts to reproduce build failures locally; and (iii) patch synthesis, which feeds information from the fault reproduction to one of three APR tools (*NPEFix* [19], *Nopol* [53], and *Astor* [40]) to generate a patch. Analysts then review the produced patches manually. During the first 11 months of operation, the bot analyzed 11 523 test failures from 1609 open-source projects on GitHub and produced 15 patches, all suffering from overfitting. Despite its shortcomings, the bot has collected valuable data on the challenges of APR and its practical application. The authors share some insights on their experience designing *Repairnator* and propose actionable recommendations for future APR bots.

***iFixR* [31]** is a patch generation tool driven by bug reports. It disregards post-fix test-cases and works under the practical constraint that *when a bug report is submitted to the issue tracking system, a relevant test case reproducing the bug may not be readily available*. The buggy program and bug-reports are first fed to an Information retrieval (IR)-based fault localizer. From the top 20 suspected statements, patches are generated using template-based methods, by matching fix patterns with each suspicious statement's AST nodes, and validated using regression testing. The validated patches are then prioritized using heuristics so that the first patches being presented to the developer are more likely to be plausible or correct. The results of the authors' experiments suggest that *iFixR*, with the constraint of test-case unavailability when bugs are reported, achieves performance results comparable to state-of-the-art test-based APR techniques.

***SapFix* [39]** is an industrial scale automated end-to-end APR tool-set, from test design to fault reporting to developers, deployed on Facebook's CI system. It uses *Sapienz* [38], an intelligent continuous search-based software testing and test design tool, for identifying candidate crashes and FL, which then triggers up to four patch generation strategies: (i) `diff_revert`, where the whole changes are reverted; (ii) `partial_diff_revert`, where some of the changes are reverted, the faulty line, in particular; (iii) `template_fix`, generating template-based patches from *GetaFix* [8]; and (iv) `mutation_fix`, generating mutation-based patches. Candidate patches are verified by attempting to reproduce the crash with *Sapienz* and running the CI test-suite. Patches are selected for publishing based on heuristics such as strategy priority and information provided by *Sapienz*.

***Fix2Fit* [22]** is an integrated fuzzing and APR approach for detecting and discarding crashing candidate patches. While candidate patches are generated and validated against existing tests, new tests are generated to filter out overfitted patches. Following a grey-box fuzzing strategy, tests are prioritized based on how they separate patches, behaving equivalently with the current set of tests. The tool is integrated with OSS-Fuzz [3], which provides continuous fuzzing infrastructure to open-source projects, and it was evaluated on real-world vulnerabilities and projects.

### 3.3.3 Categorization

The mentioned **APR techniques** are categorized in Table 3.1 according to the following characteristics:

1. **Date** The publication date of the paper describing the APR techniques helps in analyzing the evolution of APR research over time.

2. **Language.** Since the APR implementation process is typically coupled with the language specification and compiler libraries, individual techniques only target a subset of programming languages, such as *C*, *C++* or *Java*.

**Table 3.1:** Categorization of APR techniques

| Solution | Date | Language | Scalability | Fault Localization | APR Method | Learning-based |
|---|---|---|---|---|---|---|
| *GenProg* [25] | 2012 | C | Medium | Spectrum-based | Genetic | No |
| *SemFix* [44] | 2013 | C | Medium | Spectrum-based | Constraint-based | No |
| *Angelix* [42] | 2016 | C | High | Spectrum-based | Constraint-based | No |
| *AllRepair* [47] | 2016 | C | Unknown | None | Mutation | No |
| *Prophet* [36] | 2016 | C | Medium | Spectrum-based | Mutation | Yes |
| *jGenProg* [40] | 2016 | Java | Medium[1] | Spectrum-based | Genetic | No |
| *jKali* [40] | 2016 | Java | Medium[1] | Spectrum-based | Mutation | No |
| *jMutRepair* [40] | 2016 | Java | Medium[1] | Spectrum-based | Mutation | No |
| *Genesis* [34] | 2017 | Java | Medium | Dynamic Analysis[2] | Templates[3] | Yes |
| *NPEFix* [19] | 2017 | Java | Medium | Dynamic Analysis[2] | Templates[4] | No |
| *Nopol* [53] | 2017 | Java | Medium[1] | Spectrum-based | Constraint-based | No |
| *Cardumen* [41] | 2018 | Java | Low | Spectrum-based | Templates | No |
| *DeepRepair* [51] | 2019 | Java | Unknown | Spectrum-based | Genetic | Yes |
| *GetAFix* [8] | 2019 | Java | High | Static Analysis | Templates | Yes |

[1] The paper mentions it is capable of generating patches for large-scale programs, but does not mention its performance.
[2] Uses stack trace logs to localize faults
[3] Uses the concept of transforms, which consist of a pair of template ASTs, before and after applying the patch
[4] Uses meta-programming to apply various templates at runtime.

3. **Scalability.** How well the technique scales with Lines Of Code (LOC) while remaining performant in practical use cases. The possible values are *Low*, *Medium*, and *High*.

4. **Fault Localization.** APR techniques can implement FL using different methods. Possible values are *Spectrum-based*, *Static Analysis*, *Dynamic Analysis*, and None.

5. **APR Method.** APR research contains a variety of methods to generate patches, as explained in Section 2.1. Possible values are *Templates*, *Mutation*, *Genetic*, and *Constraint-based*.

6. **Learning-based.** Defines if machine learning is used to augment the repair process. Possible values are *Yes*, and *No*.

The mentioned **practical APR solutions** are categorized in Table 3.2 according to the following characteristics:

1. **Fault Identification:** Practical APR tools need to identify faults to trigger APR processes continuously. This identification process requires some form of infrastructure or artifact to integrate with the SDLC. Values include *CI*, and *Bug Reports*.

2. **Fault Localization:** APR techniques used by the solution can implement FL using different methods. Possible values are *IR-based*, *Search-based*, *Dynamic* or *Multiple*, if the solution applies multiple techniques.

3. **APR Method:** APR research contains a variety of methods to generate patches, as explained in Section 2.1. Possible values are *Templates*, or *Multiple*, if the solution applies multiple techniques.

**Table 3.2:** Categorization of practical APR solutions within the SDLC software integration step

| Solution | Fault Identification[1] | Fault Localization | APR Method | Patch Verification | | Learning -based | Patch Selection[6] |
|---|---|---|---|---|---|---|---|
| | | | | Regression | Dynamic | | |
| *iFixR* [31] | Bug Reports | IR-based | Templates | Yes | No | No | Heuristic |
| *Repairnator* [50] | CI | Multiple[2] | Multiple[3] | Yes | No | No | Manual |
| *SapFix* [39] | CI | Dynamic[4] | Multiple[5] | Yes | Yes | Yes | Heuristic |
| *Fix2Fit* [22] | CF | Dynamic | Mutations | Yes | Yes | No | N/A |

[1] Continuous Integration (CI), Continuous Fuzzing (CF)
[2] *NPEFix* [19] uses dynamic analysis, *Nopol* [53] and Astor [40] use spectrum-based FL.
[3] *NPEFix* [19] and Astor [40] use heuristics-based methods, *Nopol* [53] uses constraint-based methods.
[4] Information about FL is vague, other than that it is performed by *Sapienz* [38], a search-based testing tool.
[5] Two heuristics-based patch generation methods: templates and mutation operators.
[6] Not Applicable (N/A)

4. **Regression Patch Verification:** Defines if generated patches are verified using regression testing. Regression testing verifies that positive tests keep passing after the changes (i.e., applying the generated patch). In addition, fault-triggering tests should pass to ensure the fault is fixed by the patch. Possible values are *Yes*, and *No*.

5. **Dynamic Patch Verification:** Defines if generated patches are verified with dynamic analysis. This patch verification method includes attempting to reproduce crashes or executing the patched program. It does not refer to using dynamic analysis to localize faults. Possible values are *Yes*, and *No*.

6. **Learning-based:** Defines if machine learning is used to augment the repair process. Possible values are *Yes*, and *No*.

7. **Patch Selection:** Verified patches have to be reported to developers to close the program repair cycle. A fault can result in many candidate patches, which may or may not be correct. Selecting the most plausible patches avoids spamming developers with suggestions and avoids suggesting incorrect or less desirable patches. Values include *Manual*, and *Heuristic*.

### 3.3.4 Analysis

This section describes the conclusions obtained by analyzing the techniques described in Section 3.3.1 based on the categories established in 3.3.3 and the Research Questions defined in 3.1.

#### 3.3.4.1 Categorical Analysis of APR Techniques

**Date** Between 2012 and 2016, APR research is focused on C, but eventually transitions to Java.

**Target Language** C and Java are the two dominant languages preferred by the research community when it comes to implementing APR techniques.

**Scalability** The majority of techniques have acceptable (medium) scalability, being capable of generating patches for large-scale projects. We can therefore conclude that APR techniques can be executed in industrial-scale software.

**Fault Localization** Spectrum-based FL is the most predominant, being used in 10 out of 13 test-based techniques. *AllRepair* [47] does not have fault localization. Two APR techniques use stack traces (Dynamic Analysis) to localize faults, while *GetAFix* [8] uses results from Static Analysis.

**APR Method** 10 out of 13 test-based techniques use a generate-and-test method rather than constraint-based methods. Within generate-and-test, there is no dominant traversal heuristic or patch generation method.

**Learning-based** The majority of tools do not rely on machine learning during the repair process. The tools that rely on it use it to augment specific steps in a *traditional* generate-and-test repair loop, rather than perform end-to-end repair.

#### 3.3.4.2 Categorical Analysis of practical APR solutions

**Fault Identification** Two of the implemented solutions use faults discovered during CI as the entry point for APR. *Fix2Fit* uses faults discovered by Continuous Fuzzing, while *iFixR* [31] suggests feeding bug reports to APR.

**Fault Localization** *Repairnator* [50] defers FL to the individual APR techniques it supports, while *SapFix* [39] defers FL to *Sapienz* [38]. *Fix2Fit* [22] gathers runtime information, by instrumenting the target program, to localize faults. *iFixR* [31] proposes information retrieval to localize faults based on the information present in bug reports.

**APR Methods** Generate-and-test is the predominant method, with *Nopol* [53] being the only APR technique using constraint-based methods. Within generate-and-test, there is no dominant traversal heuristic or patch generation method.

**Patch Verification** All solutions rerun tests to ensure the patch does not regress the program. In addition, *Fix2Fit* [22] generates new tests to filter out overfitting patches, and *SapFix* [31] attempts to reproduce crashes to verify the fault is no longer present.

**Learning-based** *SapFix* [31] is the only solution that integrates learning-based methods in its APR process.

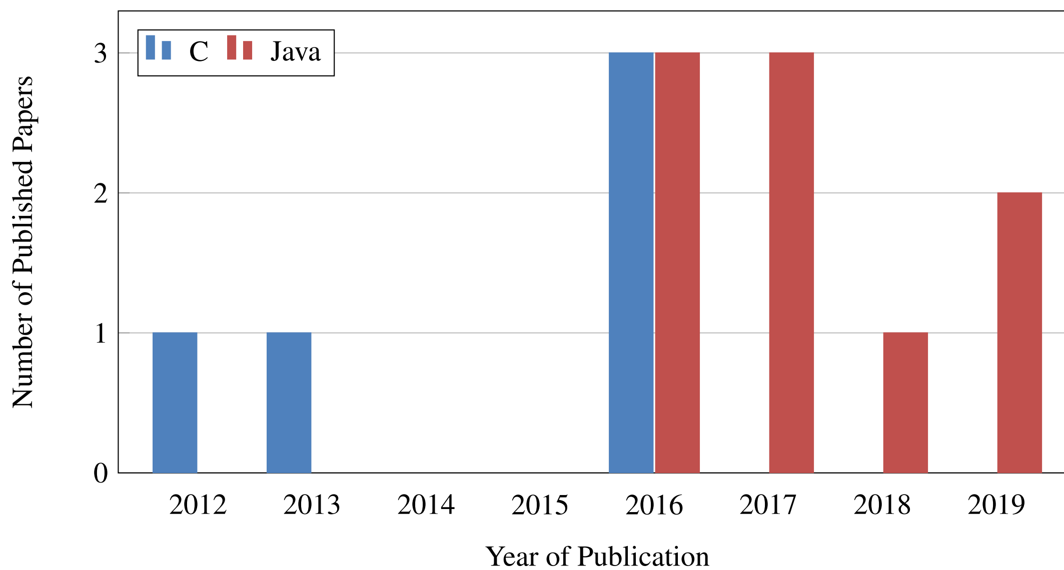


**Figure 3.1:** Evolution of Target Languages used by APR techniques

**Patch Selection** *Repairnator* [50] is the only solution which uses human analysts to filter patches before suggesting them to developers. *Fix2Fit* [22] does not fully integrate the SDLC loop, with no mention on how patches are selected and suggested to developers. The other solutions propose fully automated patch selection, using heuristics to prioritize them.

#### 3.3.4.3 Research Questions

The survey research questions presented in Section 3.1 guided the research of this literature review to obtain relevant answers regarding the current state of APR and its integration with the SDLC. We now revisit these questions and provide our findings:

- **SRQ1** *What current, relevant open-source APR techniques exist?* We have found 14 APR techniques, 13 of which have publicly available implementations, as described in Section 3.3.1 and presented in Table 3.1. Most tools do not rely on learning-based methods and use a wide range of methods and heuristics to generate or infer patches.

- **SRQ2** *Which target languages are preferred by the techniques found in **SRQ1**?* Table 3.1 provides an overview of the main characteristics of the found APR techniques. C and Java are the two languages targeted by these techniques. Comparing the paper publication dates with the target language, we notice that C was used in papers between 2012 and 2016, while Java was used from 2016 forward. Figure 3.1 shows this, suggesting that the research community started with C but eventually started using Java for their APR research.

- **SRQ3** *How fitting are the techniques found in **SRQ1** to be applied in practical large-scale projects?* Table 3.1 provides an overview of the main characteristics of the APR techniques found. Most of them claim to perform their function in large-scale projects. Among them,

two techniques stand out: (i) *Angelix*, which can repair faults in multiple locations, and (ii) *GetAFix*, whose authors claim to have performance equivalent to a static analysis tool. APR techniques with these characteristics are successfully applied in practical APR solutions, as described in Section 3.3.2. However, aside from *GetAFix*, which is integrated with *SapFix*, these APR techniques have not been validated in practical settings, relying on benchmarks and other controlled experiments to back their claims.

- **SRQ4** *What solutions exploring the application of APR in the software integration step of the SDLC exist?* We have identified four practical solutions, described in Section 3.3.2, that explore the integration of APR with the SDLC. Three entry points are explored: CI builds, continuous fuzzing, and Bug Reports. Two of these solutions complete the SDLC loop, with patches being suggested to developers by proposing a limited ranked set or manually filtering them previous to a pull request.

## 3.4 Summary

Section 3.1 specifies the research questions this literature review aims to answer, while Section 3.2 described the followed procedure. In Section 3.3 we describe the results, followed by their categorization in Section 3.3.3, and their analysis in Section 3.3.4.

The results show that APR research is not focused on practical applications, with most APR techniques being validated in controlled environments. There is little support for the practical application of APR, with only three solutions being implemented, two of them fully integrated into the SDLC, and one of them being proprietary. Chapter 4 goes into more detail on the research gaps briefly introduced here.

# Chapter 4

# Problem Statement


4.1 Research Gaps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.2 Hypothesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.3 Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.4 Research Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24


This chapter describes the problem tackled by this dissertation, presenting planned features, the hypothesis we aim to validate, and research questions to guide our dissertation. Section 4.1 mentions research gaps identified in the current state-of-the-art. Section 4.2 presents our hypothesis, and Section 4.3 defines the scope and objectives for the solution. Section 4.4 proposes research questions to guide development towards validating the hypothesis and proposed objectives. Finally, this chapter is summarized in section 4.5.

## 4.1 Research Gaps

In Chapter 3, we present a literature review where we identify 14 APR techniques. Besides *GetAFix* [8], whose implementation is not publicly available, neither of these techniques were validated in practical environments. To our knowledge, such techniques were not applied and evaluated in practical use-cases, like integrating with the SDLC, leading to a risk of them overfitting their respective benchmarks and experiments.

Supporting the claim that APR research is not focused in practical settings, is that, in the literature review, only four practical APR solutions proposing the integration of APR with the SDLC during software integration are identified: *Repairnator* [50], *iFixR* [31], *Fix2Fit* [22], and *SapFix* [39]. *iFixR* [31] is not evaluated in practical environments, and therefore lacks empirical support. *Fix2Fit* [22] does not fully integrate with the SDLC, as produced patches are not suggested to developers to complete the loop. *SapFix* [39] is heavily coupled with Facebook's toolchain, and

its implementation is not publicly available. With *Repairnator* [50], the following issues were identified:

1. **Effective coverage of faults:** To maximize the adoption and benefits of APR, it should cover a wide range of faults. Repairnator [50], in its first experiment, could only reproduce 30.82% of selected bugs.

2. **Effective correct patch generation:** Over the course of two experiments, Repairnator processed 6173 faults and, after manual filtering, drafted 12 correct patches [43]. This result, in our opinion, is an indicator that APR, and its practical application, is immature. Even considering the bottlenecks due to the fault identification, filtration, and reproduction processes, it is possible that APR techniques are underperforming in this more practical environment.

3. **Friendly patch suggestion:** From the 12 drafted patches, which were reported to developers, only 5 of them were accepted [43]. Repairnator reports patches by creating a plain pull request containing the patch without any further information pertaining the patch, the fault, or the underlying context. APR solutions integrated with the SDLC should be able to communicate information to its peers when suggesting patches. Interaction with the developers is a key component of the patch review and integration process.

From our analysis of the current state of APR, we are led to believe that its effectiveness when fully integrated in the SDLC is not clear since: (i) most APR techniques are validated in controlled environments, instead of practical ones; (ii) there are few proposals exploring the practical application of APR; and (iii) solutions which explore a complete end-to-end integration with the SDLC are close-sourced or have unsatisfying results.

## 4.2 Hypothesis

APR applied with continuous fuzzing, as a means of integration with the SDLC, shows potential due to fuzzing being a software testing technique for finding faults that (i) generates test-cases for the fuzzed target, which can also be fed into APR, and (ii) can be used within the APR workflow to improve the patch filtration and validation process, as performed by *Fix2Fit* [22]. Taking this into consideration, along with the gaps detected in APR research, this dissertation is built around the following hypothesis:

> *"Using fuzzing as a source of faults and of test cases allows for APR techniques to be effective when integrated with the SDLC."*

We define the effectiveness of an APR technique as the ability to generate at least one correct patch for a given fault. As a result, the effectiveness of an APR technique can be measured and compared against other techniques, for a given set of faults, by the percentage of faults with at least one correct patch generated.

This effectiveness does not correspond to the effectiveness of fully integrating APR with the SDLC as a whole, since that depends on more factors such as the effectiveness in capturing detected faults into APR, and the effectiveness in suggesting a generated correct patch to maintainers. While these aspects, necessary for APR integration, are tackled, the validation focus is on the APR techniques themselves within this practical context, since we believe that to be one of the gaps in APR research.

## 4.3 Scope

The goal of this dissertation is to implement a workflow leveraging coverage-guided fuzzing to generate fault-triggering test cases that are fed to an APR technique to generate a patch. Generated patches are then suggested to developers using pull requests managed by a GitHub Bot that interacts with developers using textual commands. The following aspects are **not** a part of this dissertation's scope:

1. **Exploring other fuzzing methods** such as blackbox and structure-aware fuzzing are not within the scope. However, using different coverage-guided fuzzing engines such as AFL++ [20] and *LibFuzzer* [2] are considered within the scope.

2. **Implementing a new APR technique** is out of the scope of this thesis. Since there is a decent range of APR techniques publicly available, for instance *Fix2Fit* [22] which was developed with fuzzing in mind and has been used in faults discovered by fuzzing, there is little motivation to implement a new APR technique. By using an existing technique, effort can be dedicated in the fuzzing and SDLC integration.

## 4.4 Research Questions

To guide the development of this dissertation towards the validation of the proposed hypothesis, we present the following research questions (RQs):

- **RQ1** *Are the selected APR techniques capable of generating correct patches when applied in faults discovered by fuzzing?* The effectiveness of APR techniques is determined by their ability to generate correct patches. If APR techniques cannot generate correct patches, then they are not effective.

- **RQ2** *What types of faults, in the context of fuzzing, are selected APR techniques more capable of generating correct patches?* Typically, the faults discovered by coverage-guided fuzzing are security vulnerabilties, in particular, memory corruption faults (e.g., buffer overflows, use after free, double free). Identifying the faults where APR techniques appear to more effectively repair may enable further research and discussion about why it is more effective, and how APR techniques can be adapted to better fit other fault types.

- **RQ3** *What are the main challenges in applying APR to projects within the context of fuzzing?* Coverage-guided fuzzing is typically leveraged by low-level libraries handling file formats, cryptography, and operating system utilities (e.g., OpenSSL[1], Libxml2[2], coreutils[3]). These projects are more environment-dependent, and it may be troublesome to integrate them with APR techniques. Such challenges directly affect the viability of APR and, as a result, its effectiveness.

## 4.5 Summary

Section 4.1 mentions some research gaps present in the current state-of-the-art of APR, particularly in its practical application. Section 4.2 presents our hypothesis, and how we measure APR effectiveness. Section 4.3 defines the scope and main goal, while Section 4.4 proposes research questions to guide development towards meeting the goal and validating the proposed hypothesis.

[1] OpenSSL: https://www.openssl.org/
[2] The XML C parser and toolkit of Gnome: http://xmlsoft.org/
[3] Coreutils - GNU core utilities: https://www.gnu.org/software/coreutils/

# Chapter 5

# Preliminary Work


| | | |
|---|---|---|
| 5.1 | Patch Suggestion through a GitHub Bot | 26 |
| 5.2 | Early Experiments with *Angelix* | 27 |
| 5.3 | Early Experiments with Coverage-guided Fuzzing | 30 |
| 5.4 | Overview | 33 |


With the goal of validating our hypothesis, defined in Section 4.2, we need first to investigate its feasibility. To this purpose, we divide the problem into a sequence of steps so that we can start our investigation by tackling each sub-problem independently. In order to do a full integration of APR with the SDLC, in particular with fuzzing, we identify the following steps:

1. **Continuous Fault Identification:** The first step in achieving full integration with the SDLC is to identify occurring faults and feed them to APR continuously. We want to leverage coverage-guided fuzzing to identify faults and collect generated test cases for the APR step, which requires experimenting with fuzzing engines, automating the fuzzing process, kick-starting the fuzzing process when a new software build is available (e.g., CI build), and collecting the data generated by fuzzing sessions to feed into the next step.

2. **Automated Program Repair:** This is the typical APR process of generating patches for a software fault, given the faulty project source and a set of tests as specification. While we will be using one or more existing APR techniques for this purpose, we need to experiment with selected techniques and automate the process of adapting and configuring each APR tool for the desired target project and set of test cases.

3. **Patch Suggestion:** When a patch is generated, it has to be proposed to developers so that it can be integrated with the corresponding project. Depending on the quality of patches produced by APR, we may need an intermediate step that filters patches unfit for integration. As we mention in Section 4.1, patch suggestion has to be interactive and facilitate the integration process. Implementing this step requires thinking about the suggestion process and about what functionality the developer needs to support it.

We start by exploring, in Section 5.1, the patch suggestion problem by describing the concept and initial development of a GitHub bot that automates the creation and management of pull requests with generated patches. Then, in Section 5.2, we describe some experiments we made using simple synthetic programs to get familiar with the usage of APR techniques. Next, in Section 5.3, we present our initial experimentation with fuzzing and start developing a solution to automatically fuzz projects. Finally, we present an overview of our findings in Section 5.4, which motivates us to halt further development and develop a study instead.

## 5.1 Patch Suggestion through a GitHub Bot

We start by tackling the issue of suggesting patches generated by APR to developers and integrating them in their respective projects. Typically, in the current state of the SDLC, software changes are integrated using pull requests. To this end, we decide to conceptualize and develop a bot capable of creating and maintaining pull requests in a version-control repository hosting service. We choose GitHub since (i) Git is one of the more popular version control systems, (ii) GitHub has a large open-source community, and (iii) GitHub allows for the creation of apps to integrate with GitHub[1].

In Section 5.1.1 we specify requirements that the GitHub bot needs to support to provide developer-friendly patch suggestion and integration services using pull requests. Then, in Section 5.1.2, we present our initial implementation of the bot before canceling further development, reasons for which are explained in Section 5.4.

### 5.1.1 Objectives

In order to suggest and facilitate the integration of patches to project maintainers using pull requests, the GitHub bot should be able to:

- **Conform to existing protocols for suggesting patches** such as following a pull request template and project-specific validation checklists before suggesting;
- **Assume responsibility for the suggested changes**, which corresponds to being capable of keeping a pull request updated with the merging branch without breaking its changes, and merging the pull request on demand;
- **Interact with peer developers to discuss and integrate suggestions**, which is the ability to request or bump reviewers, recognize and integrate external code review suggestions, and giving/requesting feedback to/from developers.

The requirements specified above can be represented as an activity diagram capturing the flow of a pull request from the bot's perspective, as depicted in Figure 5.1.

[1] About GitHub Apps: `https://docs.github.com/en/developers/apps/getting-started-with-apps/about-apps`

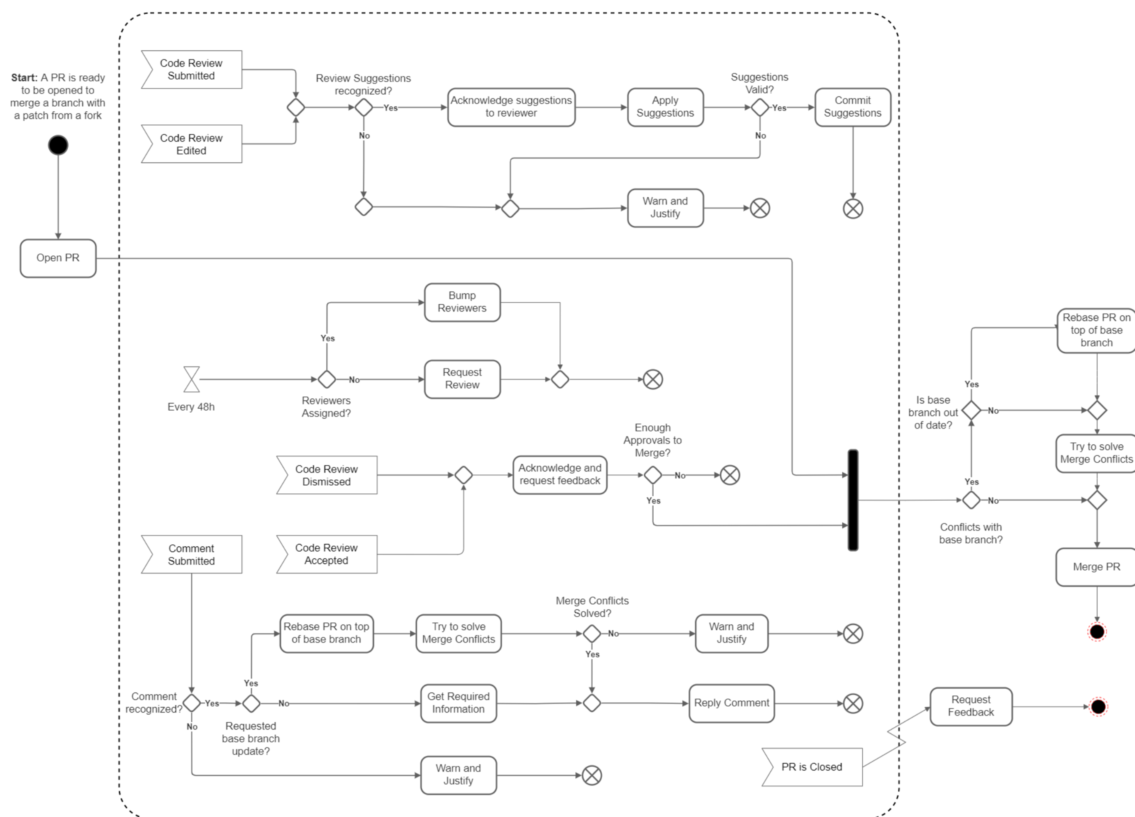


**Figure 5.1:** Activity diagram of a pull request workflow from the bot's perspective

### 5.1.2 Initial Implementation

We start implementing the bot in TypeScript using Probot[2], a framework for building GitHub Apps using Node.js[3]. Using Probot, we can easily interact with GitHub's API to manage Pull Requests. During an initial state of the implementation, the bot is capable of creating pull requests, processing pull request comments, and making a comment requesting feedback when a created pull request is closed.

While we were implementing the ability to rebase the pull request, which required keeping track of git repositories and interacting with git programatically (for this purpose, we used NodeGit[4], which provides Node.js bindings for libgit2[5]), developments in other sub-problems under exploration leads us to cancel further development, as explained in Section 5.4.

## 5.2 Early Experiments with *Angelix*

In order to gain experience on how to use APR techniques and evaluate their ability of being applied in projects to generate patches, we develop a set of simple programs in C (coverage-

[2]Probot - GitHub Apps to automate and improve your workflow: https://probot.github.io/
[3]Node.js - JavaScript runtime built on V8 engine: https://nodejs.org/en/
[4]NodeGit - Asynchronous native Node bindings for libgit2: https://www.nodegit.org/
[5]libgit2: https://libgit2.org/

```
@@ -73,4 -73,4 @@ main(int argc, char const *argv[])
     printf("Prove you are admin.\n");
-    fgets(input, 64, stdin);
+    fgets(input, 20, stdin);

     if (strcmp(input, check) == 0)
```

**Figure 5.2:** Patch diff file for the small program with a stack overflow vulnerability (see Appendix A.1)

guided fuzzing is mostly used in C/C++ projects) to try out an APR technique. We choose to experiment with *Angelix* [42], as it is the most recent APR technique and appears to have a more robust implementation from the set of APR techniques listed in Section 3.3.1 targetting C.

We start experimenting with *Angelix* on a synthetic program that can be exploited using stack-based buffer overflow (see Appendix A.1). This program contains the function `doAdminStuff` that should only be accessed with a credential. However, due to a fault in the `fgets` function call, this program is vulnerable to a controlled overwrite of the stack by at most 43 bytes. This vulnerability can be exploited to execute `doAdminStuff` without knowing the credential in two different ways:

1. Override the `check` buffer to represent the exact string as `input`, thus passing the string comparison in line 17. This exploit can be achieved using the payload generated by the following command as standard input: `python -c 'print "\x00" * 40'`.

2. Override the return address of `main` with the address of `doAdminStuff`. This requires first getting the address of the target function and the offset between the return address and the `input` buffer (stack pointer).

The fix for this fault is simple, it consists of changing the second argument of `fgets` from `64` to `20` (the size of `input`), as presented in Figure 5.2. Other partial fixes include adding boundary checks or using `strncmp`, but these fixes only prevent the first exploit from being successful, not the second one, as the overflow still occurs.

We attempt to run *Angelix* to fix this fault automatically for us. To this end, we first instrument the program by injecting macros specifying the output of the program, and its code reachability. Figure 5.3 presents a diff between the program and a instrumented version of it for *Angelix*. We decide that specifying reachability is sufficient for the purposes of our program, since the underlying fault relates to which parts of the code are reached, rather than what output is produced. Second, we need to setup a test oracle script (see Figure 5.4), for executing three tests to be fed into *Angelix*: (i) two positive tests representing the expected behavior when the right and wrong credential is input, and (ii) a negative test exploiting the stack buffer overflow vulnerability. Finally, we need to specify the correct reachability of the negative test, so *Angelix* knows the correct

```
@@ -2,9 +2,15 @@
 #include <string.h>

+#ifndef ANGELIX_OUTPUT
+#define ANGELIX_OUTPUT(type, expr, label) expr
+#define ANGELIX_REACHABLE(label)
+#endif
+
 void doAdminStuff()
 {
   printf("Here's the special cookie for admins only.\n");
+  ANGELIX_REACHABLE("admin");
 }

 int main(int argc, char const *argv[])
@@ -18,6 +24,7 @@ main(int argc, char const *argv[])
     doAdminStuff();
   } else {
     printf("Not admin, go away.\n");
+    ANGELIX_REACHABLE("notadmin");
   }

   return 0;
```

**Figure 5.3:** Diff file between the small program with a stack overflow vulnerability (see Appendix A.1) and its instrumented version

```
#!/bin/bash

case "$1" in
    admin)
        echo "trustmeimadmin" | $ANGELIX_RUN ./vuln1 \
            | grep "Here's the special cookie for admins only." &>/dev/null
        ;;
    notadmin)
        echo "notadmin" | $ANGELIX_RUN ./vuln1 \
            | grep "Not admin, go away." &>/dev/null
        ;;
    shouldnotbeadmin)
        python -c 'print "\x00" * 40' | $ANGELIX_RUN ./vuln1 \
            | grep "Not admin, go away." &>/dev/null
        ;;
esac
```

**Figure 5.4:** Test oracle script with two positive tests and one negative test

```
      american fuzzy lop ++2.66d (test-floatingpoint) [explore] {0}
- process timing ---------------------------------- overall results -
        run time : 0 days, 0 hrs, 0 min, 49 sec    cycles done : 125
   last new path : 0 days, 0 hrs, 0 min, 32 sec    total paths : 6
 last uniq crash : 0 days, 0 hrs, 0 min, 32 sec   uniq crashes : 1
  last uniq hang : none seen yet                    uniq hangs : 0
- cycle progress ----------------- map coverage -----------------------
  now processing : 0.125 (0.0%)     map density : 28.12% / 50.00%
 paths timed out : 0 (0.00%)     count coverage : 1.00 bits/tuple
- stage progress ----------------- findings in depth ------------------
  now trying : splice 5          favored paths : 6 (100.00%)
 stage execs : 31/32 (96.88%)     new edges on : 6 (100.00%)
 total execs : 592k              total crashes : 8 (1 unique)
  exec speed : 11.2k/sec          total tmouts : 0 (0 unique)
- fuzzing strategy yields ------------------------ path geometry ------
   bit flips : 0/184, 0/178, 0/166                  levels : 4
  byte flips : 1/23, 0/17, 0/5                     pending : 0
 arithmetics : 0/1283, 0/471, 0/33                pend fav : 0
  known ints : 0/121, 0/417, 0/218               own finds : 5
  dictionary : 0/0, 0/0, 0/0                      imported : n/a
havoc/splice : 3/228k, 2/360k                    stability : 100.00%
   py/custom : 0/0, 0/0
        trim : n/a, 0.00%                                [cpu000: 50%]
```

**Figure 5.5:** Example of the AFL++ status screen (Reprinted from [1])

behavior. Running *Angelix* with this configuration results in no patch being generated, with FL not locating any suspicious statements. We tuned the program and *Angelix* command line options to try to obtain a better result, however we were never able to produce a correct patch.

*Angelix*'s inability to generate a patch for this scenario led us to start doubting of current APR techniques being able to be applied in practice, which eventually motivated us to cancel the development of a practical APR workflow integrated with the SDLC, as mentioned in Section 5.4.

## 5.3 Early Experiments with Coverage-guided Fuzzing

To become familiar with coverage-guided fuzzing and the process of identifying faults through fuzzing, one of the identified steps in integrating APR with the SDLC, we decide first to select a popular coverage-guided fuzzing engine and, in Section 5.3.1, experiment it with small synthetic programs. Then, in Section 5.3.2, we describe our initial planning and development of an automated fuzzing solution.

### 5.3.1 Fuzzing with AFL++

AFL++ [1] is an improved fork of AFL [54] providing better performance, more mutators, and better instrumentation. We chose this fuzzing engine due to its effectiveness and ease of use. AFL++ presents a user interface providing information about the status of the ongoing fuzzing session (Figure 5.5), and fuzzing sessions can be paused and resumed freely. We start by selecting a set of three target programs (see Appendix A), each suffering from a specific security weakness, to experiment with AFL++ and generate a test case triggering the fault for each target:

- **Stack Buffer Overflow:** Program suffering from a stack-based buffer overflow vulnerability (described in Section 5.2).
- **Divide by Zero:** Program that calculates the float division of two numbers. Division by zero leads to undefined behavior.
- **Format String:** Program that prints a user-controlled format string, which can be exploited to read or overwrite arbitrary memory[6].

Each program (i.e., fuzz target) is then instrumented for fuzzing, including a sanitizer that crashes the program when a fault is detected (e.g., an AddressSanitizer[7] detects memory errors). Then, the fuzz target is executed with a command-line utility (`afl-fuzz`) to start a fuzzing session. This configuration process is better described in Section 6.2.4, as part of a description of the planning and development of a combined fuzzing–APR process.

For all three programs, we are able to initiate a fuzzing session and generate a fault-triggering input after a short time. As expected, these payloads expose the fault but do not exploit it. For example, in the program with a stack buffer overflow weakness (Appendix A.1), the generated payload does overwrite the buffer but it does not bypass the credential check, which means that fuzzing can be used to generate a negative test to feed into APR, but the oracle has to be based on whether the fault is triggered or not (i.e., the sanitizer detects the fault), rather than program behavior/output.

### 5.3.2 Automating the Fuzzing Process

Following our success in fuzzing programs manually, we start planning and developing an automated workflow for triggering a fuzzing process and capturing fault-triggering test cases. We focus on automating fuzzing with AFL++ but design the system to be extensible and support multiple fuzzing engines and target languages. Additionally, the fuzzing process will have to be integrated with APR and the GitHub bot described in Section 5.1 eventually. With these requirements in mind, we develop an independent service responsible for the fuzzing process, listening for messages from a broker to start and stop fuzzing sessions.

As displayed in Figure 5.6, to start a fuzzing session, we provide the fuzzing service with the fuzz target's root source folder and executable paths, and with the language and build system it uses. There can only be a single active fuzzing session for a given project (root source folder), so a message with this field can be used to identify which fuzzing session to stop (see Figure 5.7). Figure 5.8 shows a class diagram of the fuzzing service. The fuzzing configuration and execution process is deferred to the `Engine` class, which is selected based on the target language by the `EngineFactory` class (we only support AFL++ at the moment). Additionally, there is a `CrashMonitor` responsible for monitoring the file system for new fault-triggering test cases generated during the fuzzing session.

[6]Exploiting Format String Vulnerabilities: https://cs155.stanford.edu/papers/formatstring-1.2.pdf
[7]AddressSanitizer Documentation: https://clang.llvm.org/docs/AddressSanitizer.html

```
{
    "fuzz_target": "project-folder/",
    "fuzz_binary": "fuzzer",
    "language": "C/C++",
    "build_system": "makefile"
}
```

**Figure 5.6:** Example of a message to start a new fuzzing session

```
{
    "fuzz_target": "project-folder/"
}
```

**Figure 5.7:** Example of a message to stop an ongoing fuzzing session

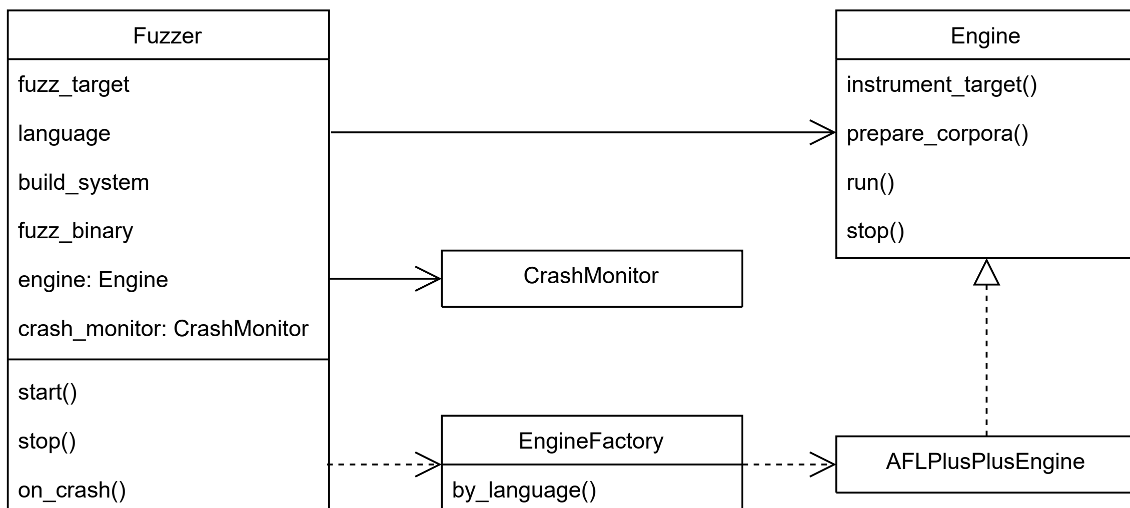


**Figure 5.8:** Class diagram of the fuzzing service

During this stage of the implementation, we are able to start a fuzzing session and monitor generated fault-triggering test cases. We repeat the experiments described in Section 5.2, but this time using the tests generated by fuzzing instead of manual tests. The results are similarly unsatisfactory, motivating us to stop development as mentioned in Section 5.4.

## 5.4 Overview

In Section 5.1 we describe the development of a GitHub bot for suggesting patches. Meanwhile, we perform APR experiments using *Angelix*, described in Section 5.2. Finally, in Section 5.3, we describe our initial experiments with AFL++ and the automation of the fuzzing process. Unfortunately, the results of our APR experiments indicate that current APR techniques are not suitable to be applied with fuzzing in practical use cases, meaning we would have to develop our own APR technique to perform the intended APR integration with fuzzing and the SDLC. Since this development is a bit out of scope and, more importantly, in our opinion, cannot be achieved within the timeframe of this dissertation, we opt for stopping current development and start an empirical study, described in Chapter 6, to better evaluate the current effectiveness of APR techniques when integrated with fuzzing and applied in more practical scenarios.

# Chapter 6

# Empirical Study


6.1 Objectives . . . 35
6.2 Planning . . . 36
6.3 Results . . . 46
6.4 Replication of Original APR Experiments . . . 49
6.5 Threats to Validity . . . 50
6.6 Discussion . . . 53
6.7 Summary . . . 55


In Chapter 5 we present our initial planning and development of an integrated fuzzing and APR solution, as well as our initial experimentation with coverage-guided fuzzing and APR techniques. From the results of our experiments with APR, we realize that these APR techniques are not currently viable enough to be applied in practical use-cases, so we decide to, instead of implementing a solution, make an empirical study about APR effectiveness when applied with fuzzing. In Section 6.1 we present our motivation and goals with the empirical study, followed by Section 6.2 which describes how this study is planned out and the experimentation process. The results of our experiments are presented in Section 6.3, and in Section 6.4 we present the results of our replication of *Angelix*'s original experiments [42]. Section 6.5 presents a list of identified threats towards the validity of our results. Finally, Section 6.6 discusses and analyzes the results in the context of validating/invalidating our hypothesis and answering the research questions proposed in Chapter 4.

## 6.1 Objectives

The initial experimentation of APR techniques on synthesized targets, described in Section 5.2, revealed a lack of effectiveness when applied to simple, synthesized scenarios. This observation brought up some cynicism on the capabilities of APR techniques, motivating more thorough experimentation to assess APR effectiveness when applied to faults discovered by fuzzing. Towards

this end, we conceptualize a fuzzing-powered APR process, where projects are fuzzed to reproduce a fault, and an APR technique is then run to generate a patch. A key rationale behind this process is that it can be then applied in practice, so that experiments based on it capture the effectiveness of APR techniques as if they were integrated with the SDLC. Adapting the process for a full SDLC integration would be achievable, for example, by kickstarting the process with new integration builds and suggesting generated patches to developers via pull requests; or by leveraging existing continuous fuzzing jobs to replace the fuzzing step, and link generated patches to the corresponding fuzz issue mentioning the identified fault.

With the goal of evaluating the effectiveness of APR techniques, we apply this process to a matrix of programs and APR techniques. This process is applied twice, one where generated test cases are fed to APR, and another one where generated test cases and the project's tests are fed to APR. The set of programs each reproduce a security fault and are further categorized as synthetic or real. Synthetic programs (see Appendix A) are simple, custom-made fuzz targets with the purpose of expressing and capturing a common security weakness (e.g., stack-based buffer overflow, externally-controlled format string, divide by zero). Real programs are source-code of specific releases of software libraries, coupled with a fuzz target, that together allows for the reproduction of a known security vulnerability (e.g., Heartbleed[1]). The set of APR techniques are selected from the APR techniques and solutions identified in Section 3.3. Applying this process to each unique program–APR technique pair in the matrix allows for evaluating and comparing different techniques over different types of faults, helpful to test our hypothesis and answer the research questions defined in Chapter 4.

## 6.2 Planning

Creating a process combining fuzzing and APR similar to a practical environment requires careful planning to ensure the results are unbiased and best represent the capabilities of APR techniques when integrated with the SDLC. Section 6.2.1 presents an high-level overview of this process, explaining both its components and the software modules in each. Section 6.2.2 goes into detail on the security vulnerabilities selected for the study, while Section 6.2.3 specifies the selection process of APR techniques for the study. Section 6.2.4 explains how the fuzzing engines are configured, and Section 6.2.5 explains how the APR techniques are setup and configured for the purposes of this experimentation.

### 6.2.1 Joint Fuzzing–APR Process

For each vulnerability and APR technique, we produce patches following a process comprising two components in a pipeline with a fuzzing step followed by an APR step. The fuzzing step lasts until a fault-triggering test case is generated. This test case, along with all the test cases generated in the fuzzing step, are then fed to an APR technique. Figure 6.1 presents an high-level overview

[1] The Heartbleed Bug: https://heartbleed.com/

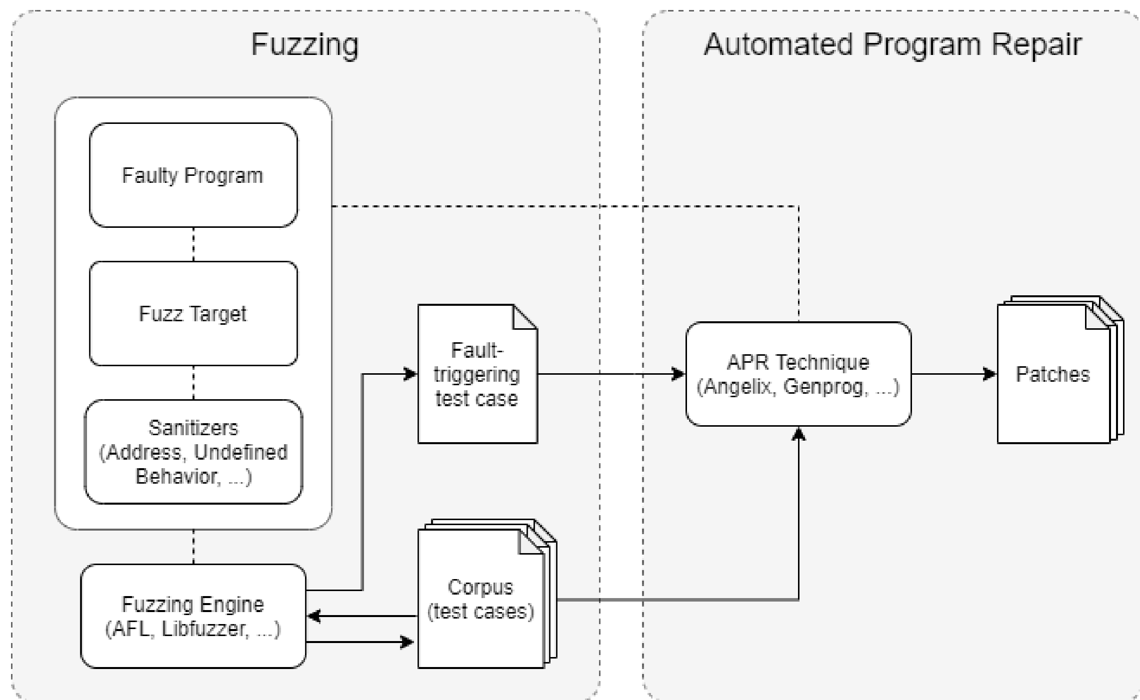


**Figure 6.1:** High-level architecture of the fuzzing–APR process

of this process. The fuzzing step of the process is the same and only performed once for each set of projects/faults. The APR step of the process is done once for each APR technique, using the gathered fuzzing information of the respective project/fault. Arrow lines represent data flows (inputs and outputs) between a software module and an artifact, while dashed lines represent the linking of software modules. Each software module is described as follows:

**Fuzz Target and Faulty Program:** The fuzz target serves as an entry point for the faulty program. The fuzz target may be the program under test itself, but, since most programs are software libraries, it usually contains a series of function calls to test certain behavior. As a result, it is common for a project to maintain multiple fuzz targets. For our study, we use a single fuzz target, the one that reproduces the desired fault.

**Sanitizers:** Sanitizers are compiler instrumentation modules that allow for the detection of faults during runtime. For fuzzing purposes, popular sanitizers are the AddressSanitizer[2], capable of detecting memory and addressing issues such as out-of-bounds access to the heap or stack, and the UndefinedBehaviorSanitizer[3] for detecting null pointer, integer overflow, and type conversion issues.

**Fuzzing Engine:** The fuzzing engine is responsible for handling the fuzzing feedback loop. The fuzz target and faulty program are first instrumented with fuzzer instrumentation and sanitizers to

[2] AddressSanitizer: https://clang.llvm.org/docs/AddressSanitizer.html
[3] UndefinedBehaviorSanitizer: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

**Table 6.1:** List of synthetic programs based on their vulnerability types

| Program Description | Vulnerability Type | CWE-ID | Exploitation Impact |
|---|---|---|---|
| Restricted Access | Stack Overflow | CWE-121 | Unauthorized access to resource |
| Brainfuck Interpreter | Heap Overflow | CWE-122 | Read out-of-bounds memory |
| Division Operation | Divide by Zero | CWE-369 | Unexpected output |
| String Customizer | Format String | CWE-134 | Read out-of-bounds memory |
| Useless Computation | Integer Overflow | CWE-190 | Infinite Loop |

allow the fuzzing engine to control and monitor its execution. An initial corpus (set of test cases) is initially fed to the fuzzing engine, and, during execution, generated test cases that cover new paths are added to the corpus. When an execution crashes, typically due to a fault detected by a sanitizer, the fuzzing engine reports back the fault-triggering test case.

**APR Technique:** The APR technique generates patches based on an oracle and a set of test cases. In regards to the oracle, we typically use the exit code of the executable (i.e., the test passes if the exit code is zero). In some synthetic programs, we match the standard output against an expected substring.

### 6.2.2 Project and Fault Selection

As mentioned in Section 6.1, there are two types of projects/faults: (i) synthetic programs with security weaknesses and (ii) real-life projects with known vulnerabilities. Including both these kinds of projects allows us to evaluate the effectiveness of APR in both simple and complex industrial programs. For the synthetic programs, we use the programs described in Appendix A. The list of synthetic programs is presented in Table 6.1. Each program instantiates a unique security weakness, as defined in the Common Weakness Enumeration (CWE) list[4]. The program description and vulnerability exploitation impact description provide some context on the synthesized programs. Some vulnerabilities have a direct security impact, while others facilitate further exploits.

**Table 6.2:** List of real-life vulnerabilities and respective projects

| Vulnerability Type | Project | CVE-ID | NVD CVSS v3 Base Score[1] |
|---|---|---|---|
| Heartbleed | OpenSSL 1.0.1f | CVE-2014-0160 | 7.5 (High) |
| Stack Overflow | GnuTLS 3.5.7 | CVE-2017-5336 | 9.8 (Critical) |
| Use After Free | LibPNG 1.6.36 | CVE-2019-7317 | 5.3 (Medium) |
| Heap Overflow | FFmpeg 4.2.2 | CVE-2020-12284 | 9.8 (Critical) |

[1] National Vulnerability Database (NVD) Common Vulnerability Scoring System (CVSS) Version 3: https://nvd.nist.gov/vuln-metrics/cvss

[4] CWE List Version 4.4: https://cwe.mitre.org/data/

For the real-life projects, one of the vulnerabilities we include is the Heartbleed bug[5] as it is reported to have been automatically repaired by *Angelix* [42]. Heartbleed is a vulnerability present in early versions of OpenSSL 1.0.1. The fault is present in OpenSSL's implementation of the Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) heartbeat extension (Request For Comments [RFC] 6520[6]). It can be exploited to leak memory contents from the server to the client or from the client to the server, is classified as a buffer over-read of the heap, and can be patched by adding a bounds check. Figure 6.2 shows a diff of the patch made by the maintainers of OpenSSL for TLS[7]. Figure 6.3 shows the patch generated by *Angelix*, as claimed by the authors [42]. Both patches are semantically equivalent, and the *Angelix* patch is an example of a patch that would be considered correct for the objectives of this study. We select more known vulnerabilities from a list of all fixed issues in OSS-Fuzz[8]. By using faults discovered in OSS-Fuzz, we can find the faulty project version, the fuzz target and configuration (more details in Section 6.2.4), and the patch produced by developers. The full list, presented in Table 6.2, is a result of a selection of OSS-Fuzz issues matching the following criteria:

1. The project's source code is publicly available and written in the **C language**.
2. The issue has been assigned a record in the **Common Vulnerability Enumeration** (CVE)[9].
3. The instantiation of the vulnerability identified in the issue has to be **repairable in a single code location**. The vulnerability itself may be instantiated in multiple locations (e.g., Heartbleed is instantiated both in TLS and in DTLS), but the instantiation identified by the fuzzing job described in the issue cannot require multi-location repair.

### 6.2.3 APR Technique Selection

In Section 3.3.1 we identify APR techniques with publicly available source code of the implementation, allowing them to be experimented with and, potentially, applied in practical applications. We select the test-based tools targeting projects in C from this set of techniques, resulting in the list of APR techniques presented in Table 6.3.

*Fix2Fit* [22], which integrates fuzzing in the APR process and is built on top of OSS-Fuzz [3], was one of the prime candidates for our study. However, we were unable to configure and run the tool properly. Initially, we had issues going around old and deprecated OSS-Fuzz infrastructure to build the tool, which we overcame (the infrastructure was updated by the authors eventually, too). However, due to incorrect instructions for running the tool, we do not know how to run it. As a result, we do not know if errors brought up during execution attempts are due to a fault in the tool, improper build/configuration, or because the executed script is not the right entry point.

[5] The Heartbleed Bug: https://heartbleed.com/

[6] RFC6520: https://datatracker.ietf.org/doc/html/rfc6520

[7] OpenSSL commit diff with the Heartbleed patch: https://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=96db9023b881d7cd9f379b0c154650d6c108e9a3;

[8] List of verified (closed) issues in OSS-Fuzz: https://bugs.chromium.org/p/oss-fuzz/issues/list?q=status%3AVerified&can=1

[9] CVE List: https://cve.mitre.org/cve/

```
1  --- a/ssl/t1_lib.c
2  +++ b/ssl/t1_lib.c
3  @@ -2588,16 +2588,20 @@ tls1_process_heartbeat(SSL *s)
4          unsigned int payload;
5          unsigned int padding = 16; /* Use minimum padding */
6  
7  -       /* Read type and payload length first */
8  -       hbtype = *p++;
9  -       n2s(p, payload);
10 -       pl = p;
11 -
12         if (s->msg_callback)
13                 s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
14                         &s->s3->rrec.data[0], s->s3->rrec.length,
15                         s, s->msg_callback_arg);
16 
17 +       /* Read type and payload length first */
18 +       if (1 + 2 + 16 > s->s3->rrec.length)
19 +               return 0; /* silently discard */
20 +       hbtype = *p++;
21 +       n2s(p, payload);
22 +       if (1 + 2 + payload + 16 > s->s3->rrec.length)
23 +               return 0; /* silently discard per RFC 6520 sec. 4 */
24 +       pl = p;
25 +
26         if (hbtype == TLS1_HB_REQUEST)
27                 {
28                 unsigned char *buffer, *bp;
```

**Figure 6.2:** Official Heartbleed patch diff for the TLS implementation of OpenSSL

```
1  --- a/ssl/t1_lib.c
2  +++ b/ssl/t1_lib.c
3  @@ -2596,7 +2596,7 @@ tls1_process_heartbeat(SSL *s)
4          if (s->msg_callback)
5              s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
6                      &s->s3->rrec.data[0], s->s3->rrec.length,
7                      s, s->msg_callback_arg);
8  -       if (hbtype == TLS1_HB_REQUEST)
9  -               {
10 +       if (hbtype == TLS1_HB_REQUEST
11 +           && payload + 18 < s->s3->rrec.length) {
12                 unsigned char *buffer, *bp;
```

**Figure 6.3:** *Angelix*'s Heartbleed Patch diff for the TLS implementation of OpenSSL

**Table 6.3:** List of selected APR techniques, and their primary attributes

| Solution | Target Language | Fault Localization | APR Method | Learning Based |
|---|---|---|---|---|
| *GenProg* [25] | C | Spectrum-based | Genetic | |
| *SemFix* [44] | C | Spectrum-based | Constraint-based | |
| *Angelix* [42] | C | Spectrum-based | Constraint-based | |
| *Prophet* [36] | C | Spectrum-based | Mutation | Patch Ranking |

We opened an issue[10] to explain our situation to the authors, but we were unable to include this technique in our study.

### 6.2.4 Fuzzing Configuration

The fuzzing step of the process is done once for each vulnerability, until a triggering test case is found. We use two coverage-guided fuzzing engines, AFL++ [20] and *Libfuzzer* [2]. Despite the differences in both implementations, their configuration comprises of three steps: (i) environment setup, (ii) program instrumentation, and (iii) the fuzzing process.

**Environment Setup** Some projects have specific dependencies that have to be installed first (e.g., GnuTLS depends on Nettle, a cryptographic library). In most cases, these dependencies can be installed using the operating system's native package manager, but when an older version is required, they have to be built and installed from the source. As for the fuzzing engines, *Libfuzzer* is included in Clang version 6.0 and above[2] and there is a docker image available for AFL++[11]. Table 6.4 provides some information on the environment of each fuzzing engine.

**Program Instrumentation** In *libFuzzer*, a fuzz target is a C function with a reserved name that typically receives bytes as input (test case) and does something in the library under test with it, as shown in Figure 6.4. Compiling and linking the target with clang using the `-fsanitize=fuzzer` flag will do the necessary instrumentation and link it with the *libFuzzer* library. The target can also be built with sanitizers by adding them to the `-fsanitize` flag (e.g., "`clang -g -O1 -fsanitize=fuzzer,address fuzz_target.cc`" builds the fuzz target with AddressSanitizer) [2].

**Table 6.4:** Fuzzing engine environment information

| Environment | Operating System | Clang Version |
|---|---|---|
| *LibFuzzer* | Kali Linux 2020.2 - WSL2 | 11.0.1-2 |
| AFL++ | Ubuntu 20.04.2 LTS | 12.0.1[1] |

[1] Version of `afl-clang-fast` and `afl-clang-fast++`.

[10] *Fix2Fit* issue: https://github.com/gaoxiang9430/Fix2Fit/issues/3
[11] AFL++ docker image: https://hub.docker.com/r/aflplusplus/aflplusplus

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;
}
```

**Figure 6.4:** Example of a fuzz target in *libFuzzer* (Extracted from [2])

AFL++ [20] supports *libFuzzer*'s fuzz target format, but it also supports a regular program with a `main` function as the entry point. The target is instrumented using one of the available AFL++ compilers. The most recommended compiler is with LLVM Link Time Optimization (LTO) (`afl-clang-lto/afl-clang-lto++`), claimed to be faster and providing the best coverage for AFL++. However, this compiler requires clang version 11 or above, which was released in 2020 and may not be available in all of the APR techniques' environments. To avoid not being able to reproduce the identified fault in the APR environment due to using an older compiler version, and consequently an older sanitizer version, we decide to use their second most recommended compiler (`afl-clang-fast/afl-clang-fast++`). Sanitizers can be enabled by setting environment variables when compiling the target (e.g., `AFL_USE_ASAN=1` will enable the AddressSanitizer) [1].

Some of the selected faults require the corresponding sanitizer to be configured with specific options. In both fuzzing engines, this can be configured by setting the `ASAN_OPTIONS` environment variable when running the corresponding fuzzer. Since the APR techniques do not support C++, and most fuzz targets used by selected projects are in C++, we have to convert these fuzz targets to C.

For each individual program, we use a specific fuzzing engine and a single sanitizer that detects the corresponding fault:

- **AFL++ with AddressSanitizer:** CWE-121 (Stack Buffer Overflow), CWE-122 (Heap Buffer Overflow), CWE-134 (Format String), CVE-2014-0160 (Heartbleed), and CVE-2017-5336
- **AFL++ with UndefinedBehaviorSanitizer:** CWE-369 (Divide by Zero), and CWE-190 (Integer Overflow)
- ***LibFuzzer* with AddressSanitizer:** CVE-2019-7317, and CVE-2020-12284

**The Fuzzing Process** The fuzzing process requires first creating a corpus directory with several test cases, each in its own file. *Libfuzzer* allows for an empty corpus directory, but this behavior can be loosely replicated in AFL++ by creating a corpus directory with a single one-byte-sized file.

Running the fuzzer with *libFuzzer* is done by running the compiled executable and passing the initial corpus directory as a command-line argument. During execution, *libFuzzer* will add

```c
void read_file(char *filename, uint8_t **Data,  size_t *Size) {
  FILE *file = fopen(filename, "rb");
  if (file == NULL) {
    printf("Cannot open %s\n", filename);
    exit(1);
  }

  fseek(file, 0, SEEK_END);
  int program_size = ftell(file);
  rewind(file);

  *Size = program_size;
  *Data = (uint8_t *) malloc(sizeof(uint8_t) * *Size);
  fread(*Data, sizeof(uint8_t), program_size, file);
  fclose(file);
}
```

**Figure 6.5:** *read_line* function that reads the contents and size of a file

generated test cases covering new paths through the code under test to this corpus directory. The fuzzing process lasts indefinitely until a crashing test case is discovered [2].

AFL++ includes the command-line utility `afl-fuzz` to initiate the fuzzing process. It can be started by specifying the input corpus directory, the output directory (which will hold the fuzzing process state, including generated corpus and crashing test cases), the fuzz target binary, and the input format (which specifies if test cases are fed into the fuzz target as, for example, standard input or a command-line argument with the test case file path). The fuzzing process can be monitored through the status screen, as shown in Figure 5.5 (see Section 5.3.1). Since AFL++ persists the fuzzing process state in the output directory, it can be freely halted and resumed later. Unlike *libFuzzer*, the fuzzing process has to be manually halted by the user in AFL++. On the top-right corner of the status screen, the cycle counter (each cycle is a full pass of the test cases in the queue) is conveniently color-coded to indicate if further execution is worth it or not. It is magenta during the first cycle, becomes yellow if new finds are still being made in subsequent rounds, then blue when that ends. When the fuzzing process continues lacking progress for some more time, it finally becomes green. For our purposes, we terminate the fuzzing process when the cycle counter turns green on synthetic cases. On real-life projects, we terminate the fuzzing process when a crashing test case for the desired fault is discovered.

### 6.2.5 APR Configuration

The APR step of the process is done once for each APR technique and each faulty project. As stated in Section 6.2.3, we use four APR techniques for each project: (i) *Angelix* [42], (ii) *SemFix* [44], (iii) *GenProg* [25] and (iv) *Prophet* [36]. A common step for all four of these projects is

**Table 6.5:** APR technique environment information

| Environment | Operating System | GCC Version |
|---|---|---|
| *Angelix*<br>*SemFix* | Ubuntu 14.04.6 LTS | 9.4.0 |
| *GenProg* | Ubuntu 18.04.5 LTS | 10.3.0 |
| *Prophet* | Ubuntu 16.04.7 LTS | 9.4.0 |

to adapt the fuzz target to serve as an entry point for the test cases generated in the fuzzing step. Most of the real-life project fuzz targets are, as described in Section 6.2.4, a function receiving the test case contents and size as parameters. We adapt these fuzz targets by replacing this function with a `main` function that wraps the same functionality, but first receives the test case path as a command-line argument and calls a function (Figure 6.5) that reads the contents and size from that file. Since the contents are allocated on the heap, they have to be freed before returning. Otherwise, if the AddressSanitizer is being used, then it will trigger on every execution.

The oracle/test execution script for each APR technique consists of executing the modified fuzz target for generated test cases (including the fault-triggering test case) and executing project-specific tests. For project-specific tests, the projects we use support the execution of each test individually, but this execution and verifying if the test passes or not depends on the project. Since the inclusion of these tests cannot be generalized without developer input, we separate the results with and without project-specific tests in Section 6.3.

The four techniques are executed in three different environments (*Angelix* and *SemFix* share the same environment, as *Angelix* contains an implementation of *SemFix*'s algorithm). Details about these environments are presented in Table 6.5 and detailed as follows:

***Angelix/SemFix*:** For *Angelix/Semfix*, we need to manually instrument the target project with output values (`ANGELIX_OUTPUT(type,expr,label)`), and coverage labels (`ANGELIX_REACHABLE(label)`). This instrumentation allows *Angelix/Semfix* to know the actual values and reachability of negative tests, to compare against the expected values and coverage, as such information has to be provided to *Angelix/Semfix* when executing it. We do not know if this instrumentation serves other purposes in the APR process. In our situation, following the rationale of the process being integrable in practical use-cases, meaning this instrumentation has to be automatable, our instrumentation consists of a single reachability label at the end of every normal exit path (exit code 0). Reaching this label means that the program did not crash (i.e., the fault was not triggered).

In all projects, we use the following command-line arguments: (i) `--klee-ignore-errors` which silences KLEE errors, as they are irrelevant and obfuscate the logs (KLEE is a symbolic execution engine built on top of the LLVM compiler infrastructure and used by *Angelix*); (ii) `--klee-timeout 300`, (iii) `--synthesis-timeout 180000`, and (iv) `--generate-all` so that *Angelix* doesn't halt execution after generating one patch (i.e., generates all patches). In some

cases, *Angelix* runs indefinitely (up until the machine runs out of available memory) without the two timeout flags. The values used for timeout correspond to the values used by *Angelix* authors to patch Heartbleed [42]. For each individual program, we use specific command-line arguments to indicate which defect classes to consider and which repair components to use in synthesis, as described in their manual[12]:

- **CWE-121 (Stack Buffer Overflow), CWE-134 (Format String):** `--defect guards if-conditions --synthesis-levels alternatives integer-constants extended-arithmetic extended-logic extended-inequalities --synthesis-used-vars`
- **CWE-122 (Heap Buffer Overflow):** `--defect if-conditions --synthesis-levels alternatives integer-constants extended-arithmetic extended-logic extended-inequalities --synthesis-used-vars`
- **CWE-369 (Divide by Zero):** `--defect guards --synthesis-levels alternatives integer-constants extended-arithmetic extended-logic extended-inequalities --synthesis-used-vars`
- **CWE-190 (Integer Overflow):** `--defect assignments loop-conditions --synthesis-levels alternatives integer-constants extended-arithmetic extended-logic extended-inequalities --synthesis-used-vars`
- **CVE-2014-0160 (Heartbleed):** `--synthesis-levels extended-arithmetic --synthesis-used-vars`
- **CVE-2019-7317:** `--defect guards if-conditions --synthesis-levels alternatives integer-constants extended-arithmetic extended-logic extended-inequalities`
- **CVE-2017-5336, CVE-2020-12284:** `--defect guards --synthesis-levels extended-arithmetic --synthesis-used-vars`

***GenProg*:** Besides requiring a set of tests and build instructions for the target program, *GenProg* also requires the preprocessed source code of the program. In order to meet this requirement in real-life projects, we add the `--save-temps` flag (makes it so that temporary intermediate files such as preprocessed and, potentially, object files are saved), to `CFLAGS` when configuring the target project. While this solution is crude, it does not require changing the target project build script, and can be applied to any project with a build system.

[12] *Angelix* manual: https://github.com/mechtaev/angelix/blob/master/doc/Manual.md

**Table 6.6:** APR results with synthetic programs identified by their weakness identifier

| **CWE-ID** | $T^+_{\text{sample}}$ | $T^+$ | $T^-$ | ***Angelix***[1] | ***SemFix***[1] | ***GenProg***[1] | ***Prophet***[1] |
|---|---|---|---|---|---|---|---|
| CWE-121 | 2 | 2 | 1 | 0/0 | 0/0 | 0/0 | 0/0 |
| CWE-122 | 3 | 89 | 8 | 0/0 | 0/0 | 0/0 | 0/0 |
| CWE-369 | 1 | 1 | 1 | 0/0 | 0/0 | CR | CR |
| CWE-134 | 1 | 1 | 1 | 0/0 | 0/0 | 0/0 | 0/0 |
| CWE-190 | 1 | 7 | 4 | 0/0 | 0/0 | PE | 0/0 |

[1] `x` correct patches out of `y` generated patches (`x`/`y`), Cannot Reproduce fault (CR), Parsing Error (PE)

***Prophet*:** In order to maximize the patch search-space, we use the following command-line argument flags with *Prophet*: `-blowup -full-synthesis -full-explore -first-n-loc 200 -consider-all`. *Prophet* comes with a set of pre-trained models for ranking candidate patches. We use the `para-all.out` feature parameters for the model, as they appear to be the most suitable for our situation since they are not targeted towards a specific project.

## 6.3 Results

In this section, we present the results of applying the joint fuzzing-APR process, described in Section 6.2, and a brief analysis of the results for each fault and APR technique. We start by presenting our results with the synthetic programs in Section 6.3.1. Then, in Section 6.3.2, we present our results for real-life programs with and without the project's tests being fed to APR. Finally, we analyze the results in Section 6.3.3.

### 6.3.1 Synthetic Programs

For each synthetic program, identified by the CWE-ID of their corresponding vulnerability type, as listed in Table 6.1, we start the fuzzing process with an initial sample of $T^+_{\text{sample}}$ valid tests cases. Then, we feed each APR technique with $T^+$ positive test cases (sample + generated test cases), and $T^-$ generated fault-triggering test cases. The results, presented in Table 6.6, show the number of patches generated by each technique, as well as the number of generated patches that are correct.

In terms of test case generation, we notice three of the five programs have few generated test cases before finding a fault-triggering test case, which is expected as the programs were developed to be simple and the sample test cases are meant to cover the whole program. The exception to this is CWE-122 since the program is an interpreter. AFL++ generated test cases for CWE-190, likely due to how it handles path coverage in loops or compiler-level optimizations.

Regarding the effectiveness of the APR techniques, when applied to these synthetic programs, none of the techniques generate at least one patch. In CWE-369 (Divide by Zero), we are unable to reproduce the fault in *GenProg*'s or *Prophet*'s environment. The sanitizer does not reliably

**Table 6.7:** APR results with real-life projects identified by their vulnerability identifier

| **CVE-ID** | $T^+_{\text{project}}$ | $T^+$ | $T^-$ | ***Angelix***[1] | ***SemFix***[1] | ***GenProg***[1] | ***Prophet***[1] |
|---|---|---|---|---|---|---|---|
| CVE-2014-0160 | 0 | 47 | 2 | 0/1 | IE | PE | 0/0 |
| | 38 | 85 | 2 | 0/0 | IE | PE | 0/0 |
| CVE-2017-5336 | 0 | 116 | 1 | RIE | RIE | PE | 0/0 |
| CVE-2019-7317 | 0 | 0 | 1 | 0/0 | 0/0 | PE | 0/0 |
| | 8 | 8 | 1 | 0/0 | 0/0 | PE | 0/0 |
| CVE-2020-12284 | 0 | 383 | 1 | CR | CR | CR | CR |

[1] `x` correct patches out of `y` generated patches (`x/y`), Inference Error (IE), Parsing Error (PE), Repairable Instrumentation Error (RIE), Cannot Reproduce fault (CR)

trigger in these environments for reasons we could not ascertain. Running *GenProg* with CWE-190 (Integer Overflow) results in a parsing error when preprocessing the program. This issue is described in more detail in Section 6.3.3.

### 6.3.2 Real-life Projects

Unlike the synthetic programs, for each real-life project, identified by the CVE-ID of the corresponding vulnerability, as listed in Table 6.2, we start the fuzzing process with an empty corpus (i.e., without an initial set of test cases). For every project, we feed each APR technique with $T^+$ positive generated test cases, and $T^-$ generated fault-triggering test cases. For projects with inspiring results (i.e., at least one APR technique did not crash or gave up early), we fed an additional $T^+_{\text{project}}$ positive tests, extracted from the project's tests, to the $T^+$ total number of positive tests. The results, presented in Table 6.7, show the number of patches generated by each technique, as well as the number of generated patches that are correct, both with and without project tests in designated projects.

When applied to real-life projects, only *Angelix* was capable of producing a patch, in this case for Heartbleed, as shown in Figure 6.6). However, this patch is not correct, as it simply avoids the feature (i.e., is overfitting the provided tests). By adding OpenSSL's tests to the set of tests, this patch is no longer generated. *SemFix* crashes during the inference step when attempting to patch Heartbleed. In CVE-2017-5336, *Angelix/Semfix* crash when attempting to instrument the repairable source file. Furthermore, we cannot reproduce CVE-2020-12284 in the APR techniques' environments, and *GenProg* always results in a parsing error when preprocessing the source code. These issues are described in more detail in Section 6.3.3.

### 6.3.3 Analysis

Throughout the nine selected projects, only a single patch is generated (Heartbleed patch by *Angelix*). Furthermore, this patch is not correct, and the fault corresponds to one with which *Angelix* was validated and that the authors claim to be fixable with *Angelix* [42]. This result motivates a

```
1  --- a/openssl-1.0.1f/ssl/t1_lib.c
2  +++ b/openssl-1.0.1f/ssl/t1_lib.c
3  @@ -2568,7 +2568,7 @@
4                              &s->s3->rrec.data[0], s->s3->rrec.length,
5                              s, s->msg_callback_arg);
6
7  -        if (hbtype == TLS1_HB_REQUEST)
8  +        if ((hbtype == s->msg_callback_arg))
9                  {
10                 unsigned char *buffer, *bp;
11                 int r;
```

**Figure 6.6:** Generated Heartbleed patch diff file using our process

replication of the original experiments, described in Section 6.4. Adding project tests to the set of tests fed into APR appears not to affect FL or increase the search space of candidate patches. However, it contributes towards filtering overfitted patches, so they are a positive addition if available. In the following paragraphs, we describe identified issues for each APR technique, and in reproducing some faults:

***Angelix/Semfix*:** *SemFix* crashed when attempting to repair CVE-2014-0160 (Heartbleed). The execution goes along similarly with *Angelix*'s, but during one of the inference steps, *SemFix* throws an error ("Inconsistent Variables"). In CVE-2017-5336, *Angelix/SemFix* cannot successfully instrument the source file containing the fault. We plan on creating issues in *Angelix*'s repository relating to these issues once our experiments become public.

***GenProg*:** *GenProg* uses C Intermediate Language (CIL)[13] to parse the source code and manipulate ASTs (in contrast to the other techniques which use LLVM). CIL is the source of *GenProg*'s parsing errors, which appear to be caused by unknown symbols present in preprocessed source code. For example, when applied to OpenSSL, it fails to parse mathematical functions from the C standard library due to CIL not supporting 128bit floats (we identified an open issue related to this problem in another project dependent on CIL[14]). Unfortunately, we could not find a workaround for this issue. Issues related to CIL have been created for *GenProg*, such as (i) issue #10[15], opened on 18th March 2018, which was closed by the maintainers stating it is an issue with CIL, not

[13] C Intermediate Language (CIL): https://github.com/cil-project/cil

[14] Obliv-C issue #48 - Fails to compile when including <math.h> of recent glibc: https://github.com/samee/obliv-c/issues/48

[15] *GenProg* issue #10 - Error parsing pthread_mutex_t: https://github.com/squaresLab/genprog-code/issues/10

```
1 FROM mechtaev/angelix:1.1
2 WORKDIR /src/
3
4 RUN apt-get install -y libffi-dev gtk-doc-tools libgtk2.0-dev libpcap-dev
5
6 COPY angelix-experiments/ .
7 RUN chmod +x ./repair ./repair-* ./fetch ./get-options
```

**Figure 6.7:** Dockerfile for reproducing *Angelix*'s original experiments

*GenProg* (although they have indicated a potential reason for that specific instance of the error); and (ii) issue #37[16], opened on 17th May 2021.

**Fault Reproduction:** In both CWE-134 and CVW-2020-12284, we have issues reproducing the fault in some APR environments due to the underlying sanitizer not triggering when executing the fault-triggering test case generated in the fuzzing environment. Even after updating GCC for the four APR techniques to the versions specified in Section 6.2.5, we could not reliably trigger the UndefinedBehaviorSanitizer on a division by zero (CWE-369) with *GenProg* or *Prophet*. Furthermore, we could not reproduce CVE-2020-12284 in any of the APR techniques.

## 6.4 Replication of Original APR Experiments

Due to the results obtained in Section 6.3, we decide to replicate *Angelix*'s original experiments. We set up a docker container based on the *Angelix* official image, with the experimental data[17] in it and dependencies installed. The Dockerfile for this container is shown in Figure 6.7. We could not generate any of the original patches. The experimental scripts are dependent on an unavailable Uniform Resource Locator (URL) to generate all patches except for Heartbleed. We can run the original experiment for Heartbleed, but no patch is generated. We have an issue open in *Angelix*'s repository[18], but we did not obtain an answer before the publication date.

We do a more thorough analysis of the Heartbleed patch reproduction script to ensure it is being properly executed. We notice that, before running *Angelix*, OpenSSL's source code is manually modified (see Figure 6.8) in the same location as the Heartbleed patch presented by *Angelix*'s authors (see Figure 6.3) [42]. We assume the authors apply this change because *Angelix* cannot extend if-statements with more conditions. Hence, they denormalize the code manually so that

[16] *GenProg* issue #37 - Error when running on Coreutils: https://github.com/squaresLab/genprog-code/issues/37

[17] *Angelix* experimental data: http://www.comp.nus.edu.sg/~release/angelix/angelix-experiments.tar.gz

[18] *Angelix* Issue #27 - Can't reproduce experimental results with docker image: https://github.com/mechtaev/angelix/issues/27

```
1  --- a/ssl/t1_lib.c
2  +++ b/ssl/t1_lib.c
3  @@ -2596,7 +2596,7 @@ tls1_process_heartbeat(SSL *s)
4          if (s->msg_callback)
5              s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
6                      &s->s3->rrec.data[0], s->s3->rrec.length,
7                      s, s->msg_callback_arg);
8  -       if (hbtype == TLS1_HB_REQUEST)
9  +       if (hbtype == TLS1_HB_REQUEST && 0 < 1)
10             {
11                 unsigned char *buffer, *bp;
```

**Figure 6.8:** OpenSSL source diff applied before running *Angelix*

*Angelix* can patch Heartbleed. In light of this discovery, it appears that *Angelix* is actually not capable of patching Heartbleed by itself.

## 6.5 Threats to Validity

In this section we identify and discuss a number of threats to the validity of this empirical study, namely threats to internal validity (Section 6.5.1), and to external validity (Section 6.5.2).

### 6.5.1 Internal Validity

Internal validity represents how valid the cause-and-effect relationships are between the independent variables and observed effects in dependent variables. We identify and describe the following threats to the internal validity of this empirical study:

**Fuzz-generated Test Cases** The fuzzing step of the process is crucial not only to identify faults (we were able to reproduce all sample faults and generate a fault-triggering test case for each) but to generate a set of test cases to feed APR with to help filtering overfitted candidate patches. We have not measured the actual impact of feeding these generated test cases to APR. While we expect that these test cases create a better specification of the desired behavior, it is possible that these test cases have the opposite effect (i.e., create an overfitted specification) and guide the APR process away from a correct patch. Judging from *Angelix* execution logs, only the one fault-triggering test case was relevant for FL, so we believe adding other generated test cases does not affect FL.

Another aspect to consider is how the fuzzing process is configured to generate test cases. Specific strategies are more or less relevant, from a fuzzing perspective, depending on the target project (e.g., deterministic fuzzing strategies like bit flipping are not well suited for programs dealing with structured data, such as LibXML2). The strategies used in the fuzzing process likely affect which relevant test cases are generated. AFL++ has a crash exploration mode, which takes

crashing test cases as input and tries to cover all the paths that can be reached while continuing in the crashing state [1]. This mode can be used to encounter further faults caused by the initial fault and can perhaps be used to generate more fault-targeted test cases.

**APR Technique Configuration** Due to our lack of knowledge about each APR technique's implementation details and internals, we cannot guarantee that the lack of effectiveness of the techniques is not due to improper setup or configuration. We have read the available documentation and used provided Docker images when available to minimize this threat. Regardless, we identify the following concerns:

- ***Angelix* Timeouts Configuration:** As mentioned in Section 6.2.5, the timeout flags for KLEE and patch synthesis in *Angelix* correspond to the values used by *Angelix* authors to patch Heartbleed. As a result, *Angelix*'s results may be overfitted for Heartbleed, and patches may not be being generated because of early timeouts. Unfortunately, there is not much we can do about this other than pinpointing the timeout values for each case, which is not a feasible solution in a practical use case. We believe that an APR tool should suggest whether further exploration is relevant or not dynamically (similarly to AFL++ color-coded cycle counter) instead of relying on "magic values" for each particular case.
- ***Angelix* Program Instrumentation:** In Section 6.2.5 we describe the *Angelix* manual instrumentation process. This process is only applied to the adapted fuzz target. Therefore project-specific tests are not instrumented. We believe the sole purpose of this instrumentation is for *Angelix* to know the actual output values and coverage of negative tests. Under this assumption, since the project-specific tests are positive, instrumenting these tests would not affect *Angelix*'s effectiveness. However, we do not know how *Angelix* leverages this instrumentation since the usefulness of this instrumentation is not documented.
- **Suspicious *Prophet* Execution:** We slightly suspect that *Prophet* is not set up correctly because all of its executions are almost instantaneous and produce the same result as shown in Figure 6.9. The lack of feedback provided by *Prophet*, such as the fact that no error is reported, leads to a confusing interpretation of the results and an inability to act on the issue.

**Cross-Environment Process** As described in Section 6.2, we perform the fuzzing and APR steps in different environments. A side-effect of this is that a test case triggering the fault in the fuzzing environment may not trigger the fault in the APR environment. When this happens, we mark the result as that the fault cannot be reproduced. As shown in Section 6.3, CWE-369 and CVE-2020-12284 have results with this mark. In CVE-2020-12284, we believe the issue is due to using an older version of AddressSanitizer (The CVE is more recent than the others, so we assume it was discovered with a more recent version of AddressSanitizer). We had a similar issue initially in CVE-2019-7317 that we solved by upgrading GCC in the APR environments to the version specified in Table 6.5. Upgrading GCC to even more recent versions could potentially solve this

```
Initialize the program!
Verify Test Cases
All passed!
Done Verification
Generating repair candidates!
Total 0 different repair schemas!!!!
Total 0 different repair candidate templates for scoring!!!
Total 0 different partial repair candidate templates!!
Trying different candidates!
BasicTester pointer: 0x1d1dcc0
StringConstTester pointer: 0x1d1db50
CondTester pointer: 0x1d1d9c0
The total number of synthesis runs: 0
The total number of concrete conds: 0
The total number of explored concrete patches: 0
Repair process ends without working fix!!
Total 0 different repair schemas!!!!
Total 0 different repair candidate templates for scoring!!!
Total number of compiles: 0
Total number of test eval: 0
```

**Figure 6.9:** Standard output of a *Prophet* execution

reproduction issue in CVE-2020-12284. Another option would be to use the same compiler in both types of environments (instead of Clang in the fuzzing step and GCC in the APR step). Using the same version of Clang for each APR technique, as the version of Clang used in the fuzzing step would most likely solve the fault reproduction issue in CVE-2020-12284 (assuming that the APR techniques support Clang, which we have not checked). As for CWE-369, we are unsure if using the same compiler would solve the fault reproduction issues in the *GenProg* and *Prophet* environments.

### 6.5.2 External Validity

External validity represents how valid it is to apply the conclusions outside this study. We identify and describe the following threats to the external validity of this empirical study:

**Sample Size** In this empirical study, we apply the process described in Section 6.2 to 5 vulnerable synthetic programs and 4 real-life vulnerabilities. We are well aware that the sample size of real-life vulnerabilities is too low to make any reliable conclusions on the overall effectiveness of APR when applied to fuzz-identified vulnerabilities (for reference, there are over 26000 issues in OSS-Fuzz marked as fixed and verified[19]. Regardless, the findings of our study reveal issues in

[19]List of OSS-Fuzz issues marked as fixed and verified (Last accessed in 20-06-2021): https://bugs.chromium.org/p/oss-fuzz/issues/list?q=status%3AVerified&can=1

the current state of APR techniques and motivate a more thorough study of APR applicability in continuous fuzzing and other practical use cases.

**Sample Charateristics** When developing the synthetic programs, we ensured each program represented a unique type of vulnerability. However, the selection of which weaknesses to include was ad hoc. A better criterion for selecting which weaknesses to include would be to analyze the type of faults being reported in OSS Fuzz and select the most predominant ones. In hindsight, some weaknesses like Divide by Zero do not bring much value to the study. Regarding the real-life vulnerabilities, we filter for OSS-Fuzz vulnerabilities that have a CVE assigned to them, with the goal of selecting vulnerabilities with a higher impact. However, this filtering may also be a source of bias.

**APR Technique Selection** This study only evaluates APR techniques targeting C as fuzzing engines and projects leveraging coverage-guided fuzzing are in C/C++. Furthermore, we are only able to use APR techniques that are publicly available. During our research of the state of the art, we found some APR techniques focusing on security vulnerabilities [21, 37]. However, we could not include them due to the lack of a publicly available implementation.

## 6.6 Discussion

The planning and development of this experimentation were done to evaluate the validity of the following hypothesis, as first introduced in Section 4.2:

> *"Using fuzzing as a source of faults and of test cases allows for APR techniques to be effective when integrated with the SDLC."*

The results of this study appear to invalidate this hypothesis, as suspected by our initial exploration of APR techniques in Section 5.2. In Section 6.5 we present some threats to the validity of this study, which may compromise our ability to evaluate the hypothesis reliably. Regardless, we believe our study is a strong indicator of our hypothesis's invalidity, supported by the failure to reproduce *Angelix*'s original experiments, described in Section 6.4. To help guide the evaluation of the validity (or lack thereof) of our hypothesis, we present answers to our research questions, as defined in Section 4.4, based on the findings of this empirical study:

**Research Question 1** *Are the selected APR techniques capable of generating correct patches when applied in faults discovered by fuzzing?*

For this study, we created a joint fuzzing–APR process, described in Section 6.2.1, to evaluate APR's ability to generate patches from faults identified through fuzzing. As we present in Section 6.3, in the scope of our study, the selected APR techniques are not capable of generating correct patches, producing only a single overfitted patch. Despite the short sample size, taking into account that we include five small synthetic programs and that one of the selected APR techniques

had reportedly successfully patched one of the chosen vulnerabilities (*Angelix* and Heartbleed, respectively), we are confident that this lack of effectiveness is not just due to an unlucky choice of sample faults, and probably applies to a majority of faults.

**Research Question 2** *What types of faults, in the context of fuzzing, are selected APR techniques more capable of generating correct patches?*

This Research Question was created to help identify which types of faults are better suited for APR so that practical solutions can focus more on these types of faults, and potentially establish a correlation between APR strategies and the types of faults suiting them best. However, as we present in Section 6.3, none of the fault types included in the study appear to be suitable for APR as no correct patches are generated.

**Research Question 3** *What are the main challenges in applying APR to projects within the context of fuzzing?*

The development of the fuzzing–APR process and the analysis of the results helped us identify multiple challenges in integrating APR with fuzzing and identify issues in how APR techniques are designed. The key challenges that we identify that bring down APR's ability to be effective and applicable in practical use cases are the following:

- Techniques can quickly become outdated and unusable if they do not keep up with language standards and the software ecosystem on which both the technique and its target projects depend. As with most practical software, it needs active maintainers to maintain its functionality.

- A lack of user-friendliness from techniques and feedback from their executions reduces their potential of being adopted and successfully integrated with the SDLC. Techniques should be well documented and as straightforward to set up and configure as possible. More importantly, setting up and configuring an APR tool should be project-driven and never fault/patch-driven (e.g., require the user to specify timeout values and which repair elements to use explicitly). In our opinion, a developer should only have to specify a single time how to set up and configure a project for usage with APR. A developer should never have the configure an APR tool based on the characteristics of a specific fault or on how it can be repaired, as that defeats the purpose of APR.

- The lack of adoption of APR techniques means that they are barely used, and, as a result, there are greater chances of their functionality not being correct due to a lack of community feedback. On the other hand, APR techniques may not be being adopted because their functionality is not correct. To break this cycle, we suggest that APR tools are developed with the primary goal of being applied in practice, rather than for research purposes, starting with a subset of faults we know APR can be effective at, then expanding in functionality as community adoption and support evolves.

## 6.7 Summary

We start this chapter by explaining the objectives of this empirical study in Section 6.1. In Section 6.2 we detail the planning of our study, namely the development of a joint fuzzing–APR process, and the selection and configuration of fuzzing engines and APR techniques. We present and analyze the results of our study in Section 6.3, followed by our attempt at replicating *Angelix*'s experiments in Section 6.4. We then analyze some identified threats to the validity of this study in Section 6.5 and, finally, present a critical discussion of the results based on the proposed hypothesis and research questions in Section 6.6.

# Chapter 7

# Conclusions

7.1 **Summary** . . . . . . . . . . 57
7.2 **Contributions** . . . . . . . . . . 58
7.3 **Future Work** . . . . . . . . . . 58

## 7.1 Summary

Software development is an increasingly complex task, as software is applied to an expanding scope of applications requiring faster, more efficient, and more secure software. As a result, software faults become more common and have a greater negative impact. To combat this trend, software researchers are exploring the development of automated program repair solutions capable of generating patches for a given fault, using a set of tests as specification. Most of current APR research is done in an academic environment, lacking tools developed for practical use-cases. We identify several issues, described in Section 4.1, in the practical APR tools we find that apply APR during software integration. With the addition of more layers of abstraction, software in the lower layers such as operating system utilities, file format libraries, and cryptography and networking libraries become more critical since a fault in them impacts a broader range of software products and devices. Coverage-guided fuzzing is a software testing technique well suited to these types of software. Fuzzing allows for the identification of faults through the automated generation of test cases. APR has the potential of being integrated with fuzzing as a continuation of the fuzzing process that attempts to repair the identified fault, using the generated test cases as part of the test-suite being used as specification.

We propose to apply APR techniques after fuzzing sessions, using coverage-guided fuzzing engines, and integrate with the SDLC (i.e., be able to fuzz a project continuously and suggest generated patches of identified faults to developers for eventual review and integration). Through this development, we aim to evaluate the effectiveness of APR techniques (i.e., their ability to generate correct patches) on practical environments, using coverage-guided fuzzing as a source of

information (*c.f.* Section 4.2). Our initial development, described in Chapter 5, shows potential in using a GitHub bot to suggest patches and facilitate interaction with developers, and in automating the fuzzing process using a coverage-guided engine. However, we could not make current APR techniques generate patches, even in simple synthesized scenarios with straightforward fixes.

As a result, we pivoted our development towards an empirical study (see Chapter 6) aimed at validating/invalidating our hypothesis and at answering our research questions enumerated in Section 4.4. In a matrix of nine faulty programs (five synthetic programs + four real-life projects) and four APR techniques, we cannot generate a single correct patch. An overfitted patch is generated for Heartbleed when we do not include the project's tests. Another concerning aspect is that *Angelix*, one of the APR techniques selected for the study, is reported to be capable of correctly patching Heartbleed by its authors [42]. We analyzed their experiments reproduction package and discovered that *Angelix* is, in fact, incapable of patching Heartbleed by itself, requiring first a manual change to OpenSSL's source code. This discovery compromises the credibility of APR research and further supports APR's lack of effectiveness in practical use cases.

We hope that our work motivates more thorough studies on APR's capabilities when applied in realistic scenarios and APR tools to be developed for practical applications, rather than satisfy/overfit existing benchmarks and other synthetic scenarios.

## 7.2 Contributions

Despite the negative results and consequential invalidation of our hypothesis, we believe that our dissertation work produced valuable contributions to automated program repair research, such as:

- **Literature Review on APR techniques and practical APR solutions:** In Chapter 3, we analyzed the state of the art on fourteen of the latest open-source APR techniques and on three solutions integrating APR with the SDLC in the integration step.
- **Empirical Study on APR applied with fuzzing:** In Chapter 6 we conceptualize a joint fuzzing–APR process to evaluate APR effectiveness when fed with fuzzing information. We apply this process to each of four APR techniques to generate patches to nine faults (five from simple synthesized programs, and four real-life vulnerabilities from open-source projects).

## 7.3 Future Work

Our empirical study revealed a lack of effectiveness from current APR techniques when applied with coverage-guided fuzzing methods, motivating a more thorough investigation and experimentation, both with fuzzing and in other scenarios with practical use-cases. Additionally, the planning and development of this study gave us some insights and thoughts about the future direction of APR and how to develop APR tools better suited for practical scenarios. In this section, we

present some suggestions and improvements for future development in the research field of APR and its practical application.

**More thorough Empirical Study:** Our empirical study covered few real-life projects (four real-life vulnerabilities). OSS-Fuzz [3] is a good source of vulnerabilities discovered by fuzzing and has over 26000 vulnerabilities that have been fixed in the past. Section 6.5 presents other improvements that could be applied in a future study, such as:

- Avoid the fault reproduction issue caused by differing fuzzing and APR environments;
- Study the impact of enriching the test suite with all generated test cases versus just feeding the fault-triggering test case;
- Attempt to include more APR techniques, such as *Fix2Fit* [22], in the study.

**Experiment other other practical applications of APR:** There are other potential use cases for APR than continuous fuzzing, namely integrated with the SDLC. For example, *pAPRika* [13] is a Visual Studio Code extension that suggests fixes while the programmer is writing code. Live Programming is another form of SDLC integration focused on the implementation step. Given that APR research is not focused on practical use, studying its effectiveness in practical environments such as these helps identify critical shortcomings in APR and help pivot its research towards the development of tools useful for software developers.

**Develop an APR technique targeting fuzzed projects:** There is potential in developing an APR technique tailored for faults identified through fuzzing, as in aware of Sanitizers and how they affect fault localization and code coverage. Typically the fault can be localized from one of many source locations in the crashing call stack, meaning that the typical spectrum-based fault localization used by current APR techniques may not be the best solution.

During the development of our study, we notice that it is common for a fault to be instantiated multiple times in a project. For example, Heartbleed is instantiated both in TLS and DTLS implementations of OpenSSL. When a patch for a fault is made, it typically patches all instances of the fault, however detecting these instances is not trivial, and, in our opinion, is worth its own dedicated research. Identifying closely related faults could be a method of detecting instantiations of the same fault. Memory related faults is another scenario where multiple instantiations can occur (e.g., add bounds check before reading a buffer in every location the buffer is being read). In these situations, a combination of static and dynamic analysis may allow for the detection of every fault instance (e.g., location the buffer is being read).

# Appendix A

# Synthetic Programs

## A.1 CWE-121 (Stack Overflow) - Restricted Access

```
#include <stdio.h>
#include <string.h>

void doAdminStuff()
{
  printf("Here's the special cookie for admins only.\n");
}

int main(int argc, char const *argv[])
{
  char check[] = "trustmeimadmin\n";
  char input[20];

  printf("Prove you are admin.\n");
  fgets(input, 64, stdin);

  if (strcmp(input, check) == 0)
    doAdminStuff();
  else {
    printf("Not admin, go away.\n");
  }

  return 0;
}
```

## A.2 CWE-122 (Heap Overflow) - Brainfuck Interpreter

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

char *read_program(const char *filename) {
  FILE *file = fopen(filename, "r");
  if (file == NULL) {
    printf("Cannot open %s\n", filename);
    exit(1);
  }

  fseek(file, 0, SEEK_END);
  long program_size = ftell(file);
  rewind(file);

  char *program = malloc(program_size + 1);
  fread(program, 1, program_size, file);
  fclose(file);

  program[program_size] = 0;
  return program;
}

void interpret(char *program) {
  int n = strlen(program);
  int scope = 0;

  uint8_t *tape = calloc(2048, sizeof(uint8_t));
  int pointer = 1024;

  for (int pc = 0; pc < n; pc++) {
    char instr = program[pc];
    // printf("%p %d %p %d %c\n", tape, pointer, &tape[pointer], tape[pointer], instr);

    switch (instr)
    {
    case '+': // Increment
```

```
39          tape[pointer] = tape[pointer] < UINT8_MAX ? tape[pointer] + 1 : 0;
40          break;
41        case '-': // Decrement
42          tape[pointer] = tape[pointer] > 0 ? tape[pointer] - 1 : UINT8_MAX;
43          break;
44        case '<': // Move Left
45          if (--pointer < 0) {
46            pointer = 2047;
47            if (tape[pointer] != 0) {
48              puts(" [X] Tape Memory Underflow detected - Aborting\n");
49              exit(1);
50            }
51          }
52          break;
53        case '>': // Move Right
54          if (++pointer > 2048) { // BUG: Off-by-one error
55            pointer = 0;
56            if (tape[pointer] != 0) {
57              puts(" [X] Tape Memory Overflow detected - Aborting\n");
58              exit(1);
59            }
60          }
61          break;
62        case '.': // Write
63          putchar(tape[pointer]);
64          // printf("%d|", tape[pointer]);
65          break;
66        case ',': // Read - Unsupported Instruction
67          break;
68
69        // Dumb implementation of Jump Scopes
70        // to avoid implementing necessary data structures
71        case '[': // Jump If Zero
72          if (tape[pointer] != 0)
73            break;
74
75          for (int i = pc; i < n; i++) {
76            char c = program[i];
77
78            if (c == '[') {
```

```
79            scope++;
80          }
81          else if (c == ']' && --scope == 0) {
82            pc = i;
83            break;
84          }
85        }
86
87        if (program[pc] != ']') {
88          printf(" [X] No ] matching the [ at position %d - Aborting\n", pc);
89          exit(1);
90        }
91        break;
92      case ']': // Jump Unless Zero
93        if (tape[pointer] == 0)
94          break;
95
96        for (int i = pc; i >= 0; i--) {
97          char c = program[i];
98
99          if (c == ']') {
100           scope++;
101         }
102         else if (c == '[' && --scope == 0) {
103           pc = i;
104           break;
105         }
106       }
107       if (program[pc] != '[') {
108         printf(" [X] No ] matching the [ at position %d - Aborting\n", pc);
109         exit(1);
110       }
111       break;
112     default:
113       break;
114     }
115   }
116   free(tape);
117 }
118
```

```
119 /*
120   Subset of Brainfuck (Does not support input)
121 */
122
123 int main(int argc, char const *argv[])
124 {
125   char *program = read_program(argv[1]);
126   interpret(program);
127   free(program);
128   return 0;
129 }
```

## A.3 CWE-369 (Divide by Zero) - Division Operation

```c
#include <stdio.h>
#include <string.h>

float quotient(float a, float b) {
    return a / b;
}

int main(int argc, char const *argv[])
{
  float a, b;

  scanf("%f %f", &a, &b);
  printf("%f\n", quotient(a,b));
  return 0;
}
```

## A.4 CWE-134 (Format String) - String Customizer

```
#include <stdio.h>
#include <string.h>

int main(int argc, char const *argv[])
{
  char cookie[20];

  printf("Want a personalized cookie? Give us your name!\n");
  fgets(cookie, 20, stdin);

  printf("Here's your cookie: ");
  printf("C - ");
  printf(cookie);

  return 0;
}
```

## A.5 CWE-190 (Integer Overflow) - Useless Computation

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char const *argv[])
{
  int cookie_count;
  printf("I am cookie monster. How many cookies will you give me?\n");
  scanf("%d", &cookie_count);

  int actual_cookie_count = cookie_count + cookie_count;
  printf("Just %d cookies? I will eat %d cookies.\n",
    cookie_count, actual_cookie_count);

  for (unsigned long i = 1; i <= actual_cookie_count; i++) {
    int four = 2 + 2;
    int three = four - 1;
  }

  return 0;
}
```

# References

[1] American fuzzy lop plus plus (afl++). Available at https://github.com/AFLplusplus/AFLplusplus. Accessed: 2021-06-15.

[2] libFuzzer – a library for coverage-guided fuzz testing. Available at https://llvm.org/docs/LibFuzzer.html. Accessed: 2021-06-03.

[3] OSS-Fuzz: Continuous fuzzing for open source software. Available at https://github.com/google/oss-fuzz. Accessed: 2021-06-04.

[4] Rui Abreu, Peter Zoeteweij, and Arjan J.c. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46, 2006.

[5] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. pages 89–98. Institute of Electrical and Electronics Engineers (IEEE), 4 2008.

[6] Sérgio Almeida, Ana CR Paiva, and André Restivo. Mutation-based web test case generation. In *International Conference on the Quality of Information and Communications Technology*, pages 339–346. Springer, 2019.

[7] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

[8] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[9] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: Toward a stateful network protocol fuzzer. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4176 LNCS:343–358, 2006. cited By 73.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[13] Diogo Campos, André Restivo, Hugo Sereno Ferreira, and Afonso Ramos. Automatic program repair as semantic suggestions: An empirical study. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 217–228, 2021.

[14] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[15] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 121–130, New York, NY, USA, 2011. Association for Computing Machinery.

[16] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018.

[17] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265–278, March 2011.

[18] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 65–74, 2010.

[19] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. pages 349–358. Institute of Electrical and Electronics Engineers Inc., 3 2017.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[21] Fengjuan Gao, Linzhang Wang, and Xuandong Li. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 786–791, Sep. 2016.

[22] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 8–18, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS*, November 2008.

[24] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41:1236–1256, 12 2015.

[25] Claire Le Goues, Thanh Vu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38:54–72, 2012.

[26] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62:56–65, 11 2019.

[27] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 1345–1351. AAAI Press, 2017.

[28] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. volume 1, pages 215–224. ACM Press, 2010.

[29] Rene Just, Chris Parnin, Ian Drosos, and Michael D. Ernst. Comparing developer-provided to user-provided tests for fault localization and automated program repair. pages 287–297. Association for Computing Machinery, Inc, 7 2018.

[30] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. pages 437–440. Association for Computing Machinery, Inc, 7 2014.

[31] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. iFixR: bug report driven program repair. volume 19, pages 314–325. ACM, 8 2019.

[32] Matt Lake. Epic failures: 11 infamous software bugs. Available at https://www.computerworld.com/article/2515483/epic-failures-11-infamous-software-bugs.html, September 2010. Accessed: 2021-01-31.

[33] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. volume Part F130154, pages 727–739. Association for Computing Machinery, 8 2017.

[34] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 727–739, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices*, 51:298–312, 4 2016.

[36] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 298–312, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. VuRLE: Automatic vulnerability detection and repair by learning from examples. In *Computer Security – ESORICS 2017*, pages 229–246, Cham, 2017. Springer International Publishing.

[38] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. pages 94–105. Association for Computing Machinery, Inc, 7 2016.

[39] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. SapFix: Automated end-to-end repair at scale. pages 269–278. Institute of Electrical and Electronics Engineers Inc., 5 2019.

[40] Matias Martinez and Martin Monperrus. ASTOR: A program repair library for Java (Demo). pages 441–444. Association for Computing Machinery, Inc, 7 2016.

[41] Matias Martinez and Martin Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. volume 11036 LNCS, pages 65–86. Springer Verlag, 9 2018.

[42] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, May 2016.

[43] Martin Monperrus, Simon Urli, Thomas Durieux, Matias Martinez, Benoit Baudry, and Lionel Seinturier. Repairnator patches programs automatically. *Ubiquity*, 2019(July), July 2019.

[44] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 772–781. IEEE Press, 2013.

[45] Ana CR Paiva, André Restivo, and Sérgio Almeida. Test case generation based on mutations over user execution traces. *Software Quality Journal*, 28(3):1173–1186, 2020.

[46] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. pages 24–36. Association for Computing Machinery, Inc, 7 2015.

[47] Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In *FM 2016: Formal Methods*, pages 593–611, Cham, 2016. Springer International Publishing.

[48] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. pages 180–182. Institute of Electrical and Electronics Engineers Inc., 6 2017.

[49] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology*, 28:1–29, 9 2019.

[50] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. How to design a program repair bot?: Insights from the repairnator project. pages 95–104. IEEE Computer Society, 5 2018.

[51] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. pages 479–490. Institute of Electrical and Electronics Engineers Inc., 3 2019.

[52] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[53] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43:34–55, 1 2017.

[54] Michał Zalewski. American Fuzzy Lop - White Paper. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016.