From Runtime Failures to Patches: Study of Patch Generation in Production

Thomas Durieux

Supervisors: Prof. Martin Monperrus and Prof. Lionel Seinturier

University of Lille

This dissertation is submitted for the degree of
Doctor of Philosophy in Computer Science

Rapporteurs: Prof. Paolo Tonella at *University of Italian Switzerland* and Prof. Olivier Barais at *University of Rennes 1*

Examinateurs: Dr. Julia Lawall at *Inria* and Prof. Jean-Christophe Routier at *University of Lille*

Collaborateur Industriel: Dr. Youssef Hamadi

University of Lille — Thusday 25th September, 2018

Acknowledgements

I would like to thank everyone who contributed to the realization of this thesis.

First of all, many thanks to my advisors Martin Monperrus and Lionel Seinturier, as well as to Youssef Hamadi for offering me the opportunity to carry out this doctoral work. Thank you, Lionel, for managing the team and offering us a great place to do research. Thank you, Martin, for all the discussions, ideas and advices you gave me during these three years. I am also grateful to the members of my jury: Olivier Barais, Julia Lawall, Jean-Christophe Routier, Paolo Tonella for their time and feedback.

The three years that I have spent at Inria and in the Spirals have been a wonderful experience. A big thanks to all members and former members of Spirals. It was a real pleasure to meet you, work with you, and spend time with you after work. A special mention to the colleagues and co-authors who helped and supported me: Benoit Cornu, Benjamin Danglot, Fernanda Madeiral, Matias Martinez, Gérard Paligot, Simon Urli, and Zhongxing Yu. Many thanks to my Brazilian co-author Marcelo Maia and his students Fernanda Madeiral and Victor Sobreira for our great collaborations. I also want to thank the TCS group of KTH and its members: Benoit Baudry, Simon Bihel, Zimin Chen, Nicolas Harrand, He Ye, and Long Zhang who welcomed me for seven months.

Finally, I wish to thank my friends and family who encouraged and supported me during these years. Without them, this thesis would not have been possible.

Abstract

Patch creation is one of the most important actions in the life cycle of an application. Creating patches is a time-consuming task. Not only because it is difficult to create a sound and valid patch, but also because it requires the intervention of humans. Indeed, a user must report the bug, and a developer must reproduce it and fix it, which takes much time. To address this problem, techniques that automate this task have been created. However, those techniques still require a developer to reproduce the bug and encode it as a failing test case. This requirement drastically reduces the applicability of the approaches since it still relies on humans.

This thesis proposes new patch generation techniques that remove the human intervention for the patch generation. Our idea is to put as close as possible the patch generation in the production environment. We adopt this approach because the production environment contains all the data and human interactions that lead to the bug. In this thesis, we show how to exploit this data to detect bugs, generate and validate patches. We evaluate this approach on seven different benchmarks of real bugs collected from open-source projects. During the evaluation, we are particularly attentive to the number of generated patches, to their correctness, readability and to the time required for generating them. Our evaluation shows the applicability and feasibility of our approach to generate patches in the production environment without the intervention of a developer.

Résumé

Dans le cadre de la gestion du cycle de vie d'une application, la création de correctifs de bugs est une des tâches les plus importantes. Or celle-ci prend aussi le plus de temps, non seulement parce qu'il est difficile de créer un bon correctif, mais également parce que cela nécessite des interventions humaines. Un utilisateur doit en effet signaler le bug et le développeur doit le reproduire et le corriger, processus long et fastidieux. Il existe des techniques qui automatisent cette tâche mais elles exigent toujours une intervention humaine à savoir qu'un développeur crée un test reproduisant le bug, exigence qui réduit considérablement leur applicabilité.

Dans le cadre de cette thèse, nous proposons une nouvelle approche de génération de correctifs qui s'affranchit de cette exigence. Elle repose sur l'idée de rapprocher le plus possible la génération de correctifs de l'environnement de production. En effet c'est celui-ci qui contient toutes les données et toutes les interactions humaines qui mènent aux bugs. Dans cette thèse, nous présentons comment exploiter ces données pour détecter les bugs, comment générer les correctifs et comment les valider, le tout sans l'intervention d'un développeur. Nous évaluons notre approche sur sept jeux différents de correctifs réels provenant de projets open-sources en veillant, entre autres, à être particulièrement attentifs au nombre de correctifs générés, à leur validité ainsi qu'au temps requis pour leur génération. Ces évaluations démontrent l'applicabilité et la faisabilité de notre approche dans la génération de correctifs en production sans l'intervention d'un développeur.

Table of contents

List of Algorithms

Listings

List of figures

List of tables

Chapter 1

Introduction

1.1 Context

We use daily as consumers and professionals more and more smart objects and sophisticated software. Software has been used for decades in our computers, phones, TVs, watches and, more recently, in our fridges, speakers, scales, glasses, cups, mirrors and in many more different products. This diversification of products, platforms, and services increases drastically the amount of source code in production and also the number of software bugs encountered by users. This diversification has a considerable cost for companies that try to maintain the highest quality of service to outperform competitors.

The research communities and companies developed in the last years diverse techniques and tools that aim to reduce this maintenance cost. These techniques can be grouped into three categories: 1) The first category includes resources that help the software developers to understand the program, for example by providing access to the runtime state of the applications, i.e., debugger, or by standardizing the source code for improving its readability, i.e., check style. 2) The second category includes techniques that assist the developers in their tasks, for example, fault-localization techniques guide developers to the parts of the program where they should focus their attention to solve a bug. 3) Finally, there are techniques that automatize some developer tasks, which can be from the simple task of automatizing the test suite execution, i.e., continuous integration, to the complex task of automatizing the bug fixing, i.e., automatic program repair. In this thesis, we focus on this last task: the automatic program repair.

The goal of automatic program repair techniques is to handle automatically the incorrect behavior of an application. In the literature, there are two main types of such techniques: the test-based automatic program repair and the self-healing approaches.

1.1.1 Automatic Patch Generation

The test-based automatic repair techniques use the software test suite as an oracle to determine whether the software behaves correctly. These techniques take a buggy program and a test suite that contains at least one failing test case and produce a new version of the program that makes all the tests passing. In other words, they use the test suite as the specification of the application. The software engineering community introduced in the last few years several different test-based automatic patch generation such as GenProg [1], Semfix [2], Nopol [3], Elixir [4] and Genesis [5]. These techniques are meant to interact with developers by proposing patches to them.

1.1.2 Self-healing Runtime Approaches

The self-healing approaches, on the other hand, directly target running applications and handle the incorrect behavior directly in production. The goal is to enhance the availability and the security of the software system. For example, Assure [6] is a technique based on checkpointing and rollback that provides self-healing capability to the software, and failure-oblivious computing [7] prevents software crashes by anticipating them.

In the following section, we present the problems that we identify and that limit the automatic program repair and more specifically the patch generation.

1.2 Problem Statement

> The problem addressed in this thesis is to reduce the time between the detection of a bug by a user and the creation of a patch that addresses the incorrect behavior.

Valdivia et al. [8] show evidence that it takes a large amount of time to handle bugs even for popular open-source projects. They observed that the median time to fix an issue in Chromium is 16 days for non-blocking issues and 48 days for a blocking one. This metric

is even higher for the other open-source projects considered in their study — such as Eclipse, Mozilla or Open-Office.

This task takes a long time not only because of the creation of the patch itself but also because it requires human interventions. Indeed, the bug fixing task needs three important human contributions: first, that a user reports a bug; second, that a developer reproduces the reported bug; and, finally, that a developer creates a patch that handles the bug. This multi-layer of human interventions delays the creation of the patch.

The current literature on automatic patch generation only focuses on replacing the last human intervention, the patch creation. However, the automatic patch generation techniques still rely on the human intervention to reproduce the bug. Indeed, the current automatic patch generation techniques have all the same characteristic: they rely on the test suite of the application to be able to repair it. A failing test case is used to expose the bug, and the passing test cases are used to ensure that the modifications performed to fix the bug do not introduce new regressions in the application. Consequently, before the patch generation, a developer still has to create a failing test case, which can take days.

With a failing test case, the literature on automatic program repair shows that it is possible to generate patches and even several patches for the same bug [9, 10]. However, a developer is still needed to permanently validate the generated patches before integrating them in the code of the application. The review process is also time-consuming. The automatic program repair techniques should not overload the developers with invalid patches, which could delay even more the repair of the application.

An almost real-time approach for automatic program repair consists in directly handling the bug within the production environment when the bug is detected. Self-healing systems use this approach. It solves the problem of the human interventions, but it also introduces a new problem. It modifies the behavior of the application during its execution and, consequently, may introduce unwanted behavior changes.

To sum up, we identify three sub-problems to address the reduction of human interventions and therefore to speed up the patch generation task. The first problem is to create patch generation techniques that do not rely on failing test cases. The second problem is to filter the patches to only present valid or interesting patches to the developers. The last problem is to mitigate the side-effects of the patch generation techniques in the production environment.

1.2.1 Problem 1: Patch Generation without Failing Test Case

The first problem is that patch generation techniques require a failing test that describes the failing scenario precisely.

As previously mentioned, the current patch generation techniques rely on the test suite of the application and especially on a failing test case to generate a patch. However, this asset has to be created by a developer, and it takes time to reproduce the bug and create a test. This dependence on the developer increases drastically the time needed to produce a patch automatically. The state-of-the-art of automatic patch generation is presented in Section 2.2.

1.2.2 Problem 2: Automatic Patch Validation in Production

The second problem to address is to filter out incorrect patches.

The literature on automatic patch generation shows that it is possible to generate a large number of patches for a single bug [9]. But, unfortunately, it also shows that a significant number of these generated patches are incorrect [10]. The generated patches overfit the test suite of the application and do not generalize its behavior for all possible inputs of the application. In order to be useful for the developer, the generated patches have to be filtered to remove the invalid ones. The literature on patch generation search space and patch overfitting is presented in Section 2.2.3.

1.2.3 Problem 3: Side-effect of Patch Generation in Production

The last problem to address is to handle the potential side-effects that a patch generation technique can have in the production environment.

The state of a running application contains the ingredients required to generate a patch. However, these ingredients are only accessible within the running application. It means that the patch generation techniques have to access the running state of the application as self-healing techniques do. The interaction of the patch generation technique and the state of the application can have negative impacts on the running application, which is not acceptable. The side-effects range from performance overhead to behavior change

and state corruption. These effects can even be worse than the bug itself. The literature on self-healing is presented in Section 2.3.

1.2.4 Summary

To summarize, in order to improve the patch generation, we need: 1) better patch generation techniques that do not rely on a failing test case written by a human, 2) better quality control by discarding invalid generated patches, and 3) better applicability by mitigating the potential side-effects of automatic patch generation techniques that could affect the production environment.

1.3 Thesis Contributions

In this section, we present our contributions that aim to solve the problems presented in Section 1.2.

1.3.1 First Contribution: Runtime-based Patch Generation

The first contribution of this thesis consists of three novel techniques that produce patches based on a buggy execution and its running state. It addresses the first problem presented in Section 1.2, by removing the dependence on a failing test case to generate patches.

The first automatic patch generation technique is a template-based patch generation technique for null pointer exceptions in Java. A template-based automatic program repair technique is a technique that uses a predefined set of patch templates to repair an application, e.g., [11]. We consider null pointer exception since it is a runtime failure, therefore it does not require a failing test case to be able to generate a patch for it. The location of the null pointer exception can be determined directly with the stacktrace.

The second technique is a metaprogramming-based automatic program repair technique. In this context, a metaprogram is a program that can modify itself and change its behavior during its execution. We create a metaprogram specialized in handling null pointer exceptions, which can change its behavior dynamically to handle null pointer exceptions. It does not require a failing test case to reproduce the error.

The third and final technique is a code synthesizer. It is designed to synthesize Java expressions by combining constants, variables, and methods with Boolean and mathematical operators, e.g., &&, ||, +, −. These expressions are used to replace buggy conditions and create missing pre-conditions. This technique is the first code synthesizer of the literature that directly uses the runtime information to synthesize code.

1.3.2 Second Contribution: A Study of Patch Validity of Runtime-based Patch Generation

The next contribution presented in this thesis is an analysis of the repair search space of runtime patch generation techniques. This contribution focuses on the second problem presented in Section 1.2, the patch validation. It provides a better picture on the number of valid patches in the search space to enable the creation of new techniques to select the relevant patches without the asserts of a failing test. The repair search space is defined by Martinez [12] as "*all explorable bug fixes for a given program and a given bug (whether compilable, executable or correct)*". In our contribution, we define the repair search space of null pointer exception bugs.

We answer to the following questions: How many patches are generated and how many valid patches is in the search space? In the previous contribution, we were only interested to see if it was possible to generate patches. These new questions are relevant to address our second problem on patch validity. We need a deep understanding on what makes the repair search space grows and, more importantly, which criteria are important to determine the patch validity.

1.3.3 Third Contribution: Practical Patch Generation in Production

The final contribution of this thesis aims to group all the acquired knowledge of the two first ones to create the first patch generation techniques ever dedicated to the production environment. This final contribution targets the three problems of this thesis. It proposes a new patch generation technique that does not use failing test case. It also proposes a new way to validate the generated patches and provides an architecture to handle the potential side-effects of the patch generation techniques.

We design two automatic patch generation systems: one is dedicated to client-side applications and the other one is dedicated to server-side. BikiniProxy is the first technique, it generates patches for JavaScript applications. The JavaScript client-side applications are by nature distributed to each client and are only used for rendering and user interface (UI) interactions. Consequently, the side-effects of the patch generation techniques are limited to only one client and persistent state is also not involved. This environment opens new opportunities for patch generation, patch validation, and side-effects handling. The technique consists of an HTTP proxy that injects error monitoring code in the web application. Once an error is detected, it is considered as a collective knowledge that will be shared between all the clients. This collective knowledge is then used to patch the buggy JavaScript for the next users. The patch is then evaluated by monitoring the patched application. If the monitoring detects an invalid behavior, the generated patch is discarded and no further users will employ it. A database of patches for the web application is thus constituted and can be used by the developers to fix their web applications permanently.

The second technique, Itzal, produces patches and validates them directly from production traffic. The server-side applications are by nature centralized for the data persistence and all the business processing. It implies that any wrong manipulation impacts several users, and potentially the persistent state of the application. In this context, it is not realistic to directly modify the running application to generate patches. Our solution is to duplicate the application and production traffic in a sandboxed environment that can be used by the patch generation techniques. The sandboxing prevents all the potentials side effects that the patch generation techniques can have on the production application. Another characteristic of Itzal is that it uses the sandboxed environment to deploy the generated patches and to compare its behavior with the production application. This way Itzal is able to assert the validity of the patches.

1.4 Outline

The remainder of this dissertation is composed of five chapters as follows:

Chapter 2: State of the Art

This chapter presents the most relevant works on program monitoring and analysis, automatic program repair, and self-healing techniques. We classify these approaches according to their purpose and usage.

Chapter 3: Runtime Approaches for Automatic Patch Generation. This chapter presents three new automatic patch generation techniques that use the application runtime state to generate patches. This chapter is a revised version of the following papers:

Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 349–358. IEEE, 2017

Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST) colocated with ICSE*, pages 85–91. ACM, 2016

Chapter 4: A Study of the Runtime Repair Search Space. In this chapter, we characterize the repair search space of an automatic patch generation technique. We collect the size, the validity and the time required to explore the repair search space exhaustively. This chapter is a revised version of the following paper:

Thomas Durieux, Youssef Hamadi, Zhongxing Yu, and Martin Monperrus. Exhaustive exploration of the failure-oblivious computing search space. In *Proceedings of the 11th International Conference on Software Testing, Validation and Verification (ICST)*, pages 139–149. IEEE, 2018

Chapter 5: Contributions to Automatic Patch Generation in Production. This chapter presents two novel automatic patch generation techniques designed for the production environment. The first technique is dedicated to client-side applications and the second technique is designed for server-side applications. This chapter is a revised version of the following papers:

Thomas Durieux, Youssef Hamadi, and Martin Monperrus. Fully automated HTML and JavaScript rewriting for constructing a self-healing web proxy. *Under submission, arXiv:1803.08725*, 2018

Thomas Durieux, Youssef Hamadi, and Martin Monperrus. Production-driven patch generation. In *Proceedings of the 37th International Conference on Software Engineering (ICSE), track New Ideas and Emerging Results*, pages 23–26. IEEE, 2017

Chapter 6: Conclusion and Perspectives This final chapter summarizes this thesis and discusses its contributions, impact, limitations, and perspectives.

1.5 Publications

Published

- Thomas Durieux, Youssef Hamadi, Zhongxing Yu, and Martin Monperrus. Exhaustive exploration of the failure-oblivious computing search space. In *Proceedings of the 11th International Conference on Software Testing, Validation and Verification (ICST)*, pages 139–149. IEEE, 2018
- Thomas Durieux, Youssef Hamadi, and Martin Monperrus. Production-driven patch generation. In *Proceedings of the 37th International Conference on Software Engineering (ICSE), track New Ideas and Emerging Results*, pages 23–26. IEEE, 2017
- Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 349–358. IEEE, 2017
- Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST) colocated with ICSE*, pages 85–91. ACM, 2016

Under Submission

- Thomas Durieux, Youssef Hamadi, and Martin Monperrus. Fully automated HTML and JavaScript rewriting for constructing a self-healing web proxy. *Under submission, arXiv:1803.08725*, 2018
- Thomas Durieux, Zhongxing Yu, Youssef Hamadi, and Martin Monperrus. Automatic patch synthesis and validation in production. *Under submission, arXiv: 1609.06848*, 2018

Collaborations

- Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Test case generation for program repair: A study of feasibility and effectiveness. *Empirical Software Engineering (EMSE)*, 2017
- Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140. IEEE, 2018
- Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering (EMSE)*, 2016
- Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *Transactions on Software Engineering (TSE)*, 2016

Chapter 2

State of the Art

This chapter reviews the works that are closely related to the topic of this thesis. We identify three main related research fields: program monitoring and analysis (Section 2.1), automatic program repair (Section 2.2) and self-healing software (Section 2.3). Figure 2.1 graphically summarizes those three research fields and their subsections.

2.1 Program Monitoring and Analysis

We start by presenting the publications on the topic of monitoring and analysis of program behavior.

2.1.1 Program Monitoring

In this first subsection, we discuss the state of the art of runtime monitoring techniques. These works are related to the second problem of this thesis: to be able to automatically validate patches in the production environment (see Section 1.2). Runtime monitoring consists of collecting data from a running application in order to monitor its behavior. The techniques of this section can also be applied on patched application to monitor the changes and therefore get insights on the validity of the patch. In this section, we present an overview of monitoring techniques that inspired us to create automatic patch validation techniques. The monitoring techniques are grouped according to the program property that they target: fault, security, and performance.

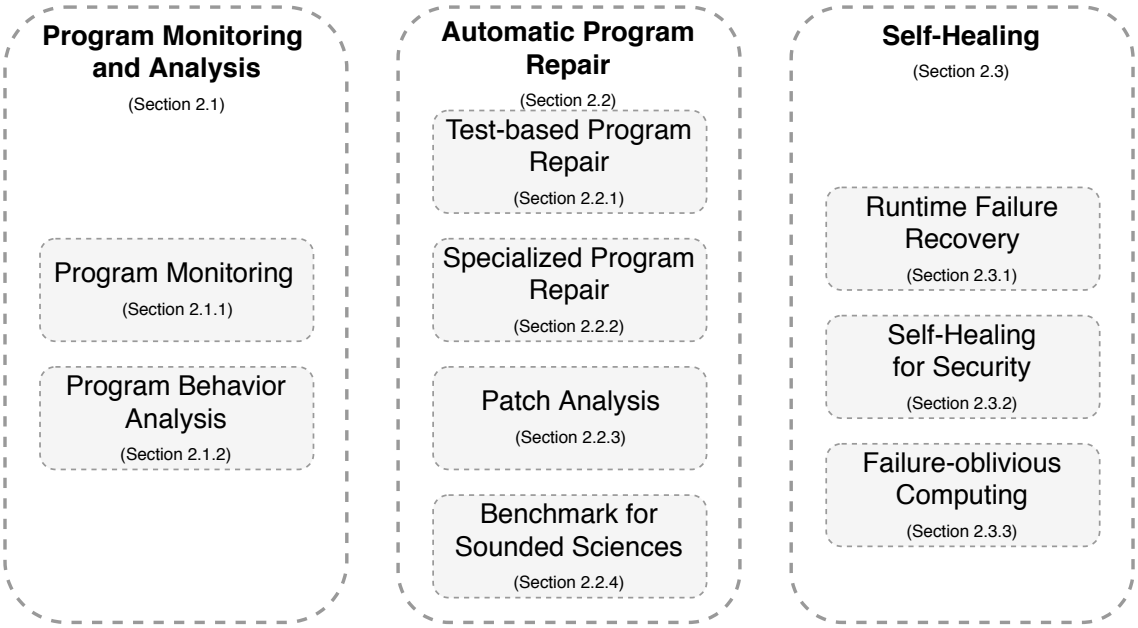

Figure 2.1 Research areas related to this thesis.

This related work has no vocation to be exhaustive, for an exhaustive review of monitoring techniques, we refer you to Delgado et al. [20] survey. It presents a large picture of the different techniques that monitor faults in applications. This survey also has the particularity to classify the techniques depending on the type of system they target. This is valuable information to understand how the different systems can be monitored. One example of fault monitoring techniques is MonitorRank [21]. It monitors service-oriented applications to detect the root cause of a failure. They achieve this goal by tracking the different monitoring point that has been triggered before the failure event. They succeed to create the patch of execution between each service. This technique could also be used to characterize the interaction between the different services of an application and tracking down incorrect interaction.

Monitoring techniques are also largely used for detecting security intrusions. Salamat et al. [22] and Kwon et al. [23] have an interesting approach: they compare parallel executions of the same application to detect information leaks and attacks. Magazinius et al. [24] focus on how to inject security monitoring techniques in JavaScript client-side applications. Those techniques explore new ways to detect and monitor applications that can also be used to design patch validation techniques.

A large section of this literature focuses on performance monitoring of production applications. van Hoorn et al. [25], present Kieker a generic performance monitoring tool that

is designed to monitor the performance of Java and .NET applications. Other techniques specialize on performance monitoring of specific areas. For example, Appinsight [26] is performance monitoring for mobile applications. It tracks the user's actions to determine which sequence of operations slow down the application. AjaxScope [27] is another example of performance monitoring techniques, but this one is designed for JavaScript applications. Performance monitoring techniques provide valuable knowledge on how to instrument application without affecting the performance. And the performance data for the tools could also be used as a metric to identify invalid patches.

2.1.2 Program Behavior Analysis

We now present an overview of program behavior analysis. We only consider the works that do runtime analysis of the behavior of running applications, since we are mainly interested in the program during its execution. Recall that understanding the program behavior is the key to address the two first problems of this thesis: generate patches without failing test case and assess the validity of the patches. The generation of valid patches relies on the understanding of the application behavior. And patch validation is a technique that asserts that the patch does not introduce incorrect behavior in the application. The works in this subsection are sorted according to the purpose of the behavior analysis.

Marceau [28] uses a n-grams derived technique (finite state machine) to characterize the behavior of an application. In this work, he shows that this technique is able to detect when the application is under-attacks by comparing the finite state machine of a nominal execution to the current execution. Locasto et al. [29] also use an n-gram representation to characterize the behavior of the application. But they use it to detect anomalies in applications after self-healing. Their analysis shows that it is possible to detect behavior deviation due to self–healing repairs that modify program state. Those techniques show that it is possible to characterize the behavior of the application to detect behavior change and can consequently by use to detect behavior changes due to a code modification.

Sherwood et al. [30] have a different perspective on behavior analysis. They use it to predict future behavior of the application and optimizes its execution. They implemented the idea in the tool SimPoint [30], they achieve this goal by automatically clustering the executions of production applications into phases. It is then possible to predict the next phase and the associated resources. The loading of the resources can be optimized. The

prediction of the next phase can be used to detect behavior change if the next phase is not the one predicted it means that potentially something wrong happened.

The next study by Wang et al. [31] compare the behavior of the production application with the behavior of the application under test. They observe that the tests are not representative of the production execution and that the generated tests suffer from the same problem. It is an interesting finding for the patch validation, it means that the production traffic is complementary to evaluate the behavior and characterize the behavior of the applications.

And finally, Candea and Fox [32] study the behavior of production applications and observe that the applications behave differently after a crash compared to a normal reboot. Therefore, they recommend that the developers should design their application by considering that the normal behavior is to crash. This improves the reliability and the availability of production applications. This is an important feature for patch generations techniques since it reduces the impact on the system of the generated patch does not have the expected behavior.

2.1.3 Conclusion

We presented the related work about monitoring and analysis of program behavior in production. Among this related work, there are two works that are particularly interesting to address our problems: patch generation without failing test case and handling the side-effects of patch generation. The first work, *Behavioral Execution Comparison: Are Tests Representative of Field Behavior?* by Wang et al. [31] observes that the behavior of the application is different in the test environment and in the field. This suggests that the state of a production application can be more valuable than the state triggered by the tests. In this thesis, we leverage this idea in the context of patch generation for JavaScript errors (see Section 5.1). The second work, *LDX: Causality Inference by Lightweight Dual Execution* by Kwon et al. [23] presents an approach that uses a dual execution to detect information leaks. This approach is an interesting direction for separating the patch generation from the production application. We use this idea in Itzal, a patch generation technique that uses dual executions to generate patches (see Section 5.2).

2.2 Automatic Program Repair

Automatic program repair is a task that aims to change automatically the behavior of a program to fix an invalid behavior. This section presents several aspects of automatic program repair. First, we present the test-based automatic program repair techniques. Second, we present techniques that repair specific aspects of an application, such as repairing inconsistencies in a data structure. Third, we present the literature on analyzing generated patches. Finally, we list the benchmarks of bugs that are used in the evaluation of patch generation techniques.

There are two surveys by Gazzola et al. [33] and Monperrus [34] that summarize the literature of automatic program repair techniques. We refer to these two surveys for an exhaustive summarization of existing works on such field. In this related work section, we focus on the works that are the most relevant to patch generation without failing test cases.

2.2.1 Test-based Automatic Program Repair

Test-based automatic repair is an automatic program repair technique that uses the test suite of the program as its specification. It means that the passing test cases specify the correct behavior of the program and at least one failing test case describes the invalid behavior of the program. The goal of test-based repair techniques is to generate a patch that makes the entire test suite to pass. In the literature, there are three main approaches to generate patches: the search-based approach, the deductive-based approach, and the template-based approach. These works are state-of-the-art for the patch generation techniques that we present in this thesis.

2.2.1.1 Search-based Program Repair Techniques

Search-based repair technique consists in searching for patches in a search space of patch candidates. We present the most important search-based program repair techniques in chronological order.

Arcuri et al. [35] present Jaff, the first approach that uses evolutionary optimization for program repair. Debroy and Wong [36] use the mutation operators of mutation testing as repair strategies. This work combines fault localization with program mutation to exhaustively explore a space of candidate patches. GenProg [1] introduces the concept

of genetic programming for patch generation. It generates patches by adding, deleting, or replacing elements of the code with existing code in the application. AE [37] is optimized GenProg by reducing its repair search space. RSRepair [38] uses a random search, and it is more efficient in finding patches than the genetic programming approach. Their follow-up work [39] prioritizes test cases to reduce the time to detect invalid patch candidates.

SPR [40] defines a set of staged repair operators. It discards as early as possible candidate patches that cannot make the test suite a passing one. Consequently, SPR has more time to explore a smaller and more valuable patch search space. ELIXIR [4] focuses on the generation of patches that contain method calls. They rank the candidate patches with machine learning techniques to filter the patches and to reduce the search space. They extract the machine learning features from the context of the buggy statement. CapGen [41] is a patch generation technique that uses the context of the buggy statement to rank the potential candidates of patches. Their results show that a context-based approach is a valuable technique to address the patch overfitting problem.

2.2.1.2 Deductive-based Program Repair Techniques

Deductive-based repair technique consists of inferring the correct behavior of the application and then synthesizing a patch that has the behavior assumed as correct.

SemFix [2] is the pioneering work in deductive-based approach. It extracts the expected behavior of the application with symbolic execution. Then, it encodes the extracted behavior as a constraint problem. The solution to this problem is a patch that has a required behavior to make the failing test case passing. Nopol [42, 3] also uses a constraint-based problem to generate patches. It introduces the concept of angelic value to extract the expected behavior of a condition to make the failing test case passing. DirectFix [43] aims to generate the simplest patch to fix buggy expressions. It encodes the repair problem into a partial Maximum Satisfiability problem (MaxSAT). Then, it uses a Satisfiability Modulo Theory (SMT) solver to convert the satisfiability problem into a concise patch. Angelix [44] uses a lightweight repair constraint representation called “angelic forest” to increase the scalability of DirectFix. The angelic forest allows Angelix to generate multi-line patches.

2.2.1.3 Template-based Program Repair Techniques

Template-based technique uses predefined patch templates to fix bugs. The main advantage of this technique is the readability of the generated patches since the patch templates are generally inspired by human patches. The humans are consequently used to these patches. Consequently, it is easy for humans to read them and assess the validity of these patches.

PAR [11] is the first template-based repair work proposed in the literature. It uses ten different patch templates for common programming errors, such as adding a missing null check. Relifix [45] uses the same approach than PAR but it targets different types of bugs. It uses templates that are designed to fix regression bugs. Kern and Esparza [46] presents the second work based on template proposed in the literature. The idea of this work is the following: a developer describes which part of the project can be buggy using a code template; for example, the template $i < N$ describes all the expressions that contains i lower than something. With such template, the developer specifies some suitable alternatives, for example, $i < N + 1$. The alternatives are then selected during the program execution to find a valid patch. The advantage of this approach is that it allows the generation of multi-point patches.

Xiong et al. [47] design their tool, ACS, to repair buggy conditions. They analyze the context of the failing test case to determine which template to use. For example, it extracts the expected exception that the application should throw from the failing test case. Prophet [48] and Genesis [5] improve the previous techniques by automatically learning the patch templates from successful human patches. These techniques target more types of bugs and consequently improve the repair success rate. SOFix [49] also automatically extracts patch template from human code. It uses StackOverflow website as the source of the generated templates.

2.2.2 Specialized Program Repair Techniques

The literature also contains automatic program repair techniques that target specific types of bugs. We present in this section the most relevant of these works in chronological order.

AutoFix-E [50] is an automatic repair technique that uses the pre and post-conditions present in Eiffel applications as the specification of the application. It uses these contracts to create code modifications that validate the pre and post-conditions. AFix [51] is an

automatic program repair technique that is designed to repair common concurrency bugs. It uses static analysis to detect and repair atomicity-violations.

Samirni et al. [52] presents two tools to repair PHP bugs that are related to HTML generation. The first tool, PHPQuickFix, is designed to repair simple PHP code by statically analyzing the generated HTML code. On the other hand, PHPRepair targets more complex bugs, and it uses the test cases that assert the generated HTML to guide the patch generation. Ocariza et al. [53] present Vejovis, a technique that suggests JavaScript code modifications to handle DOM-related JavaScript bugs. The suggestions are based on seven strategies, including four that are relative to loop.

Gopinath et al. [54] present an approach that suggests potential fixes for database applications. They determine the correct behavior by using a combination of SAT-based search and prediction generated by support vector machines. MT-GenProg [55] is a tool that uses the same core idea as GenProg for the generation of the patches. The difference is that MT-GenProg uses metamorphic tests instead of traditional test case like GenProg uses.

2.2.3 Analysis of Generated Patches

In this section, we present existing works in the literature that analyzes the patches generated by automatic program repair tools. These studies are significant for this thesis because they discuss on the limitations of the current patch generation techniques. It is a precious knowledge to design new patch generation techniques, which is the first target problem of this thesis (see Section 1.2.1). It is also essential to understand how humans validate patches. This provides us ideas on how to automatize this process, which is the second target problem of this thesis (see Section 1.2.2).

We first present the studies that analyze maintainability and usability of the generated patches for the developers. Then, we present studies that compare different repair techniques based on benchmarks. Finally, we present works on patch overfitting.

Fry et al. [56] conduct a study to determine the maintainability of generated patches. They asked 150 participants to evaluate the maintainability of generated patches for 32 real-world bugs. They show that generated patches are slightly less maintainable than human-written ones. Tao et al. [57] perform a similar study to determine that generated patches can be used to assist human during the debugging phase.

Few studies compare the repairability of patch generation techniques on a common benchmark. Kong et al. [58] compare four repair techniques for C language: GenProg [1], RSRepair [38], Brute-force-based repair [59], and AE [37] on Siemens [60] and SIR [61] benchmarks. Martinez et al. [9] perform a similar comparison but between three different automatic repair techniques for Java language: Nopol [3], jGenProg [62], and jKali [62] on Defects4J benchmark [63]. In this study, they point out the overfitting problem of these techniques.

Several recent studies show that overfitting is a serious problem associated with test suite-based automatic program repair techniques. Long and Rinard [64] show that it is common to have many equivalently correct yet syntactically different patches in the search space of automatic patch generation techniques. The search space contains mostly invalid patches. Qi et al. [65] find that the vast majority of patches produced by GenProg [1], RSRepair [38], and AE [37] avoid bugs simply by removing functionalities. A subsequent study by Smith et al. [66] confirms that the patches generated by GenProg and RSRepair do not generalize its behavior. The empirical study conducted by Martinez et al. [9] reveals that among the 47 bugs fixed by jGenProg [62], jKali [62], and Nopol [3], only nine bugs were correctly fixed.

More recently, the study by Le et al. [67] confirms the severity of the overfitting problem for deductive-based repair techniques. They also study how the test suite size, provenance, and number of failing tests can affect the overfitting problem for synthesis-based repair techniques. Yi et al. [68] dedicate a complete survey on the correlation between test suite metrics and the quality of patches generated by automated program repair techniques. They find that with the increase of traditional test suite metrics, the quality of the generated patches tends to improve. Yu et al. [10] present a technique that generates additional test cases to try to reduce the overfitting problem. Xin et al. [69] also use test case generation to minimize the overfitting problem. The difference with the previous study is that they use the test generation technique to cover the behavior difference between the buggy and the patched program version. This way they ensure that the generated test case covers new inputs. The limitation of the approach is that it requires a manual validation to ensure that the generated test does not specify a buggy behavior. Jinqiu et al. [70] propose an automatic approach to validate the generated test cases. They validate them by monitoring inherent oracles, such as crashes and memory-safety problems. Liu et al. [71] exploit the behavior similarity of the test cases to generate new test cases that behave similarly to the passing tests, and that can be used to assess the validity of patches.

2.2.4 Benchmark for Automatic Program Repair

Benchmarks of bugs are important assets for automatic program repair. These benchmarks are relevant because they provide a reproducible environment to evaluate repair approaches, and they also allow direct comparison between techniques. We present the benchmarks in chronological order.

iBugs [72] is a benchmark for bug localization obtained by extracting historical bug data. BugBench [73] is a benchmark that has been built to evaluate bug detection tools.

Other benchmark includes a failing test case that reproduces the bugs. Defects4J [63] is a bug database that consists of 357 real-world bugs from five widely-used open-source Java projects. Despite the fact this benchmark was first proposed to the software testing community, it has been used for several works on automatic program repair. Le Goues et al. [74] proposed the first benchmarks dedicated to the automatic program repair community. It includes IntroClass, a benchmark of small C programs written by students at the University of California Davis. The programs consists of 5-20 lines of code, usually in a single method, and are specified by a set of input-output pairs. We later propose IntroClassJava [75], a transpilation of IntroClass to Java.

Due to the lack of benchmarks of bugs in the automatic program repair field, three benchmarks have been recently proposed in the literature. Codeflaws [76] consists of 3902 defects extracted from programming competitions and contests available on Codeforces [77]. These defects are from C programs, and the programs range from 1 to 322 lines of code. QuixBugs [78] contains single line bugs from 40 programs, which are translated into both Java and Python languages. This is the first multi-lingual program repair benchmark. Bugs.jar [79] is dedicated to Java language. It consists of 1,158 bugs from 8 apache projects. The bugs were extracted by the identification of bug-fixing commits with the support of Jira issue-tracking system and the execution of tests on the bug-fixing program version and its reverse patch (buggy version).

In this thesis, we also propose new benchmarks of bugs: a benchmark of real null pointer exceptions (Section 3.1.4.3) and a benchmark of JavaScript production failures (Section 5.1.3.2). These benchmarks fill a gap in the literature, and we also reused existing benchmarks to compare our work to the literature.

2.2.5 Conclusion

In this section, we have presented the related works on automatic patch generation techniques. The take away of this section is that the overfitting problem on patch generation has an impact on the ability of this type of repair technique. However, it seems that specialized techniques are less impacted by this problem since they target specific cases. We explore this idea of specialized technique in NPEFix (see Section 3.1).

Another take away of this section is that all the techniques that suffer overfitting use the test suite of the application as specification. Wang et al. [31] show that the test suite is not a good representation of the production behavior, which may explain the overfitting problem. In Chapter 5, we present two patch generation architectures that use the production inputs for generating patches.

2.3 Self-healing

In this last section of the related work, we present the literature on self-healing techniques.

The idea of those techniques is to inject self-healing capabilities in the application. The strategies consist of modifying the application state to prevent and handled incorrect behaviors. This literature is related to this thesis since it changes the production application state to handle invalid behavior. This literature provides us the knowledge to identify the key point in the application state that we should consider. This knowledge helps us to design new patch generation techniques that exploit the production state to generate and validate patches.

2.3.1 Runtime Failure Recovery

We now present self-healing techniques that provide automated failure recovery to applications. These works are used in this thesis to understand how to modify applications to handle invalid behavior which is related to our first problem of patch generation without a failing test-case.

One of the earliest self-healing technique is presented by Ammann and Knight [80]. The idea of this paper is to replace the input data in the application by a crafted input when a failure occurs. This goal is that the new input slightly changes the behavior of the application in order to recover from the failure. Demsky et al. [81] focus on the data

consistency of the application. They present a technique that checks the constancy of the data structure during the execution. If the constancy of the data is broken, the technique modifies it until the consistency is fixed.

Locasto et al. [82] aboard the problem of self-healing with a different angle, they propose to use a community of applications that share failure information. The idea is to propagate the information and make all the application able to handle a failure. Each application generates its own fix the handle the failure, which increases the diversity and the failure resilience among the applications. Tallam et al. [83] present a self-healing technique that targets generic failure and handles them with safe execution perturbations. They monitor the application to detect failures and inject checkpoint to replay the failure execution. They replay the execution with some three different kinds of perturbations. They design the perturbations not to modify the semantics of the application. Carzaniga et al. [84] handle web applications failure with a set of manually written, API-specific workarounds. In subsequent work, the same group has proposed a way to mine those workarounds [85] automatically.

Jula et al. [86] create a self-healing technique that is specifically designed to handle deadlocks. They achieve this goal by monitoring the application to detect deadlock. Once a deadlock is detected, they inject additional synchronization lock in the application to prevent further deadlock. Kling et al. [87] also propose a technique that addresses an infinite execution state. They present, Bold, a self-healing technique that detects and breaks infinite loops. They use two different strategies to stop the infinite loop, they either beak the loop or exit the method execution. They use a checkpoint-and-rollback system to try the strategies and restore the state if the strategy fails.

Rx [88], Assure [6] and ARMOR [89, 90] are also techniques that use the checkpoint-and-rollback technique. However, they use the checkpoint-and-rollback to restore the state before the failure. The three contributions target different types of failures and different strategies to handle them. ClearView [91] also uses checkpoint-and-rollback technique to restore the application state before the crash by they use an advanced technique to determine when was the previous valid state. They learn application invariants by monitoring the access of a low-level registry. The idea of ClearView is to propose a platform that supports different self-healing techniques. Gu et al. [92] design their self-healing technique to have a lower overhead on the application compared to checkpoint-and-rollback techniques. They design, Ares, a technique that handles Java exception by injecting and modifying the try catch of the application.

Qin et al. [93] exploit the hardware components to monitor the system without impacting the application performance. They use the ECC-memory to detect illegal memory accesses (such as buffer overflow) at runtime. Papavramidou et al. [94] also use the ECC-memory to recover from a failure. They present an algorithm that trades latency and costs to handle memory corruption in high defects densities environment.

2.3.2 Self-Healing for Security

Self-healing techniques are also used to protect the application against vulnerabilities. Those techniques rely on different aspects of the state of the application to detect them and put in place countermeasures.

Sidiroglou et al. [95] present a self-healing architecture to protect the application against worms that cause for example buffer overflows. This architecture consists of sending suspicious requests to a sandboxed version of the application. In this sandboxed environment, the application is instrumented to detect the worms. Once a worm is detected, the application is modified to take care of it. Then, the modifications are validated with the test suite to ensure that the modifications do not change the application behavior. Once validated, the modifications are sent to the production application to enhance its security.

CAWDOR [96] is also a self-healing technique that protects the application against worms. The idea of CAWDOR is to crowdsource the detection of the worms on different hosts. All the hosts have the same instrumented application but instead of activated all the self-healing points, CAWDOR only actives some specific points on each host. Once a host detects a worm, it informs the other hosts of the presence of worms that attacks a specific section of the code. The other hosts can then their worm's defense at that specific location. That way the application is protected and the overhead on the application is mitigated. Berger and Zorn [97] adopt in DieHard a different technique to increase the security of the application. They randomize the memory allocation and provide memory replication to increase the memory safety.

Exterminator [98] provides more sophisticated self-healing ability than DieHard [97]. They perform fault localization before applying memory padding to reduce the overhead on the application. Son et al. [99] are interested in a different aspect of the security. With Fix Me Up, they present a technique that automatically repairs access-control in PHP Web applications. It ensures that only authorized users can access restricted sections of the application.

More recent works consider self-healing technique for mobile applications. Appsealer [100] is a technique that prevents components hijacking in Android mobile application. Firstly, they perform a static analysis of the bytecode to identify program the slices that can lead to the component hijacking. Then, they monitor each suspicious slice to detect malicious usage and prevent it. Self-healing techniques can also be used in a nonbusiness oriented system. For example, Appelt et al. [101] present a self-healing technique for firewalls that updates the blacklist rules to prevent SQL injection. They use machine learning and generic algorithm to determine which traffic is malicious and construct the rules to ban it.

2.3.3 Failure-Oblivious Computing

In this section, we present the few works that use the failure-oblivious computing approach. The idea of this approach is to detect and prevent a failure from occurring. This approach is different from traditional self-healing techniques since the failure-oblivious computing prevents the failure before it happens instead of handling the failure afterward.

There is a link, completely unexplored, between failure-oblivious computing and patch generation that we explore in this thesis. There is still a significant difference between failure-oblivious computing and automatic program repair. Failure-oblivious computing handles the bug by modifying the state of the application. The automatic program repair handles the bugs by modifying the code. Failure-oblivious computing is really interesting for this thesis because this technique does not require a failing test case to handle failures. It is what we want to achieve with a patch generation technique.

Rinard et al. [7] are the first to present the concept of "failure-oblivious computing". They use this concept in a technique that avoids illegal memory accesses by injecting their framework in each memory operation. The framework is then able to detect when a memory operation fails and prevent the failure by applying a self-healing strategy. Dobolyi and Weimer [102] adopt the same idea of failure-oblivious computing but adapt it for null dereference in Java. They apply source code transformation to insert their framework where a null dereference can occur. When a null dereference is detected, the framework is then responsible for creating a default object to replace the null dereference. Kent [103] also proposes a similar technique to handle null dereferences but he suggests different strategies to manage them. He proposes to skip the line that contains the null dereference or to stop the execution method by returning a crafted value.

Long et al. [104] present a more sophisticated technique that handles null dereferences and division by zero. The technique consists of replacing the expression that causes the failure with a manufactured value. The critical difference with the previous works is that they track the manufactured values to know if it is passed to system calls or files or if it disappears during the execution of the application. Keeping track of the manufactured values is important to ensure that the value is not stored in the state of the application.

A recent contribution by Rigger et al. [105] presents a failure-oblivious technique that handles buffer overflow. The failure-computing strategy consists of computing the correct size during the creation of the buffer. They mitigate the performance overhead of the technique by using a static analyzer to determine the potential location of buffer overflow.

2.3.4 Conclusion

In this chapter, we presented the literature on self-healing systems. We observe that this literature is mainly based on two techniques to offer self-healing capabilities: monitoring and checkpoint-and-rollback techniques. The monitoring is used to detect the errors or to ensure that the recovery mechanism did not break the application behavior. Moreover, the checkpoint-and-rollback technique is used to restore and try different strategies. Monitoring and mitigating the side-effects of the recovery system seems to be the key features to create a repair system. We use these features in Chapter 5, to create two new patch generation techniques in production.

Among all this literature, the sandboxing technique of Sidiroglou et al. [95] seems particularly interesting since it mitigates the side effects on the production environment without introducing the overhead of checkpoint and rollback techniques. We design a patch generation technique that also uses a sandboxing environment to try code modifications Section 5.2.

The crowdsourcing approach of Yuan et al. [96] also opens new research directions to crowdsource the detection of bugs. This crowdsourcing could also be used for validating the patches among several hosts and several users. We present BikiniProxy (Section 5.1), a patch generation technique for internet applications. It uses the distributed nature of JavaScript to crowdsource the detection and the validation of patches among users.

Chapter 3

Runtime Approaches for Automatic Patch Generation

Automatic patch generation consists of introducing changes in the code of a buggy program to fix a bug. The existing automatic patch generation techniques use the test suite of the application to generate patches, such as GenProg [1], Semfix [2], Nopol [3], Elixir [4], and Genesis [5]. More specifically, at least one failing test case is used to describe the incorrect behavior of the application. The passing test cases are used to ensure that the generated patch does not break other features of the application. This approach is difficult to be used in practice since it involves developers to create at least one failing test case, which decreases the automatic aspect of the techniques drastically.

This chapter presents three novel automatic patch generation techniques that generate patches from a crashing execution. Our hypothesis in this chapter is that there is no need of a failing test case created by a developer to be able to reproduce a crashing execution. The first technique, presented in Section 3.1.2, is a template-based automatic patch generation technique. It applies predefined patch templates that mimic human behavior at the location of the crash. We create this technique to generate patches that are close to the human behavior, therefore they are easy to read and understand.

The second technique, presented in Section 3.1.3, uses a metaprogramming approach to generate patches for null pointer exceptions. This technique takes as input a program and produces a new version of the program containing null pointer exception handlers directly embedded in the program. Our metaprogramming approach embeds nine different strategies for generating patches with null pointer exception handlers, and it selects which strategy to use at runtime (i.e. during the execution of the program). This

approach optimizes the generation of the patches since all the strategies are included in the application and have direct access to the state of the application to select the best strategy.

The third technique, presented in Section 3.2, is a code synthesizer that uses the state of the application as ingredients for patch generation. It collects the state via the debug interface of the application and synthesizes Java expressions by combining constants, variables, and methods with Boolean and mathematical operators, e.g., &&, ||, +, -. The generated expressions are then used in patches for fixing buggy conditions and missing pre-conditions.

Those three techniques share a common characteristic: they use the state of the running application and their design only requires to have access to the invalid execution to generate patches.

In the evaluation of these techniques we are interested in three important metrics: 1) the ability of the techniques to generate patches for real bugs; 2) the proportion of valid patches from the generated patches; and 3) the amount of time that the technique spends to generate patches.

For the two first techniques that target null pointer exceptions, we create a benchmark of real null pointer exceptions by looking at the Apache issue tracker. For the final technique, we use the Defects4J benchmark [63] and 224 of its real bugs from four open-source projects.

The remainder of this chapter is organized as follows. Section 3.1 presents the two patch generation techniques that are dedicated to null pointer exceptions. Section 3.2 presents DynaMoth, a code synthesizer for Java. Finally, Section 3.3 synthesizes our findings on patch generation.

This chapter contains material published in the proceeding of AST'16 [14] and SANER'17 [13].

3.1 Automatic Patch Generation for Null Pointer Exception

This section presents the two first patch generation techniques which generate patches for null pointer exception bugs.

Null pointer exceptions are the number one cause of uncaught crashing exceptions in production [106]. Li et al. found that 37.2% of all memory errors in Mozilla and Apache are due to null values [106]. Other studies[1] found that up to 70% of errors in Java production applications are due to null pointer exceptions. It is an inherent fragility of software in programming languages where null pointers are allowed, such as C and Java. A single null pointer can make a request or a task to fail, and in the worst case, can crash an entire application.

One way of fixing null pointer exceptions is to use a template [11]. For instance, one can reuse an existing local variable as follows:

```
1 + if (r == null) {
2 +   r = anotherVar;
3 + }
4   r.foo(p);
```

This example illustrates a template that depends on the context (called parametrized template in this contribution): the reused variable "anotherVar" is the template parameter. When repairing a statically typed language, such as Java that we consider in this contribution, the static type of the template parameter has to be compatible with the variable responsible for the null pointer exception. This is done with static analysis of the code. The resulting technique is called TemplateNPE.

However, there may be other variables in the context of the null pointer exception, typed by a too generic class, which may be compatible. *The static analysis of the context at the location of the null pointer exception may not give a complete picture of the objects and variables that can be used for fixing a null pointer exception.* There is a need to make a dynamic analysis of the repair context.

We propose to perform a dynamic analysis of the repair context. This completely changes the way we perform patch generation: instead of applying templates one after the other, we propose to create a metaprogram that is able to dynamically analyze the runtime context and to dynamically alter the execution of the program under repair to identify a working patch. The resulting technique is called NPEfix.

This contribution is structured as follows. Section 3.1.1 presents a taxonomy of repair strategies for null pointer exceptions. Section 3.1.3 describes our metaprogramming approach to apply those repair strategies. Section 3.1.4 discusses the evaluation of two

[1] http://bit.ly/2et3t79

Table 3.1 Repair strategies for null pointer exceptions.

Strategy			Id	Description
replace	reuse	local	S1a	local reuse of an existing compatible object
		global	S1b	global reuse of an existing compatible object
	creation	local	S2a	local creation of a new object
		global	S2b	global creation of a new object
skipping	line		S3	skip statement
	method	null	S4a	return a null to caller
		creation	S4b	return a new object to caller
		reuse	S4c	return an existing compatible object to caller
			S4d	return to caller (void method)

patch generation techniques. Section 3.1.5 explores the limitations of our approach. And Section 3.1.6 concludes.

3.1.1 A Taxonomy of Repair Strategies for Null Pointer Exceptions

This subsection presents a taxonomy of run-time repair strategies for null pointer exceptions. It unifies previous work on that topic [102, 103, 107] and proposes new strategies.

When a harmful null pointer exception is going to happen, there are two main ways to avoid it. First, one can replace the null value with a valid object. Second, one can skip the problematic statement. In both cases, no null pointer exception happens. In this section, we refine those two techniques in 9 different strategies to repair null pointer exceptions. There are grouped along those two families: replacement of the null by an alternative object, and skipping the execution of the statement associated with the null pointer exception.

3.1.1.1 Strategies Based on Null Replacement

One way to avoid a null pointer exception to happen is to change the reference into a valid instance. We can provide an existing value (if one can be found) or a new value (if one can be constructed). To facilitate the presentation, r is an expression (usually a variable reference). We basically want r to reference a valid (non-null) value in order to prevent a null pointer exception when executing r.foo(p). Symbol p is a method parameter.

Reuse (S1b) A first strategy is the case of repairing the null pointer exception with an existing object as follows:

```
1 + if (r == null) {
2 +   r = anotherVar;
3 + }
4   r.foo(p);
```

Strategy S1b is parameterized by the variable $anotherVar$. This variable is taken from the set S of accessible objects composed of the local variables, the method parameters, the class fields of the current class and all the other static variables. S is further filtered to only select the set of all well typed and non-null values V according to the type of r.

Local Reuse (S1a) A variant of the reuse strategy consists of replacing one null reference by a valid object, without modifying the null variable itself.

```
1 + if (r == null) {
2 +   anotherVar.foo(p);
3 + } else {
4     r.foo(p);
5 + }
```

With local reuse, all the other statements using r will still perform their operations on *null*.

Object Creation (S2b) Another strategy consists of creating a new value.

```
1 + if (r == null) {
2 +   r = new Foo();
3 + }
4   r.foo();
```

Object Creation Local (S2a) A rare possible patch for a null pointer exception consists of providing a disposable object. This is what we call local object creation. This is interesting if method "foo" changes the state of p based on the method call receiver.

```
1 + if (r == null) {
2 +   new Foo().foo(p);
3 + } else {
4     r.foo(p);
5 + }
```

This sums up in 4 possible strategies for null replacement (see Table 3.1): use an existing value locally (S1a), use an existing value globally (S1b), use a new value locally (S2a) and use a new value globally (S2b).

3.1.1.2 Strategies Based on Execution Skipping

We also propose strategies to skip the statement where a null pointer exception would happen. There are different possible ways of skipping.

Line Skipping (S3) First, the straight-forward strategy S3 consists in skipping only the problematic statement and allows to avoid the null pointer exception at this location.

```
1 + if (r != null) {
2     r.foo(p);
3 + }
```

The other skipping strategies skip the rest of the method execution. There are four different possibilities that can be considered to achieve this goal. Let us consider that the method returns an object of type "Bar".

Return Null S4a If the method expects a return value, one can return null: this is a reasonable option because it is possible that the caller has a non-null check on the returned object.

```
1 + if (r == null) {
2 +   return null;
3 + }
4   r.foo(p);
```

Return New Object S4b One can return a new instance of the expected type. As for strategy S2b, this is a strategy parameterized by all possible constructor calls.

```
1 + if (r == null) {
2 +   return new Bar();
3 + }
4   r.foo(p);
```

Return Variable S4c Another one can search in the set of accessible values one which corresponds to the expected return type and return it. As for strategy S1b, this is a strategy parametrized by all possible type-compatible variables.

```
1 + if (r == null) {
2 +   return anotherVar;
3 + }
4   r.foo(p);
```

Return S4d When the method does not return anything (`void` return type in Java), inserting a simple "`if (r==null) { return; }`" is a valid option.

All strategies are listed in Table 3.1. The table represents the different dimensions of the analysis: replacement vs skipping, local vs global, reusing objects vs creating new ones.

3.1.1.3 Novelty

Among those 9 strategies, some of them have already been explored. Dobolyi et al. [102] have proposed two of them: S2b and S3. Kent [103] have defined S2b, S3, S4a and S4d. Long et al. [107] have explored S3, S4b and S4d. This means that 4/9 strategies presented in this contribution are new: S1a, S1b, S2a, S4c. The novelty lies on the idea of reusing existing objects (S1a, S1b, S4c) and performing local injection (S1a, S2a).

3.1.2 Template-Based Patch Generation for Null Pointer Exception

We now present our first patch generation technique, TemplateNPE. This technique consists first, of analyzing the stacktrace of the null pointer exception, to find the location of the failure. Second, it applies one by one all the strategies presented in Section 3.1.1. The algorithm of this technique is detailed in Algorithm 3.1. The idea of this algorithm is to explore one strategy after each other, using source code transformation techniques, and then to verify whether the strategies repairs the null pointer exception under consideration. A *"tentative patch"* is created each time that the patched application compiles. Hence, this algorithm explores the search space of all tentative patches. Note that this technique requires recompiling one file for each pair $(parameter, strategy)$.

All patch templates of TemplateNPE are parametrized. The first template parameter contains the Java expression that may be null, it is located in the condition of the template as follows: `if (<parameter> == null) ...` . The second template parameter refers to the expression that is used to replace the null expression. This expression can be a variable (S1a, S2b), a new instance or a predefined constant (null, 0, 1, "", ' ') for S2a, S2b. This template parameter is not present in S3, S4a and S3). The replacement expressions are statically created based on the static analysis of the context of the line that produces the null dereference. We use code analysis to know which variables are accessible by the expression causing the null pointer exception, and also to list the constructor calls for building new instances.

Algorithm 3.1 TemplateNPE: Exploration of all tentative patches based on parametrized templates.

Input: p: a program
Input: t: a test case reproducing a NPE
Input: e: expression that triggers the NPE
Input: S: a set of repair strategies
Output: P: set of tentative patches
Output: Q: set of valid patches

```
1: compile p
2: for s in S do
3:    A ← possible parameter value for s in e
4:    for a : A do
5:       x ← apply s on p parametrized by a
6:       recompile x
7:       if x compiles then
8:          add x to P
9:       end if
10:      run t against x
11:      if t succeeds then
12:         add x to Q
13:      end if
14:   end for
15: end for
```

3.1.3 Metaprogramming-based Patch Generation for Null Pointer Exception

We now present a NPEfix: a metaprogramming approach to generate patches for null pointer exception. NPEfix contains the 9 strategies presented in Section 3.1.1 in a metaprogram.

In this contribution, a metaprogram is a program enriched with behavior modification hooks. By default, all behavior modification hooks are deactivated, which means that by default a metaprogram is semantically equivalent to the original program.

Let us consider the program `x = y + z`. A metaprogram is for instance `x = y - z if HOOK_IS_ACTIVATED else y + z` (functional style) or `x = HOOK_IS_ACTIVATED ? y - z : y + z` (ternary expression, e.g. in Java). Variable HOOK_IS_ACTIVATED is a Boolean global variable which controls the activation of the behavior modification. This metaprogram allows one to transform at run-time an addition into a subtraction. In our context, the metaprogram is automatically created using source-to-source transformation.

The metaprogramming code transformations are realized in a tool for Java called NPEfix. NPEfix is composed of three main phases: first, it generates a metaprogram out of the program under repair, second it compiles the metaprogram, third, it runs the failing test case a large number of times, each time altering the behavior to find a way to avoid the null pointer exception and thus emulate a patch.

3.1.3.1 Core Intercession Hook

To modify the behavior when a null pointer exception happens, we encapsulate each method call and field access as shown in Listing 3.1.

The call of `doSomething` that is originally present is now wrapped with the method `checkForNull`. The Method `checkForNull` does the following things: it first assesses whether the object is null, i.e. whether a null pointer exception will occur; if it's not null, the program proceeds with its normal execution. If the object is null, it means a null pointer exception is about to be thrown, and then a strategy is applied. For sake of simplification, this is shown as a switch as in Listing 3.1, this switch case is the core intercession hook of the metaprogram.

```
//before transformation
Foo o;
o.doSomething();

// after NPEFix transformation
checkForNull(o, Foo.class).doSomething();

// with static method checkForNull
Object checkForNull(Object o, Class c){
    if (o == null) // null pointer exception detected
        switch (STRATEGY) {
            case s1b: return getVar(currentMethod());
            case s2b: return createObject(c);
            ...
        }
    return o;
}
```

Listing 3.1 Detecting harmful null pointer exceptions with code transformation.

```
//before transformation
public void method(){
    ...
    Object a = {defaultExpression};
    a = {newValue};
    ...
}

// after NPEFix transformation
public void method(){
    collectField(myField, "myField");
    ...
    Object a = initVar({defaultExpression}, "a");
    a = modifyVar({newValue}, "a");
    ...
}
```

Listing 3.2 Maintaining a set of variables as pool for replacement at run-time.

Value Replacement Strategies There are four strategies based on value replacement (the first half of Table 3.1): S1a, S1b, S2a and S2b.

Reuse Variable For replacing a null value with a variable, the challenge is to maintain a set of variables as a pool for replacement at run-time. Listing 3.2 shows how we tackle this problem: we use a stack to store all the variables of each method. Each variable initialization and assignment inside the method is registered thanks to the NPEfix' method `initVar`. In addition, at the beginning of each method, we collect all the accessible fields and parameters.

```
// before transformation
value.dereference();

// after NPEfix transformation
if (skipLine(value)){
    value.dereference();
}
Boolean skipLine(Object... objs){ // NPEfix framework
    for (Object o : objs) {
        if (o == null && cannotCatchNPE() && doSkip())
            return false
    }
    return true;
}
```

Listing 3.3 Implementation of line-based skipping.

Create New Object Now, let us consider that the strategies that create a new variable (strategies S2a and S2b).

As shown in Listing 3.1, a call is made to `createObject` that takes as parameter the static type of the dereferenced variable. `createObject` uses reflection to access all the constructors of a given type. In addition, this method is recursive so it can create objects that requires objects as parameter of the constructor. It tries to create a new instance of the class from each available constructor. Given a constructor, it attempts to create a new instance for each of the parameters recursively. The stopping condition is when a constructor does not need parameters. Note that the primitive types, which don't have constructors, are also handled with default literal values.

Skipping Strategies Now we present how we implement the strategies based on skipping the execution (the second half of Table 3.1).

Line skipping The strategy S3 requires to know if a null pointer exception will happen in a line, before the execution of the line. For this, the transformation presented in Listing 3.1 is not sufficient, because the call to method `checkForNull` implies that the execution of the line has already started. To overcome this issue, we employ an additional transformation presented in Listing 3.3.

Similarly to `checkForNull`, method `skipLine` assesses, before the line execution, whether the dereferenced value is null or not, and whether it is harmful. Method skipline takes an arbitrary number of objects, the ones that are dereferenced in the statement. This list is extracted statically.

```
// before transformation
Object method(){
    ...
    value.dereference();
    ...
    return X;
}
// after NPEfix transformation
Object method(){
    try {
        ...
        if (skipLine(value)){
            value.dereference();
        }
        ...
        return X;
    } catch (ForceReturnError f){
        if (s4a) return null;
        if (s4b) return getVar(Object.class);
        if (s4c) return createObject(Object.class);
    }
}
Boolean skipLine(Object... objs){
    if(hasNull(objs) && cannotCatchNPE() && skipMethodActivated())
        throw new ForceReturnError();
    ...
}
```

Listing 3.4 Metaprogramming for method-based skipping strategies.

Method skipping The remaining strategies are based on skipping the execution of the rest of the method when a harmful dereference is about to happen: these are strategies S4d, S4a, S4c and S4b (the last part of Table 3.1). We implement those strategies with a code transformation as follows.

A try-catch block is added in all methods, wrapping the complete method body. These try-catch blocks handle a particular type of exception defined in our framework (ForceReturnError). This exception is thrown by the `skipLine` method when one of the method-skipping strategies is activated, as show in Listing 3.4. This listing also shows a minimalist example of the code resulting from this transformation.

3.1.3.2 Strategy Selection at Runtime

Now that we have a metaprogram that embeds all strategies, we have a way to explore the search space of null pointer repair purely at run-time. The idea is to first create the metaprogram, then to activate the strategies dynamically by setting the appropriate behavior modification hooks.

From Hooks to Patches Given a combination of behavioral modification hooks, one can create the corresponding source code patch by reinterpreting the hooks according to the templates presented in Section 3.1.1.

3.1.3.3 Implementation

All those transformations have been implemented in a tool called NPEfix, which has been made publicly available for sake of reproducible research and open science[2] and we use Spoon library [108] for the code transformations.

3.1.4 Evaluation

We now evaluate TemplateNPE and NPEfix. We design a protocol to answer the following questions.

- RQ1. What is the impact of NPEfix' runtime analysis of the repair context and TemplateNPE's static analysis of the repair context on the number of explored tentative patches?
- RQ2. Do TemplateNPE and NPEfix produce valid patches?
- RQ3. What are the reasons explaining the presence of different valid patches?
- RQ4. Is the performance of TemplateNPE and NPEfix acceptable?

3.1.4.1 Protocol

In order to evaluate our approach of patch generations, we build a benchmark of real and reproducible null pointer exceptions in Java programs (see Section 3.1.4.3). Then, we compare the ability of TemplateNPE and NPEfix to find different patches and repair each bug of the benchmark. TemplateNPE is the template-based implementation of our nine repair strategies that is described in Section 3.1.2. NPEfix uses our metaprogramming approach to handle null pointer exceptions as described in Section 3.1.3. We consider the patch as "valid" when the null pointer exception is correctly handled and that no other exception (a null pointer exception or another one) is triggered.

[2] https://github.com/Spirals-Team/npefix

Table 3.2 Descriptive statistics of the subjects suffering from null pointer exceptions. Size measured by cloc version 1.60.

Subject	Domain	Size
COLL	Collection library	21 594 LOC
LANG	Utility functions	18 970 LOC
MATH	Math library	90 771 LOC
PDF	PDFBox library	64 375 LOC
Felix	Felix library	33 057 LOC
SLING	Sling library	583 LOC
Total	16 bugs	229 350 LOC

3.1.4.2 Evaluation Metrics

After the repair of each bug with each repair technique, we collect three metrics: the number of tentative patches (whether valid or not); the number of valid patches; and the execution time required to explore the search space of patches.

We interpret those metrics as follows. A larger number of tentative patches means that the search space is richer. As shown in previous work [9, 64], there are often multiple different patches that are able to fix a bug. A larger number of generated valid patches is better, it means that the developer is given more choices to get a really good choice.

The results of this experimentation, including all tentative patches are publicly available at https://github.com/Spirals-Team/npefix-experiments

3.1.4.3 Benchmark

To build a benchmark of real null pointer exceptions in Java, we consider two inclusion criteria. First, the bug must be a real bug reported on a publicly available forum (e.g. a bug tracker). Second, the bug must be reproducible. This point is very challenging since it is really difficult to reproduce field failures, due to the absence of the exact crashing input, or the exact configuration information (versions of dependencies, execution environment, etc.). As a rule of thumb, it takes one day to find and reproduce a single null pointer exception bug. We consider bugs in the Apache Commons set of libraries (e.g. Apache Commons Lang) because they are well-known, vastly used and their bug trackers are public, easy to access and to be searched. Also, thanks to the strong software engineering discipline of the Apache foundation, a failing test case is often provided in the bug report. We have not rejected a single reproducible field null pointer exception.

Table 3.3 Comparison of the the template approach (TemplateNPE) and the metaprogramming approach (NPEfix) on three key metrics.

	# Tentative Patches		# Valid Patches		Execution Time	
Bug ID	Template	NPEfix	Template	NPEfix	Template	NPEfix
collections360	7	**10**	0	0	0:00:20	0:02:32
felix-4960	9	7	4	4	0:00:43	0:03:22
lang304	44	**77**	43	**65**	0:00:11	0:00:26
lang587	21	**28**	12	**28**	0:00:23	0:00:23
lang703	16	15	0	**7**	0:00:09	0:00:20
math1115	8	**11**	6	5	0:00:47	0:02:08
math1117	8	**11**	0	0	0:00:41	0:01:52
math290	9	**10**	3	**4**	0:00:18	0:00:42
math305	2	**4**	1	**3**	0:00:08	0:00:40
math369	15	**16**	14	14	0:00:26	0:00:43
math988a	17	17	11	11	0:01:02	0:01:38
math988b	17	**25**	17	17	0:01:39	0:01:48
pdfbox-2812	6	**14**	2	2	0:00:28	0:01:46
pdfbox-2965	4	3	4	3	0:00:16	0:01:34
pdfbox-2995	4	**5**	3	1	0:00:13	0:01:35
sling-4982	18	**20**	7	**11**	0:00:05	0:00:06
Total	205	273	127	175	0:07:57	0:21:44
Average	12.81	17.06	7.94	10.94	0:00:29	0:01:21
Median	9.00	12.50	4.00	4.50	0:00:22	0:01:35

As a result, we have a benchmark that contains 16 null pointer exceptions (1 from collections, 3 from lang and 7 from math, 3 from PDFBox, 1 from Felix, 1 from Sling). It is publicly available for future research (https://github.com/Spirals-Team/npe-dataset). The main strength of this benchmark is that it only contains real null pointer exception bugs and no artificial or toy bugs. Table 3.2 shows the size of applications for which we have real field failures.

3.1.4.4 RQ1. What is the impact of NPEfix' runtime analysis of the repair context and TemplateNPE's static analysis of the repair context on the number of explored tentative patches?

Table 3.3 presents the results of our experiment. The first column contains the bug identifier. The second column contains the number of tentative patches for each bug (i.e. the size of the search space). This column is composed of two sub-columns: the number of tentative patches using the template-based approach (TemplateNPE) and

the number of patches generated by the metaprogramming approach (NPEfix). For example, TemplateNPE identifies 7 tentative patches and NPEfix identifies 10 tentative patches. The remaining top-level columns will discussed below in Section 3.1.4.5 and Section 3.1.4.7.

In 12/16 (in bold) of the case NPEfix explores more tentative patches than TemplateNPE. This validates that the static analysis and dynamic analysis of the repair context differs and that the latter is potentially richer. The difference in the number of tentative patches between TemplateNPE and NPEfix is explained as follows.

- During execution, more objects are detected as compatible with the type of the null expression. With the template approach, we do not know the actual runtime type of all variables.
- The number of different new objects created varies because NPEfix detects at runtime more compatible constructors.
- Some strategies cannot be applied at certain locations with the template approach. For example, TemplateNPE cannot apply the skip-line strategy (S3) on a local variable. This case is naturally handled in the metaprogramming approach.
- In 3/16 cases, the template-based approaches identifies more tentative patches. The reason is that NPEfix filters out equivalent patches by on the runtime variable value. This is further discussed in Section 3.1.4.6 and Section 3.1.4.6.

RQ1. What is the impact of NPEfix' runtime analysis of the repair context and TemplateNPE's static analysis of the repair context on the number of explored tentative patches? NPEfix explores 273 tentative patches, and 205 for the template-based approach. In other words, the search space of the metaprogramming technique is larger. This validates our intuition that the runtime analysis of the repair context is valuable in certain cases.

3.1.4.5 RQ2. Do TemplateNPE and NPEfix produce valid patches?

Table 3.3 also gives the number of tentative patches that are valid, i.e. that make the failing test case passing. This is shown in the 4th and 5th columns under the top-level header "# Valid Patches" presents the results of this experiment. For example, for bug Collection-360, neither TemplateNPE nor NPEfix identifies a valid patch. For lang304, TemplateNPE identifies 43 valid patches, while NPEfix finds 65 valid patches. As we can

see there is a correlation between the size of the explored search space and number of valid patches identified. This means that it is worth exploring more tentative patches to identify more valid patches.

RQ2. Do TemplateNPE and NPEfix produce valid patches? NPEfix finds 175 patches that avoid the null pointer exception and make the test case passing. Among those valid patches, 48 patches of them are uniquely found by the metaprogramming approach thanks to the runtime analysis of the repair context.

3.1.4.6 RQ3. What are the reasons explaining the presence of different valid patches?

We answer to this research question with 3 case studies.

Math-305 We now discuss the generated patches for bug Math-306, where the null pointer exception is thrown during the computation of a clustering algorithm called Kmeans. The null pointer exception is triggered when a point is added to the nearest cluster. When the library first computes the nearest cluster, it fails because the current point is at a distance largest than Integer.MAXVALUE. Then, the nearest cluster is set to null and a null pointer exception is thrown when the library tries to add the current point to it.

NPEfix succeeds to identify 4 different patches for this bug. They are all presented in Figure 3.1. The top three first patches (in green) of the figure are valid: they all avoid the null pointer exception to be thrown. As we see, they have the same behavior: they skip the line that produces the null pointer exception. The first and the third patches create a new cluster that is never used in the application: this is useless but it works. The second patch skips the line that produces the null pointer exception, resulting in the point not being added to the cluster. The last patch is invalid because it produces a division per zero later in the execution: the test case still fails (not with the original null pointer exception but with a division-by-zero exception).

Felix-4960 Felix is an implementation of the OSGI component model. Real bug Felix-4960 is about a null pointer exception that is thrown in method "getResourcesLocal(name)" which is dedicated to searching a resource in a path. The null pointer exception appears

```
--- org/apache/commons/math/stat/clustering/KMeansPlusPlusClusterer.java
+++ org/apache/commons/math/stat/clustering/KMeansPlusPlusClusterer.java
@@ -90,3 +90,7 @@
90              Cluster<T> cluster = getNearestCluster(clusters, p);
91 -            cluster.addPoint(p);
92 +            if (cluster == null) {
93 +                    new Cluster(null).addPoint(p);
94 +            } else {
95 +                    cluster.addPoint(p);
96 +            }
97          }
```

```
--- org/apache/commons/math/stat/clustering/KMeansPlusPlusClusterer.java
+++ org/apache/commons/math/stat/clustering/KMeansPlusPlusClusterer.java
@@ -90,2 +90,5 @@
90              Cluster<T> cluster = getNearestCluster(clusters, p);
91 +            if (cluster == null) {
92 +                    cluster = new Cluster(null);
93 +            }
94              cluster.addPoint(p);
```

```
--- org/apache/commons/math/stat/clustering/KMeansPlusPlusClusterer.java
+++ org/apache/commons/math/stat/clustering/KMeansPlusPlusClusterer.java
@@ -90,3 +90,5 @@
90              Cluster<T> cluster = getNearestCluster(clusters, p);
91 -            cluster.addPoint(p);
92 +            if (cluster != null) {
93 +                    cluster.addPoint(p);
94 +            }
95          }
```

```
--- org/apache/commons/math/stat/clustering/KMeansPlusPlusClusterer.java
+++ org/apache/commons/math/stat/clustering/KMeansPlusPlusClusterer.java
@@ -90,2 +90,5 @@
90              Cluster<T> cluster = getNearestCluster(clusters, p);
91 +            if (cluster == null) {
92 +                    return null;
93 +            }
94              cluster.addPoint(p);
```

Figure 3.1 The generated patches for the bug Math-305, the patches that have a green border are valid patches and the patch that has a red border is the invalid patch.

when Felix does not succeed to get a list of resources in the path. In this case, Felix tries to iterate on a list that is null, which triggers the null pointer exception.

For Felix-4960, TemplateNPE is surprisingly able to generate more tentative patches: NPEfix generates 7 tentative patches and TemplateNPE generates 9 tentative patches that compile. The reason is that template generates tentative patches based on reusing variables, however, those patches are meaningless. TemplateNPE fills the template parameters with `m_activationIncludes` and `m_declaredNativeLibs`. However, those variables are null because not initialized at this location in the code. In other words, TemplateNPE generates a patch that replaces a null by a null, which obviously results in the same null pointer exception as before.

On the contrary, NPEfix works at runtime, and hence knows that the actual value of `m_activationIncludes` and `m_declaredNativeLibs` are null. Hence, it does not even try them, because it knows in advance that such a tentative patch would be invalid. In this case, the runtime analysis of the context is interesting to discard incorrect patches early in the process.

PDFBbox-2965 PDFBbox is a PDF rendering library. This library allows one to read the properties of a PDF file. PDFBbox-2965 happens when PDFBox searches for a specific field on a PDF that contains no PDF form. Internally, PDFBox iterates on all form fields of the PDF and compares the name of the field against the searched field. But when the PDF contains no form, the list of fields is null, and a null pointer exception is thus triggered.

NPEfix generates 3 tentative patches and TemplateNPE generates 4 tentative patches for this bug. They are all valid according to the test case. As we see, TemplateNPE generates one additional valid patches. This patch is shown in Listing 3.5. It consists of returning variable `retval` if the `fields` variable is null. This patch is of low-quality because `retval` is always null in this case. In other words, a simple `return null;` is a more explicit patch, and would better help the developer. This explains why NPEfix does not generate this patch, as for Felix-4960, thanks to runtime analysis, NPEfix knows that `retval` is null, and that `return retval;` is equivalent to `return null;` (strategy S4a).

```
    --- pdfbox/pdmodel/interactive/form/PDAcroForm.java
    +++ pdfbox/pdmodel/interactive/form/PDAcroForm.java
    @@ -250,2 +250,5 @@
250
251 +   if (fields == null) {
252 +       return retval; // reval is null
253 +   }
254     for (int i = 0; i < fields.size() && retval == null; i++)
```

Listing 3.5 The additional patch of TemplateNPE for the bug PDFBbox-2965.

3.1.4.7 RQ4. Is the performance of TemplateNPE and NPEfix acceptable?

The last column of Table 3.3 presents the execution time required to explore the whole search space of tentative patches for fixing the null pointer exception. For example, for the bug Collections-360, TemplateNPE explores the search space in 20 seconds while NPEfix requires 2 minutes 32 seconds.

In most cases, the template-based approach is faster. The reason is that creating the metaprogram takes a lot of time due to the complexity of code transformations. Also, the additional code injected in the metaprogram slows down each repair attempt, i.e. slows down each execution of the failing test case. For NPEfix, the complete exploration of the search space takes at most 3 minutes 22 seconds. We consider this acceptable since a developer can wait for 3 minutes before offered a set of automatically tentative patches.

RQ4. Is the performance of TemplateNPE and NPEfix acceptable? The complete exploration of the search space of tentative patches is faster with TemplateNPE. This highlights a trade-off between the number of patches explored found and the time to wait.

3.1.5 Discussion

3.1.5.1 Patch Readability and Templates

One concern of automatic patch generation is the readability of the generated patches and its impact on the maintainability of applications [56]. Let us discuss the readability of NPEfix and TemplateNPE patches. NPEfix patches repair one specific type of bug: null dereference. We observe that most developers fix these bugs by adding a null check (`if(... != null) something()`) before the null expression. The strategies we used in this approaches resemble human-written patches and thus and can be as easily

understood and maintained as human patches. Our experience with NPEfix suggests that template-inspired patches enable one to encode – if not enforce – readable patches.

3.1.5.2 Genetic Improvement and Metaprogramming

Genetic improvement [109] refers to techniques that change the behavior of a program in order to improve a specific metric, for example, the execution time. There is an interesting relation between genetic improvement and NPEfix. Both are based on dynamically changing the behavior of an application. In this perspective, making failing tests pass can be considered as a functional metric to be optimized. Indeed, we think that metaprogramming techniques similar to that proposed in this contribution could be used for genetic improvement by creating a space of different program behaviors to explore.

3.1.5.3 Threat to Validity

A bug in the implementation of TemplateNPE or NPEfix is the threat to the validity of the findings based on quantitative comparison presented in Table 3.3.

Our benchmark is only composed of real bugs due to null pointer exceptions. However, since they are all in Java and from 6 projects, there is a threat to the external validity of our findings. Other bugs and an implementation of TemplateNPE and NPEfix in another language may uncover other behavioral differences.

3.1.6 Conclusion

In this section, we have presented two techniques to generate patches for null pointer exception bugs. The two techniques are based on 9 strategies that are specific to null pointer exceptions. The first presented technique uses a template-based approach and the second one uses metaprogramming approach to generate patches. We have evaluated our technique on 16 real null pointer exceptions: TemplateNPE generated 127 valid patched and NPEfix 175 valid patches over all considered bugs. The two techniques have shown to be able to generate patches readable and understandable patches in a short amount of time.

3.2 Enriched Expression Synthesizer

This section presents the last automatic patch generation technique of this chapter: DynaMoth. DynaMoth is code synthesizer engine that collects the runtime state of an application: variables, method calls and creates with them patches for buggy Java conditions.

This patch generation technique has two goals, the first goal is to create the first automatic patch generation technique that directly connects to the state of the application in order to synthesize patches. This removes the requirement to a failing test case since DynaMoth can connect any type of execution, which can be a test case or be directly the running application. The second goal is to create richer patches by exploiting the running state of the application, by creating patches that contain method calls.

DynaMoth is working as follows. At runtime, the synthesis engine stops the execution of the program under repair and collects the accessible variables and calls methods in order to constitute a set of ingredients for the patch generation. Then, it combines the ingredients with arithmetic operators to form new Java expressions. Finally, the behavior of the generated expression is compared to the expected behavior, the test suite of the application and fix the execution that produces the incorrect behavior (in this case a test case).

This section is organized as follows. Section 3.2.1 describes the main contribution of this work: a dynamic code synthesis engine for repair. Section 3.2.2 contains the evaluation of DynaMoth based on 224 real-world bugs from Defects4J. We present our conclusions in Section 3.2.3.

3.2.1 An Algorithm for Condition Synthesis

We now describe the operation of DynaMoth in detail.

3.2.1.1 Fault Classes

DynaMoth generates two types of patch, the first type of patch is to change the condition of an existing `if-then-else` (see Section 3.2.1.1), the second type of type of is to wrap a statement with a pre-condition (see Section 3.2.1.1).

```
1 -  if (u * v == 0) {
2 +  if (u == 0 || v == 0) {
3     return (Math.abs(u) + Math.abs(v));
4   }
```

Listing 3.6 Patch example for a buggy IF condition. The original condition with a comparison operator == is replaced by a disjunction between two comparisons. An overflow could thus be avoided.

```
1 + if (specific != null) {
2     sb.append(": "); //sb is a string builder in Java
3 + }
```

Listing 3.7 Patch example: a missing precondition is added to avoid a null reference. The patch is specific to the object-orientation of Java.

Buggy IF Conditions The first kind of bugs that Nopol targets are buggy conditions in `if-then-else` statements. Listing 3.6) gives an example of such a repair.

Missing Preconditions The second class of bugs addressed by Nopol are preconditions. A precondition is a check consisting in the evaluation of a Boolean predicate guarding the execution of a statement or a block: Listing 3.7 gives an example of bug fixed by adding a precondition. Preconditions are commonly used to avoid 'null-pointer' or 'out-of-bound' exceptions when accessing array elements.

3.2.1.2 Architecture

Figure 3.2 describes the general algorithm of DynaMoth that is designed on top of Nopol [3]. This algorithm is composed of four main steps. The first step is the localization of suspicious statements. It uses standard spectrum-based fault localization using Ochia [110]. The second step is angelic value mining, that consists in trying to identify an arbitrary value required somewhere during the execution to pass the failing test(s). As third step, input-output based code synthesis is used to generate a new Java expression which is the patch. Since DynaMoth repairs conditions, the synthesized expressions are Boolean expressions. Finally, for patch validation, DynaMoth re-executes the whole test suite on the patched program.

Angelic Value Mining The angelic value mining step determines the required value of the buggy Boolean expression to fix the buggy execution. For buggy if-conditions, angelic value mining is done by forcing at runtime the branch of the suspicious IF statement.

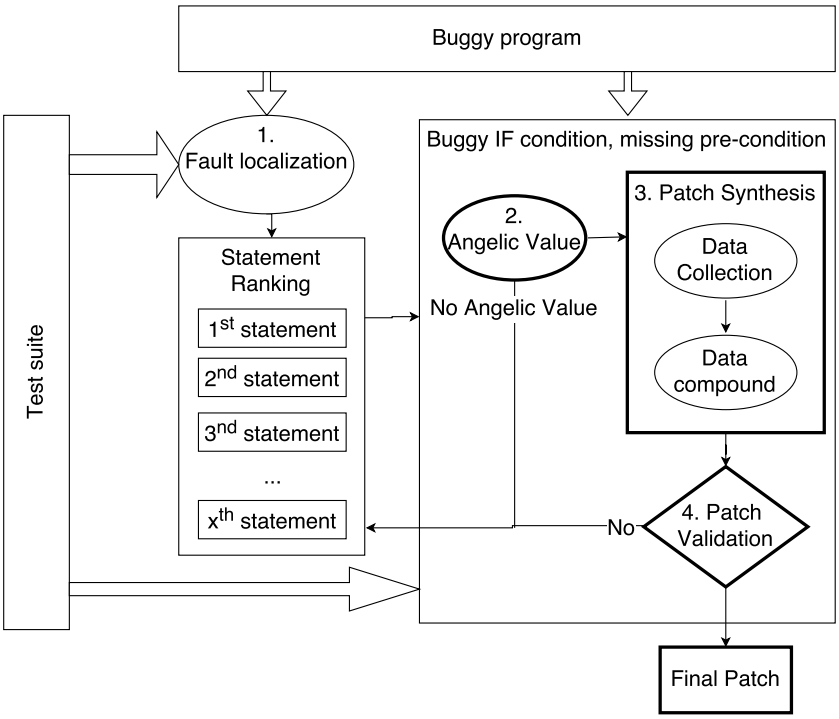

Figure 3.2 Overview of the DynaMoth repair system.

The condition expression is arbitrarily substituted for either `true` or `false`: the state of the program is artificially modified during its execution as it would have been by an omniscient oracle or *angel*. If the buggy execution succeeds with the modified execution, the used value is called the angelic value and is stored for use in synthesis later.

For missing preconditions, suspicious statements are forcefully skipped at runtime. When this makes the failing execution succeed, a precondition has to be synthesized, and the condition has to be equals to `false` in the context of the buggy execution. For example, in Listing 3.7, let us assume that two executions specify the code. The first execution, which is a test, specificity the nominal case when `specific != null`. The second execution, which is buggy, executes the buggy statement with `specific == null`. In this case, the angelic value of the precondition will be `false` for the second test case. That is the Boolean expression to be synthesized as precondition should return true when `specific != null` and false otherwise.

Once an angelic value is known for all test cases, we obtain an input-output synthesis problem. The input is the context of the statement under repair (all variables in the scope), the output is the angelic value.

Algorithm 3.2 The main algorithm of DynaMoth.

Input: A: a set of angelic values for specific statements
Output: P: a set of patches

```
1: P ← ∅
2: for all angelic value at statement s in A do
3:    collect simple EExp at runtime at s (Algorithm 3.3)
4:    combine simple EExp as compound EExp (Algorithm 3.4)
5:    compare all EExp against the expected angelic value (Algorithm 3.5)
6:    if valid EExp found then
7:       add patch to P
8:    end if
9: end for
10: return P
```

Code Synthesis Once the angelic value is mined, we know the expected behavior of the buggy condition. This expected behavior is the behavior that the synthesized expression must have to fix the application.

As shown in Algorithm 3.2, the code synthesis of DynaMoth is composed of three steps: first, the collection of the runtime contexts c of the suspicious condition (see Section 3.2.1.2) for which it exists an angelic value. The runtime context includes parameters, variables, fields and return values of method calls; second, the generation of new Boolean expressions (see Section 3.2.1.2); third, the comparison of each new EExp to the expected values (see Section 3.2.1.2).

We now introduce two definitions, for runtime context and EExp;

Runtime context The runtime context of a statement is made of the values of all local variables, static variables, parameters, fields and method call available in the scope of the statement. A runtime context also contains a reference to a test and an integer value representing the n-th times that the statement has been executed by the test.

EExpA Evaluated-Expression, denoted EExp (c), is a pair e, v where e is a valid Java expression and v the value of the Java expression in a specific runtime context c.

An example of EExp is $"e.size()", 4$ which means that for a given context, the result of the evaluation of "e.size()" is 4.

Runtime Context Collection Algorithm 3.3 presents our technique to collect the runtime context. It is achieved by stopping the execution of the program under repair at a specific location. In our case, we stop the execution only at the locations for which

Algorithm 3.3 Collecting the set of runtime contexts for a statement S during test execution. Legend: ← means "add to set".

Input: Statement S, T: a set of tests
Output: R: a set of runtime contexts

```
1: add breakpoint at S
2: for all t in T do
3:    run t
4:    if is stopped at breakpoint then
5:       eExpList = ∅
6:       eExpList ← variables ∪ fields
7:       eExpList ← (all method calls on this)
8:       for 1 to max_depth do
9:          for all eExp in eExpList do
10:            eExpList ← (fields eExp)
11:            eExpList ← (all method calls on eExp)
12:         end for
13:      end for
14:      i ← iteration number for this test
15:      R_{t,i} ← eExpList
16:      proceed test execution
17:   end if
18: end for
19: return R
```

we have identified an angelic value (the angelic value mining technique is described in Section 3.2.1.2.) The execution of the program is stopped using debugging technology (see Section 3.2.1.4). Once the execution is stopped, DynaMoth collects the runtime context of the statement by inspecting the variables in the scope. Then, DynaMoth collects the field values and calls the methods on each Java object in the runtime context. This step is executed recursively on the fields or the returned values that have just been collected. For example this recursion allows the synthesis of chained expressions such as `variable.method(p1, p2).getter()`. We limit the number of recursion to a constant (max_depth in Algorithm 3.3) in order to limit the size of the search space. For the sake of performance, we do not consider the possible side effects of method calls, which results in potentially corrupted collected values. This hinders completeness (some patches may not be found because of this) but not soundness because DynaMoth validates the generated patch candidate at the end of the execution as described in Section 3.2.1.2.

DynaMoth also collects all literals present in the method where the breakpoint is added and four literals frequently present in patches: `-1, 1, 0, null`. DynaMoth accesses all

static variables and calls all static methods that are used in the scope of the suspicious statement.

For each execution of a suspicious statement, DynaMoth collects a set of simple EExp representing an expression and the result of its evaluation in the current runtime context. That basic EExp will be used to form the actual patch.

Compound EExp Synthesis After the collection of simple EExp (described in Section 3.2.1.2), DynaMoth combines them with operators.

The first kind of compound EExp is an expression that contains null checks. For example, let us consider two runtime contexts: the first contains a variable `v` which is an `array` and the second runtime context contains the same variable `v` but this time equal to `null`. With the first context, DynaMoth can access the field: `length` of the array but in the second context, `v` does not have fields. In this case, DynaMoth can generates compound EExp of the form: `v != null && v.length == 0` evaluating to true in the first context and false in the second.

The second phase consists of generating compound expressions with binary logic and arithmetic operators. This is done by generating new expressions combining all EExp collected so far with compatible logic/arithmetic operators: `+`, `-`, `*`, `/`, `||`, `&&`, `==`, `!=`, `<`, `<=`.[3] A combination creates a new EExp and its value is compared to the expected value (this comparison is described in Section 3.2.1.2). This phase is executed recursively to create more and more complex expressions until the configuration parameter depth is reached. For example, DynaMoth can produce this type of expression: `(matrix != null && 0 == matrix.multiply(this).getReal()) && (array == null || list.getSize() < array.length)`.

EExp Validation Each time a Boolean EExp is created, DynaMoth verifies that the EExp returns the expected value predicted by angelic value. If this is the case, it means that the EExp fixes the bug locally, for this particular runtime context. When this happens, DynaMoth validates the patch for all other runtime contexts. For example, we have the EExp`array == null || array.length == 0` and two runtime contexts: `array = null` and `array = [42],array.length = 1`. The evaluation of the EExp with the first runtime context is equal to `true` because `array = null` and `false` with the second

[3]The operators `> and >=` are obtained by symmetry of `<= and <`.

Algorithm 3.4 Combining EExp's with a operators.

Input: eExpList: a list of EExp, O: a set of operators
Output: eExpList: EExp enriched with coumpound EExp

```
1: for 1 to max_depth do
2:   for all operator o in O do
3:     n ← number of required operators for o
4:     for all t: tuple of size n in eExpList do
5:       if types of t values compatible with o then
6:         eExpList ← combine(t, operator)
7:       end if
8:     end for
9:   end for
10: end for
```

Algorithm 3.5 Assessing whether a valid patch exists at a given statement S.

Input: eExp: a set of valued EExp for all tests at statement s, A a set of angelic values for all tests, T a set of tests
Output: true if eExp is a valid patch

```
1: for all t in T do
2:   for all iteration i of S in t do
3:     eExpValue ← eExp_{t,i}
4:     angelic ← A_{t,i}
5:     if eExpValue ≠ angelic then
6:       return false
7:     end if
8:   end for
9: end for
10: return true
```

runtime context because `array = null` and `array.length = 1`. If those are the two expected angelic values for a precondition, it means that a patch has been found.

3.2.1.3 Optimization Techniques

The search space of the synthesis is composed of all expressions that can be generated from basic EExp initially collected. In many cases, the search space is too large to be covered entirely. Consequently, we have designed and implemented several optimizations that either reduce the size of search space or accelerate the discovery of a patch.

Exploration of Compound EExp During the synthesis of compound EExp for a given runtime context, we only consider EExp that have different values. For example, let us consider that DynaMoth is creating a binary expression and is looking for a Boolean expression for the right operand. Furthermore, in the scope under consideration, two compatible variables are both equal to `true`. In this case, DynaMoth will only consider the first variable because the value of the new EExp will be exactly if it had considered the second variable. This greatly prunes the search space for a single runtime context. This optimization of the search space is similar to that of J. Galenson et al. in CodeHint [111]. This does not decrease the synthesis power: if two EExp may evaluate to the same value in one runtime context c_1 but may have a different value in another one c_2, the synthesis of a patch using EExp $_1$ would be discarded in c_1 but explored in c_2. If it works on c_2 it would be validated on c_1 afterwards.

Number of Collected Runtime Contexts We need a threshold on the number of collected runtime contexts. We have found that a good threshold is that DynaMoth stop collecting after 100 runtime contexts of a given suspicious statement for a given test case. For instance, if the suspicious statement is in a loop, the runtime collection will start at each loop iteration. In order to limit the execution time of the collection, DynaMoth considers only the 100 first iterations and ignores the others. This optimization is sound because the synthesized patch is still validated on the real bug with all executions of the suspicious statement.

Method Calls Used in Synthesized Patch The number of possible method calls yields of huge search space. Patches are unlikely to call methods that have never been used in the program. Consequently, DynaMoth only collects method calls which are used elsewhere in the program.

Ignore equivalent expressions DynaMoth ignores equivalent expressions such as binary expressions with a commutative operator (example: `x + y` and `y + x`). DynaMoth ignores binary expressions which contain a neutral operand (example: `0 + x` or multiplication by `0`). DynaMoth also ignores all trivially incorrect patches such as the comparison of two literals (example: `1 == 2`).

Method Invocation Time Timeout During runtime context collection, we set a threshold on the execution time of method calls (2 seconds per default). This timeout

creates an upper bound on the execution time of runtime context collection and mitigates the problem of infinite loops caused by invalid method parameters.

Ordering of Runtime Contexts All runtime contexts do not contain the same number of simple EExp, before the combination step, as shown in Section 3.2.1.2. Consequently, the runtime context that contains more EExp has more chance to produce a patch because more combinations of EExp are possible. By considering first the runtime contexts that have more EExp, we reduce the time spent searching the patch in a runtime context that cannot produce it.

Estimation of Usage Likelihood of Operators and Operands DynaMoth collects the usage statistics of variables, operators, and methods in the class that contains the suspicious statement. For instance, if we are repairing a statement in class Foo, DynaMoth extracts the information that operator "+" is used 10x in Foo, method "substring" is used 4x in Foo, etc. This usage statistics are used to drive the synthesis of compound EExp: the composition operators that are most frequently used are prioritized during synthesis. This optimization is based on our intuition that patches are more likely to contain operators and methods that have previously been used in the class under consideration.

3.2.1.4 Implementation Details

DynaMoth uses the Java Debugging Interface as to set breakpoints and collect the runtime contexts. The Java Debugging Interface (JDI) is an API of the virtual machine that provides information for debuggers and systems needing access to the running state of a virtual machine.[4]

For collecting the usages statistics (see Section 3.2.1.3 and Section 3.2.1.3) DynaMoth uses a library for analyzing Java source code: Spoon [108]. Spoon is also used during the angelic value mining (Section 3.2.1.2) and during the final patch validation. The localization of suspicious statement is delegated to the library: GZoltar [112].

The source code of DynaMoth is part of the Nopol project [3] and is publicly available on GitHub at https://github.com/SpoonLabs/nopol

[4]JDI documentation https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/.

3.2.2 Evaluation

In order to evaluate the effectiveness of DynaMoth, we execute it on the Defects4J [63] dataset of 224 real-world bugs. The methodology of this evaluation is composed of the following dimensions: the bug-fixing ability, the patch readability, and the execution performance. First, our evaluation protocol is described in Section 3.2.2.1. Second, our three research questions (**RQ's**) are detailed in Section 3.2.2.2, and finally the responses to our research questions given in Section 3.2.2.3.

3.2.2.1 Protocol

We run DynaMoth to repair the Defects4J bugs. Defects4J by Just et al. [63] is a database of 357 reproducible real software bugs from 5 open-source Java projects. Defects4J provides a framework which abstracts over compilation and test execution.

Defects4J is the largest open and structured database of real-world Java bugs. We use four out of five projects currently available in the Defects4J dataset, i.e., Commons Lang,[5] JFreeChart,[6] Commons Math,[7] and Joda-Time.[8] We do not use the Closure Compiler[9] project because the test suite used in this project is incompatible with our implementation.

The evaluation of DynaMoth on the 224 of Defects4J takes days to be completed and require a large amount of computer power. To overcome this problem, we use Grid5000, a grid for high-performance computing [113]. We define a timeout of 1 hour and 30 minutes for each repair execution.

3.2.2.2 Research Questions

1. **RQ1**. Bug fixing ability: Which bugs of Defects4J can automatically be repaired with DynaMoth? In canonical test suite based repair, a bug is considered as fixed when the whole test suite passes with the patched program. To answer this quantitative question, we run DynaMoth on each bug of the Defects4J dataset and count the number of fixed bugs.

[5] Apache Commons Lang, http://commons.apache.org/lang.
[6] JFreeChart, http://jfree.org/jfreechart/
[7] Apache Commons Math, http://commons.apache.org/math.
[8] Joda-Time, http://joda.org/joda-time/
[9] Google Closure Compiler, http://code.google.com/closure/compiler/

Table 3.4 Results on the repairability of DynaMoth and SMTSynth on Defects4J. DynaMoth repairs 27 bugs (12%).

Project	Bug Id	SMTSynth	**DynaMoth**
JFreeChart	C3	Fixed	–
	C5	Fixed	Fixed
	C6	–	Fixed
	C9	–	Fixed
	C13	Fixed	–
	C17	–	Fixed
	C21	–	Fixed
	C25	Fixed	Fixed
	C26	Fixed	Fixed
Commons Lang	**L39**	Fixed	Fixed
	L44	Fixed	–
	L46	Fixed	–
	L51	Fixed	–
	L53	Fixed	–
	L55	Fixed	–
	L58	Fixed	–
Commons Time	**T7**	–	Fixed
	T11	Fixed	Fixed
Total	43 (19.2%)	35 (15.6%)	27 (12%)

Project	Bug Id	SMTSynth	**DynaMoth**
Commons Math	**M32**	Fixed	Fixed
	M33	Fixed	Fixed
	M40	Fixed	–
	M41	–	Fixed
	M42	Fixed	Fixed
	M46	–	Fixed
	M49	Fixed	Fixed
	M50	Fixed	Fixed
	M57	Fixed	–
	M58	Fixed	Fixed
	M69	Fixed	–
	M71	Fixed	Fixed
	M73	Fixed	–
	M78	Fixed	Fixed
	M80	Fixed	–
	M81	Fixed	–
	M82	Fixed	–
	M85	Fixed	Fixed
	M87	Fixed	Fixed
	M88	Fixed	Fixed
	M96	–	Fixed
	M97	Fixed	Fixed
	M99	Fixed	Fixed
	M104	Fixed	–
	M105	Fixed	Fixed
Total	43 (19.2%)	35 (15.6%)	27 (12%)

2. **RQ2**. Patches Readability: Are the synthesized patches simpler than those generated by SMTSynth? The readability of the generated patches is an important factor when developers comprehend and validate them before including them in the code base. To answer this question, we look at the number of expressions (variable, literals, parameters), method calls and operators in each patch.

3. **RQ3**. Performance: How long is the execution of DynaMoth? For automatic repair to be applied in practice, it is important that it does not run too long. For instance, requiring one week to find a patch is not acceptable for developers. To answer this question, we analyze the execution time of DynaMoth when it finds a patch.

3.2.2.3 Empirical Results

We perform the experiment described in Section 3.2.2.1. The total execution time of this experiment is more than 11 days.

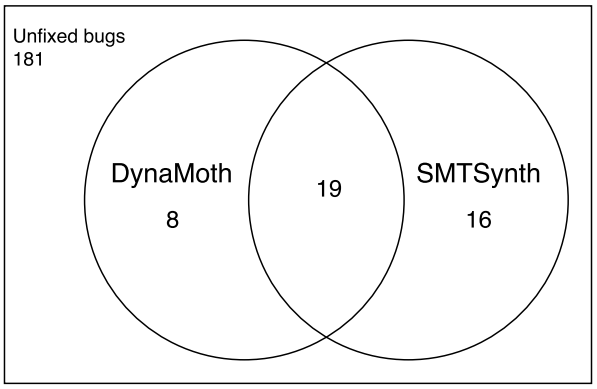

Figure 3.3 The figure illustrates the bugs commonly fixed by DynaMoth and SMTSynth.

Bug fixing Ability **RQ1**. Which bugs of Defects4J can automatically be repaired with DynaMoth?

Table 3.4 presents the bugs of the Defects4J dataset that are fixed either by SMTSynth or DynaMoth. Each line presents a bug of the dataset Defects4J, and each column contains the patched result of each approach. DynaMoth is able to fix 27 (12% of 224 bugs) bugs, SMTSynth 35 (15.6%) bugs. This shows that the new synthesis engine based on dynamic synthesis is not as good as the original one of Nopol based on constraint-based synthesis. In theory, DynaMoth should be able to generate all patches produced by SMTSynth, but due to the complexity of the DynaMoth implementation, there remain bugs for specific complex cases.

However, there are 8 of them are only fixed by DynaMoth. This is explained that the synthesis spaces of Nopol and DynaMoth do not completely overlap. We analytically know that some patches can be synthesized and not by SMTSynth. For instance, those containing method calls with parameters. This empirical result confirms this analysis and is a piece of evidence of the correctness of our implementation. Figure 3.3 presents the decomposition of the 43 bugs fixed by DynaMoth and SMTSynth and, 19 of them are fixed by both techniques.

The results of this experimentation are publicly available on GitHub.[10]

Answer to RQ1. 27 bugs of the Defects4J dataset are fixed by DynaMoth. This shows it applicability on real bugs. Among them, 8 are only fixed by DynaMoth and not by SMTSynth. This shows that SMTSynth and DynaMoth are complementary, one can try both in conjunction for repairing bugs in practice.

[10]The GitHub repository of the experimental data of DynaMoth: https://github.com/tdurieux/dynamoth-experiments

Table 3.5 Results on the Readability of 58 Bugs in Defects4J for the four repair approaches. Legends: E # expressions, M # method calls and O # operators.

Project	Bug Id	SMTSynth	**DynaMoth**
JFreeChart	C3	E: 2 M: 0 O: 1	–
	C5	E: 2 M: 0 O: 1	**E: 1 M: 1 O: 0**
	C6	–	E: 4 M: 0 O: 3
	C9	–	E: 2 M: 0 O: 1
	C13	E: 2 M: 0 O: 1	–
	C17	–	E: 2 M: 1 O: 0
	C21	–	E: 1 M: 2 O: 2
	C25	E: 2 M: 0 O: 1	E: 1 M: 1 O: 0
	C26	E: 2 M: 0 O: 1	**E: 1 M: 0 O: 0**
Lang	L39	E: 1 M: 0 O: 0	E: 1 M: 0 O: 0
	L44	E: 2 M: 1 O: 1	–
	L46	E: 1 M: 0 O: 0	–
	L51	E: 2 M: 0 O: 1	–
	L53	E: 5 M: 0 O: 5	–
	L55	E: 2 M: 0 O: 1	–
	L58	E: 2 M: 0 O: 1	–
Time	T7	–	E: 1 M: 0 O: 0
	T11	E: 5 M: 0 O: 5	**E: 2 M: 0 O: 1**
Math	M32	E: 8 M: 6 O: 8	**E: 0 M: 1 O: 0**
	M33	E: 2 M: 1 O: 1	**E: 1 M: 1 O: 1**
	M40	E: 6 M: 0 O: 5	–
	M41	–	E: 2 M: 0 O: 1
	M42	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M46	–	E: 2 M: 2 O: 1
	M49	E: 2 M: 0 O: 1	E: 3 M: 1 O: 1
	M50	E: 2 M: 0 O: 1	E: 2 M: 1 O: 0
	M57	E: 3 M: 0 O: 1	–
	M58	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M69	E: 3 M: 0 O: 2	–
	M71	E: 4 M: 1 O: 3	**E: 2 M: 0 O: 1**
	M73	E: 6 M: 0 O: 6	–
	M78	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M80	E: 2 M: 0 O: 1	–
	M81	E: 2 M: 0 O: 1	–
	M82	E: 2 M: 0 O: 1	–
	M85	E: 3 M: 0 O: 2	**E: 2 M: 0 O: 1**
	M87	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M88	E: 2 M: 1 O: 1	**E: 2 M: 0 O: 1**
	M96	–	E: 3 M: 0 O: 2
	M97	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M99	E: 2 M: 0 O: 1	E: 2 M: 0 O: 1
	M104	E: 8 M: 0 O: 6	–
	M105	E: 6 M: 0 O: 5	**E: 3 M: 1 O: 1**
Total		E: 103 M: 10 O: 70	E: 61 M: 12 O: 23
Avg.		E: 2.94 M: 0.28 O: 2	E: 2.26 M: 0.44 O: 0.85

Table 3.6 The Execution time of all Defects4J bugs.

	Average	Median	Min	Max
SMTSynth	0:37:47	0:36:49	**0:00:36**	1:23:34
DynaMoth	0:38:01	0:36:31	0:01:34	1:28:05

Table 3.7 The Execution time of patched Defects4J bugs.

	Average	Median	Min	Max
SMTSynth	**0:05:53**	**0:01:02**	**0:00:36**	0:44:53
DynaMoth	0:08:31	0:03:05	0:01:34	0:53:44

Patch readability **RQ2**. Are the synthesized patches simpler than those generated by SMTSynth?

We compare the readability of patches generated by SMTSynth and DynaMoth. Table 3.5 shows the number of expressions (E) which are variables and constants, method calls (M) and operators (O) in each patch. In 9/19 cases, DynaMoth generates patches that contain fewer elements, hence that are easier to read. In 7/19 cases, DynaMoth generates patches that contain an equals number of elements. In 3/19 cases, DynaMoth generates patches that contain more elements. In average DynaMoth contains less expressions (2.26 vs 2.94), less operators (0.85 vs 2) but more method calls (0.44 vs 0.28).

This quantitative result is confirmed by the manual analysis. The manual analysis shows that 22/27 patches synthesized by DynaMoth are easy to read and 5/27 to medium. As expected, there is a clear relationship between the small number of expressions and method calls in the patches and the readability.

Answer to RQ2. DynaMoth synthesizes patches that are simpler and easier to read compared to SMTSynth.

Performance **RQ3**. How long is the execution of DynaMoth on one bug?

The evaluation of DynaMoth has been executed on a cluster if machines with Intel Xeon X3440 Quad-core processor and 15GB of RAM.

Table 3.6 shows the total execution time of Nopol and DynaMoth for all bugs. Table 3.7 shows the execution time for the bugs which are repaired. The average execution time of all bugs is similar for DynaMoth and SMTSynth. For the repaired bugs (Table 3.7), DynaMoth is slower than the SMT-based synthesis of Nopol.

Answer to RQ3. The execution time of DynaMoth is comparable to that of SMT-Synth for all bugs but slower when we only consider the fixed bugs. However, the average repair time remains acceptable for classical repair scenarios.

3.2.3 Conclusion

In this section, we presented DynaMoth, a new patch synthesizer for Java. The synthesis algorithm starts by collecting the execution context of suspicious statements. The DynaMoth explores a rich search space of the combination of existing values. Several optimizations are necessary to tackle the large size of the search space.

The system has been evaluated on 224 bugs from the Defects4J dataset. DynaMoth synthesizes patches for 27 of them, incl. 8 which have never been repaired before. In total, DynaMoth fixes 27 bugs from the dataset. We consider those results are encouraging. In the future, we plan to implement additional optimizations, in particular, to address the combinatorial explosion of the search space caused by methods accepting many arguments and the presence of many candidate EExp to be used as parameters.

3.3 Summary

In this chapter, we have presented three automatic patch generation techniques that do not require a manually written failing test case to generate patches. The state-of-the-art techniques on patch generation use the failing test case as the specification of the incorrect behavior. Unfortunately, a developer is required to understand the bug and to specify a failing test case that exposes the bug. This requirement reduces the applicability of an automatic repair approach. The solution we proposed is to replace the humanly written failing test case by the runtime state of the application. The state of the application is constituted of all the user interactions and all the user inputs that lead to the invalid behavior. Consequently, the state of the application can be used as the specification of the incorrect behavior.

Three different techniques that use the state of the application to generate patches have been shown in this chapter. The first technique uses the stacktrace of a failure to locate the failure, and a template-based technique generates a patch based on such failure location. The second technique monitors the variables and the execution of an application to prevent failure during its execution. When a failure is detected, it uses

the available variables and the nine predefined repair strategies to generate patches. The final technique directly connects itself to the state of the application using the debug interface of the application. The debug interface allows the patch generation technique to manipulate and collect the state of the application. This technique is specialized to generate patches that contain method calls.

We evaluated these three techniques on two different benchmarks. The two first techniques were evaluated on a benchmark of 16 real null pointer exception bugs from Apache libraries. The template-based approach generated 127 patches and the second approach generated 175 patches. The last technique has been evaluated on Defects4J benchmark [63], and it was able to generate 27 patches. Those results show the possibility to generate patches by only using failing execution and the state of the running application.

Chapter 4

A Study of the Runtime Repair Search Space

In the previous chapter, we presented NPEfix, a metaprogramming approach that generates patches for null pointer exceptions. In this chapter, we exhaustively explore the repair search space of NPEfix in order to obtain a precise picture of the size of the repair search space and the number of valid patches in that search space. The repair search space is defined by Martinez [12] as "*all explorable bug fixes for a given program and a given bug (whether compilable, executable or correct)*". This characterization is important to address the two first problems of this thesis: the generation of patches without failing test case and automatic validation of patches. It is important, first, to confirm the finding presented in Chapter 3 on the feasibility of generating patches for bugs that do not have a human written failing test case. Second, it is important to address the problem of filtering the generated patches automatically before presenting them to developers. This study is required to have a better picture of the proportion of valid patches in the repair search space, which helps us to create new techniques to validate patches.

In this chapter, we use NPEfix to generate all the possible patches in its search space for each bug of our benchmark. A repair search space of NPEfix exists because a failure can hide another failure in a different location of the application. We call this type of failure as cascading failure since one failure will lead to another failure. For example, a null variable is used in two different locations inside a method. Handling the first failure will expose the second failure of the application. The repair search space is, in this case, the combination of all the potential repair candidates at both locations. A more concrete

example is presented in Section 4.1. NPEfix is able to handle this type of bug because it instruments the application and injects the repair strategies where the null pointer exception can happen. It is the same behavior as failure-oblivious computing. The only difference is that NPEfix is able to generate patches where traditional failure-oblivious computing approaches are not.

Failure-oblivious computing is one of the approaches for fault tolerance [7, 102, 104, 97]. Failure-oblivious computing techniques modify the execution of a program to reduce the impact of the software failures: instead of crashing the whole system, only the current task fails and the system remains available. For example, Rinard et al. [7] have proposed a failure-oblivious model consisting of skipping erroneous accesses happening out of an array's bounds. Another example is the probabilistic memory safety in DieHard [97], where the execution modifications are controlled by adding blank memory paddings.

The remainder of this chapter is organized as follows. Section 4.1 motivates the study of cascading null pointer exceptions. Section 4.2 details our exploration algorithm. Section 4.3 details the evaluation on 16 field null pointer exception. Section 4.4 details the threats to validity of the study. Finally, Section 4.5 presents the conclusions.

This chapter contains material published in the proceedings of ICST'18 [15].

4.1 Cascading Null Pointer Exceptions

Let us consider the example in Listing 4.1, which is an excerpt of server code that retrieves the last connection date of a user and prints the result to an HTML page. Method `getLastConnectionDate` first gets the user session, and then retrieves the last connection date from the session object. This code snippet can possibly trigger two failures that can crash the request: 1) if the session does not exist and `getUserSession` returns null, then there is a null pointer exception at line 3 (NPE1), and 2) for the first connection, `getLastConnection` returns null, and another null pointer exception can be thrown at line 6 (NPE2).

Now let us consider the possible execution modifications to overcome the failures. In Listing 4.1, to overcome NPE1 at line 3, a failure-oblivious system could modify the execution state and flow in three ways: 1) it creates a new session object on the fly, and 2) it returns an arbitrary Date object such as the current date, and 3) it returns null. As the example suggests, *there are multiple possible failure-oblivious strategies for the same failure*. However, note that not all such modifications are equivalent. For instance, if

modification #3 is applied, it triggers another failure NPE2, whereas solutions #1 and #2 do not further break the system state. This indicates that not all state modifications are equivalent, some being invalid.

In this chapter, we define a conceptual framework and an exploration algorithm to reason about the presence of multiple competing execution modifications in response to a failure.

4.2 Exploring the Repair Search Space

We have shown that there is an implicit search space in failure-oblivious computing. In this section, we formally define this search space and devise an algorithm to explore it exhaustively.

4.2.1 Basic Definitions

Failure-oblivious computing [7] is the art of changing a program's state or flow during execution such that a crashing failure does not happen anymore and that the program is able to continue its execution, and it is sometimes referred to as runtime repair [114], state repair [34] and self-healing software [82]. As the terminology of failure-oblivious computing can hardly be considered as stable, we thus first define its core terms and concepts.

Definition 1. A failure pointis the location in the code where a failure is triggered. In the simplest case, a failure point is a statement at a given line.

Definition 2. A failure-oblivious modeldefines a type of failure and the corresponding possible manners to overcome the failure. For instance, a failure-oblivious model that considers out-of-bounds write in arrays can skip the array upon failures.

```
1 Date getLastConnectionDate() {
2     Session session = getUserSession();
3     return session.getLastConnection(); // NPE1
4 }
5 ...
6 HTML.write(getLastConnectionDate().toString()); // NPE2
```

Listing 4.1 Code excerpt with two potential null dereference failures.

Definition 3. A failure-oblivious strategydefines how a failure is handled. The traditional failure-oblivious literature assumes that there is one single way to be oblivious to a failure. However, there are types of failures for which there exist multiple failure-oblivious strategies. For example, upon an out-of-bounds read in an integer array, one can return either a constant or a value that presents somewhere else in the array. This makes two different strategies.

Definition 4. A context-dependent failure-oblivious strategyis a strategy that can be instantiated in multiple ways depending on the execution context. For example, upon an out-of-bounds read in an integer array, one can return the first, the second, ..., up to the n^{th} value that presents in the array, meaning that there are n ways to instantiate this strategy.

Definition 5. A failure-oblivious decisionis the application of a specific failure-oblivious strategy to handle a failure at a specific failure point. A failure-oblivious decision is an execution modification.

4.2.2 The Failure-oblivious Computing Search Space

Taking a failure-oblivious decision means exploring a new program state after the first failure. The execution proceeding from this new program state can result in a new failure, for which a failure-oblivious decision has to be taken as well.

Definition 6. A failure-oblivious decision sequenceis a sequence of failure-oblivious decisions that are taken in a row during one single execution because of cascading failures.

In this chapter, an execution consisting of a single failure-oblivious decision is called a unary decision sequence, and a composite (or n-ary) decision sequence contains at least 2 decisions. We use Listing 4.1 to give an example of n-ary failure-oblivious decision sequence. At line 3, the execution of the method can be stopped by returning null to overcome a null pointer exception when the session is null, later on in the execution, a new Date can be crafted to handle the second null pointer exception at line 6. In certain cases, only one decision in isolation may not be enough to overcome the failure, only the sequence is a solution.

The notion of context-aware decision and decision sequence naturally defines a search space.

Definition 7. The failure-oblivious computing search space of a failure-triggering input is defined by all possible decision sequences that can happen after the first failure.

4.2.3 FO-EXPLORE: An Algorithm to Explore the Failure-oblivious Computing Search Space

After defining the failure-oblivious computing search space, we now present an algorithm to explore this search space exhaustively. The algorithm is named FO-EXPLORE and is shown in Algorithm 4.1.

4.2.3.1 Algorithm Input-Output

FO-EXPLORE requires four inputs which we explain in detail below.

Program P: a program to which failure-oblivious support will be injected. The program is automatically transformed so as to support failure-oblivious execution. The transformation also adds intercession hooks to steer and monitor failure-oblivious decisions.

Failure Triggering Input I: an input triggering a runtime failure. We assume that we have at least one program input that enables us to reproduce automatically the failures as many times as we want. An input can be a set of values given to a function. In an object-oriented program, an input is much more than values only, it is a set of created objects, interacting through a sequence of method calls.

Failure-oblivious Model R: a model containing the possible modifications of the program state or execution flow to handle the failure as defined above.

Validity Oracle O: an oracle specifying the viability of the computation if failure-oblivious computing happens. A validity oracle is a predicate on the program state at the end of program execution on the failure-triggering input. The goal of the validity oracle is to validate or invalidate the failure-oblivious decision sequence that has happened during execution. In failure-oblivious computing, the traditional validity oracle is the absence of crashing errors, but more advanced ones can be defined (e.g., use additional assertions). For instance, consider again the example in Listing 4.1, where the validity oracle is the presence of an exception that results in HTTP code 500. Returning null is a failed unary

decision sequence because the request crashes with NPE1. On the contrary, returning a fresh date object enables the request to succeed and the HTML to be generated, making it a valid unary decision sequence.

With these four inputs, FO-EXPLORE outputs a set of failure-oblivious decision sequences along with their validity.

4.2.3.2 Algorithm Workflow

Algorithm 4.1 The core exploration protocol FO-EXPLORE.

Input: P: a program
Input: I: an input for P
Input: R: a failure-oblivious model
Input: O: a validity oracle
Output: S: a set of failure-oblivious decision sequences

```
1:  failurePoints ← ∅
2:  decisions ← map_{failurePoint→decision}
3:  while last decision sequence was unknown do
4:     seq ← ∅
5:     exec P(I)
6:     while failure f happens according to R do
7:        if f ∉ failurePoints then
8:           failurePoints ← failurePoints ∪ f
9:           decisions[f] ← computePossibleDecisions()
10:       end if
11:       select an unexplored or a not completely explored decision d
12:       seq ← seq + d
13:       apply d (change the state or flow)
14:       proceed with execution
15:    end while
16:    store pair (decisionsequences, O(seq)) in S
17: end while
```

To explore the search space, the basic idea behind FO-EXPLORE is to make a different decision according to the failure-oblivious model under consideration for each time a failure point is detected and then collect all decision sequences.

We now give a detailed description of the workflow. To explore all the failure-obvious decisions sequences (line 3), FO-EXPLORE executes the input that triggers the runtime failure as much as it is required (line 5). Each time that the failure is detected at a failure point (f) according to the failure-oblivious model (line 6), FO-EXPLORE checks

whether it has already chosen a decision for the failure point (f) (line 7). If no decision has been taken for the current failure point (f) before, FO-EXPLORE computes all the possible decisions for the current failure point (line 9)[1]. Then FO-EXPLORE selects an unexplored or a not wholly explored decision (d) from the computed available decisions (line 11). More specifically, FO-EXPLORE can face two different selection cases:
Case 1: The failure point has never been detected before, which means that the failure has never happened in this location before. In this case, FO-EXPLORE selects the first decision in the set of possible *decisions*.
Case 2: The failure has already been detected before at this failure point in the program. In this case, FO-EXPLORE has two possibilities: explore a new failure-oblivious decision or use an already used decision that triggers a new failure point which is not exhaustively explored.

Once a decision has been selected, FO-EXPLORE will store it in the current decision sequence (line 12). Then FO-EXPLORE applies the decision and resumes the execution of the program (lines 13-14). At the end of the execution, FO-EXPLORE uses the validity oracle to assert the execution and finally stores the result (line 16).

4.2.3.3 Working Example

We now illustrate the workflow of FO-EXPLORE using the example in Listing 4.1. Figure 4.1 presents the actual decision tree of this example. Recall that when FO-EXPLORE executes the request for the failing input, a null pointer exception called NPE1 will be produced on line 3. This is shown as a circle "NPE1" in Figure 4.1. For this first failure point (NPE1), FO-EXPLORE explores three different decisions to handle the null pointer exception (the three arrows coming out of NPE1 in Figure 4.1): 1) uses `return null` to exit the `getLastConnectionDate` method, 2) uses `return new Date()` to exit the method, and 3) uses `session = new Session()` to initialize the null variable with a new instance and proceeds with the execution of the same method. In cases 2) and 3), the execution does not produce other exceptions.

Now consider that FO-EXPLORE selects the first decision (`return null`) and resumes the execution. This decision later produces a second null pointer exception (NPE2 in Figure 4.1) at line 6 in Listing 4.1. At this failure point, FO-EXPLORE explores two decisions: 1) uses `return` to exit the execution of the method and 2) uses `new Date()` to

[1]FO-EXPLORE uses the original NPEFix [13] algorithm to compute the search space of a single failure point.

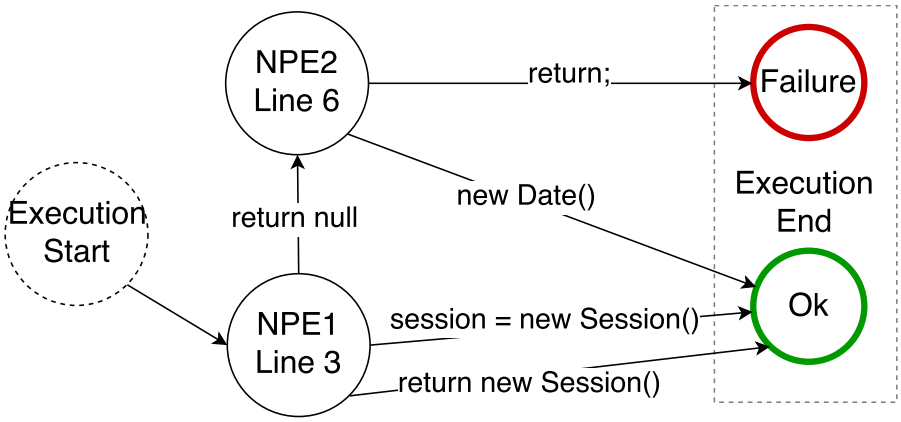

Figure 4.1 Excerpt of the decision tree of the example Listing 4.1. One path is this tree is a "decision sequence". NPEX refers to a failure point.

replace the null expression `getLastConnectionDate()` with a new instance of `Date`. The latter execution modification succeeds while the former produces a failure. The former execution modification fails as if FO-EXPLORE selects the `return` strategy at failure point of NPE2, no response is sent to the client so that a timeout will be produced on the client side.

At the end of the exploration, FO-EXPLORE eventually discovers four different decision sequences (the four different paths from Execution Start to Execution End in Figure 4.1), and three of them produce acceptable output.

4.2.4 Usefulness of Exploring the Search Space

Characterizing the search space of failure-oblivious computing can provide sound scientific foundations for future work on failure-oblivious computing. Indeed, there is very little work that studies to what extent and why failure-oblivious computing succeeds. By clearly defining and exploring the search space, we obtain comprehensive data about this unresearched phenomenon. The empirical results presented in Section 4.3 is the first step towards this direction.

Besides, by identifying multiple acceptable failure-oblivious decision sequences, it opens a radically new perspective on failure-oblivious computing: there may be specific decision sequences that are better than the others. In other words, there are cases where one adds criterion on top of the validity oracle, and this criterion is used to select the best failure-oblivious decision sequence. For instance, the best decision sequence may be the one that runs the fastest. In the presence of multiple acceptable decision sequences,

one needs to characterize and explore the search space first in order to select the best failure-oblivious decision sequence.

4.3 Empirical Evaluation

The goal of the empirical evaluation presented in this section is to study the failure-oblivious computing search space of real failures occurring in large-scale open-source software. This study is built on three research questions about the topology of the search space.

RQ1. [Multiplicity] Does it exist multiple failure-oblivious decision sequences for a given failure-triggering input? How large is the corresponding search space? We want to understand if it is possible to apply different failure-oblivious strategies to handle one specific failure. And if it is the case, how many different decision sequences can be taken.

RQ2. [Fertility] What is the proportion of valid failure-oblivious decision sequences? In the context of failure-oblivious computing, there may exist different failure-oblivious decision sequences that are all valid, i.e., they all fix the runtime failure under consideration. To us, the proportion of valid decision sequences can be considered as the "fertility" of the search space. When the goal is to find at least one valid decision sequence, it is much easier to do so if there are many points in the search space that are valid. On the contrary, if there is only one valid decision sequence in the search space, it requires in the worst case visiting the complete search space before finding the only valid decision sequence. The fertility of the search space is the opposite of what is called "hardness" or "constrainedness" in combinatorial optimization.

RQ3. [Disparity] To what extent does the search space contain composite decision sequences? Our protocol identifies a set of various failure-oblivious decision sequences, and some of them require several decisions in a row. We will observe whether there exist such composite failure-oblivious decision sequences in our benchmark.

4.3.1 Considered Failure-Oblivious Model

We have implemented FO-EXPLORE for null dereferences in Java (aka null pointer exceptions) with a failure-oblivious model derived from NPEfix Section 3.1.3. For more details on the strategies and the implementation, we refer the readers to the NPEfix Section 3.1.3.

Table 4.1 Dataset of 16 bugs with null dereference in six Apache open-source projects.

Bug ID	SVN revision	LOC	# method calls before null
Collections-360	1076034	21650	13
Felix-4960	1691137	33057	2
Lang-304	489749	17277	2
Lang-587	907102	17317	10
Lang-703	1142381	19047	9
Math-1115	1590254	90782	328
Math-1117	1590251	90794	342
Math-290	807923	38265	88
Math-305	885027	38893	8
Math-369	940565	41082	7
Math-988A	1488866	82442	136
Math-988B	1488866	82443	134
PDFBox-2812	1681643	67294	37
PDFBox-2965	1701905	64375	54
PDFBox-2995	1705415	64821	37
Sling-4982	1700424	1182	2
Total: 16 failures from 6 open-source large projects		770721	1209

4.3.2 Benchmark

We need real and reproducible production failures to conduct the evaluation. To achieve this, we reuse the benchmark of NPEFix [13] consisting of 16 field failures coming from Apache projects. Table 4.1 presents the core statistics of our benchmark. The first column contains the bug id. The second column contains the SVN revision of the global Apache SVN. The third column contains the number of lines of code. The fourth column contains the total number of method calls before the null pointer exception is triggered. For example, issue Collections-360 fixed at revision 1076034 is an application of 21650 lines of Java code.

Let us dwell on the last column, the number of executed method calls before the dereference happens. It gives an intuition on the complexity of the setup required to reproduce the field failure. As shown in Table 4.1, there are between 2 and 342 methods (application methods, not counting JDK and API methods) called for the reproduced field failures under consideration, with an average of 75.56. It indicates that the failures in our benchmark are not simple tests with a trivial call to a method with null parameters.

4.3.3 Experimental Protocol

The experiment is based on the exhaustive exploration of the search space of failure-oblivious decision sequences, as defined by our failure-oblivious model for null pointer exceptions described in Section 4.3.1. It is done on the benchmark of failures presented in Section 4.3.2.

4.3.3.1 Exhaustive exploration

We apply algorithm FO-EXPLORE to build the complete decision tree of all failures in our benchmark. Recall that the exploration of the failure-oblivious research space is done as follows:

1. we instrument each buggy program of our benchmark with our failure-oblivious model;
2. we execute each instrumented program with the test case that encodes the field failure;
3. we collect all decisions taken at runtime;
4. and finally, we execute the validity oracle at the end of the test case execution.

The time required to perform such an experiment is approximately the size of the search space multiplied by the time for reproducing the failure. Note the alternative computation that comes after the first failure-oblivious decision at the first failure point is added on top of this.

The raw data of this evaluation is publicly available on GitHub[2]. We answer all research questions based on this data.

4.3.3.2 Validity of Failure-Obliviating Decision Sequences

For a given decision sequence taken in response to a failure, we assess its validity according to the oracle. In our experiments, the validity oracle is extracted from the test case reproducing the field failure: a decision sequence is considered as valid if no null pointer exception and no other exceptions are thrown. A decision sequence is considered as invalid if the original null pointer exception is thrown (meaning that there is no possible

[2] https://github.com/Spirals-Team/runtime-repair-experiments/

Table 4.2 Key metrics of the failure oblivious search space according to FO-EXPLORE.

Bug ID	RQ1		RQ2/RQ3				
	# encountered decision points	# possible decision sequence	# valid decision sequence	% decision sequence	| Valid decision seq..|		
					Min.	Med.	Max.
Collections-360	2	45	16	35,5%	2	2	2
Felix-4960	1	10	4	40%	1	1	1
Lang-304	1	7	6	35,5%	1	1	1
Lang-587	1	28	1	3,0%	1	1	1
Lang-703	4	459	130	28,3%	2	2	2
Math-1115	1	5	5	100%	1	1	1
Math-1117	21	51785	7708	14,9%	7	8	8
Math-290	1	14	4	28,6%	1	1	1
Math-305	1	4	3	75%	1	1	1
Math-369	2	14	0	0%	—	—	—
Math-988A	3	576	383	66,5%	1	2	3
Math-988B	1	32	17	53,1%	1	1	1
PDFBox-2812	8	294	168	57,1%	1	6	7
PDFBox-2965	1	4	3	75%	1	1	1
PDFBox-2995	1	5	1	20%	1	1	1
Sling-4982	2	16	11	68,7%	1	1	1
Total	51	53298	8460	15,9%	1	1	8

decision at the failure point), or another exception is thrown and not caught. When the test case contains domain-specific assertions beyond whether exceptions occur, we keep them and a decision sequence is considered as valid if all assertions pass after the application execution modifications. It is the case for 14/16 failures.

4.3.4 Results to Research Questions

4.3.4.1 RQ1. [Multiplicity] Does it exist multiple failure-oblivious decision sequences for a given failure-triggering input? How large is the corresponding search space?

We analyze the data obtained with the experiment described in Section 4.3.3.1, consisting of exhaustively exploring the search space of execution modifications for 16 null dereferences. Table 4.2 shows the core metrics we are interested in.

Table 4.2 reads as follows. Each line corresponds to a failure of our benchmark. Each column gives the value of a metric of interest. The first column contains the name of each bug. The second column contains the number of encountered decision points for this failure. The third column contains the number of possible failure-oblivious decision sequences for this failure. The fourth column contains the number of execution modifications (valid decision sequences for which the oracle has stated that the decision sequence has worked). The fifth column contains the percentage of valid decision sequences. The sixth column contains the minimum/median/maximum number of decisions taken for valid decision sequences.

For example, the first line of Table 4.2 details the result for bug Collections-360. To overcome this failure at runtime, there are two possible decision points, which, when they are systematically unfolded, correspond to 45 possible decision sequences, 16 of which are valid according to the oracle. The size of the valid decision sequences is always equal to 2, which means that there must be two decisions taken in a row to handle the failure.

Our experiment is the first to show that there exist multiple alternative decisions to overcome a failure at runtime, as indicated by the number of explored decision sequences. The number is exactly the size of our search space when we conduct an exhaustive exploration. In this experiment, it ranges from 4 decisions (for Math-305 and PdfBox-2965) to 576 for Math-988A and 51785 for Math-1117. Overall, we notice a great variance of the size of the search space.

We also see in Table 4.2 that there is a correlation between the number of activated decision points for a given failure and the number of possible decision sequences. For instance, for Felix-4960, there is only one activated decision point (at the failure point where the null pointer exception is about to happen), and 10 possible decisions can be taken at this point. On the contrary, Math-1117 has the biggest number of activated decision points (21), which also has the biggest number of decision sequences (51785).

This correlation is expected and explained analytically as follows. Once a first decision is made at the failure point (where the null dereference is about to happen), many alternative execution paths are uncovered. Then, a combinatorial explosion of stacked decisions happens. If we assume that there are 5 alternative decisions at the first decision point and that each of them triggers another decision point with ten alternative decisions, it directly results in $5 \times 10 = 50$ possible decision sequences. Now, if we assume n decision points with m possible decisions on average, this results in m^n decision sequences, which is a combinatorial explosion. In general, the size of the search space depends on:

1. the overall number of decision points activated for a given failure,
2. the number of possible decisions at each decision point,
3. and the correlation between different decision points that is the extent to which one decision taken at a decision point influences the number of possible subsequent decisions to be taken.

For failures with a large number of explored decision sequences, it means that failure-oblivious computing unfolds a large number of diverse program states and their corresponding subsequent executions.

Answer to RQ1. We have performed an exhaustive exploration to draw a precise picture of the failure-oblivious computing search space. The result clearly shows that there exist multiple alternative failure-oblivious decision sequences to handle null dereferences. In our experiment, there are 11/16 failures for which we observe more than ten possible decision sequences (column "Nb possible decision sequence") for the same failure and according to our execution modification model, with a maximum value of 51785 (for Math-1117).

4.3.4.2 RQ2. [Fertility] What is the proportion of valid failure-oblivious decision sequences?

We now have a clear picture of the size of the failure-oblivious search space and are interested in further knowing whether there exist multiple valid decision sequences in that space. To do so, we still consider the exhaustive study protocol described in Section 4.3.3.1 whose results are given in Table 4.2. We concentrate mainly on the fourth column which shows the number of valid decision sequences. We compare it against the column representing the size of the search space, i.e., the total number of possible decision sequences. For instance, for Collections-360, the search space contains 45 possible

decision sequences, among which 16 are valid according to the oracle (the absence of null pointer exception and the two assertions at the end of the test case reproducing the failure pass), i.e., a proportion of $16/45 = 36\%$ of decision sequences in the search space are valid.

We notice several interesting extreme cases in Table 4.2. First, there are two failures – Lang-587 and PdfBox-2995 – for which only one valid decision sequence exists. Besides, there is one failure for which all the five decision sequences remove the failure: Math-1115.

Let us dwell on this proportion of valid decision sequences in the search space. This proportion depends on the strength of the considered oracle. In failure-oblivious computing, the oracle that is classically considered is the absence of runtime exceptions: we call this oracle the default oracle. In this experiment, we have an oracle that subsumes this default oracle as we also use assertions at the end of the test that reproduces the failure. We have manually inspected the tests and found that not all tests have equally strong assertions, which partly explains the variations in fertility we observe.

Answer to RQ2. In our benchmark, the proportion of valid decision sequences varies from 0 to 100% (from 0/14 to 5/5 valid execution modifications). This significant variation is due to the strength of the considered oracle, and the complexity of the code at the failure point.

4.3.4.3 RQ3. [Disparity] To what extent does the search space contain composite decision sequences?

We have shown in **RQ2** that there are multiple valid execution modifications. Now, we study their complexity as measured by the number of decisions involved in the execution modification. To do so, we again study the results of the exhaustive study protocol described in Section 4.3.3.1 whose results are given in Table 4.2. We especially concentrate on the column showing that among the set of valid decision sequences, what is the minimum, median and maximum size (recall that the size is the number of decisions in the decision sequence). For instance, for PDFBox-2812, the minimal size in term of decision number among all valid failure-oblivious decision sequences is 1, the median size is 6, and the maximal size is 7.

We have the following findings from this data. First, for 5/16 failures of our benchmark, we see that there exist failure-oblivious decision sequences composed of more than one decision. Since our failure-oblivious model is specific to null pointer exceptions, it means that there exist failures for which the null dereference problem is not solved by the first

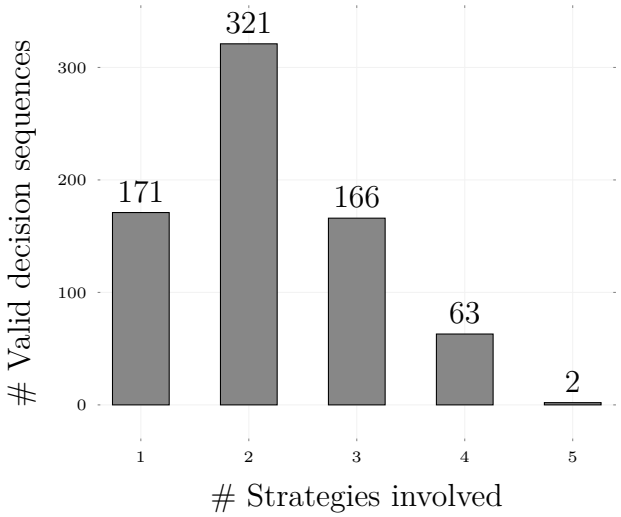

Figure 4.2 The number of valid decision sequences that use between 1 and 5 different repair strategies. We exclude Math-1117 for better visibility.

decision, and that another null dereference happens later. Among the five failures, the failure-oblivious decision sequences are of the same size 2 for two failures (Collections-360, Lang-703). For the remaining three failures (Math-988A, PDFBox-2812, Math-1117), there are failure-oblivious decision sequences of different size. For instance, for Math-988A, there exist failure-oblivious decision sequences of one, two and three decisions. Second, for 10/16 failures of our benchmark, the failure-oblivious decision sequences are always composed of a single decision. It is strongly correlated with the size of the search space (third column, number of decision sequences), indicating that the test case reproducing the production failure sets up a program state that is amenable to failure-oblivious computing. Finally, for one failure (Math-369), even though there are several decisions taken, but none of them are valid. All decision sequences are invalidated by the oracle (the assertions of the test case reproducing the field failure).

An interesting question is whether one needs to apply different strategies in the same execution to get an acceptable result. Figure 4.2 presents the number of different repair strategies that are used in valid decision sequences. We see that 76% of valid decision sequences mix at least two different strategies, this clearly hints that combining different strategies is important for efficient failure-oblivious computing.

Example of Math-988A: Now we illustrate the disparity of the number of decision sequences for the bug Math-988A. The initial null pointer exception is triggered in the return statement of method `toSubSpace` which returns an object of type Vector1D (see line 182 in Listing 4.3). The null pointer exception is triggered when a Vector2D parameter is null

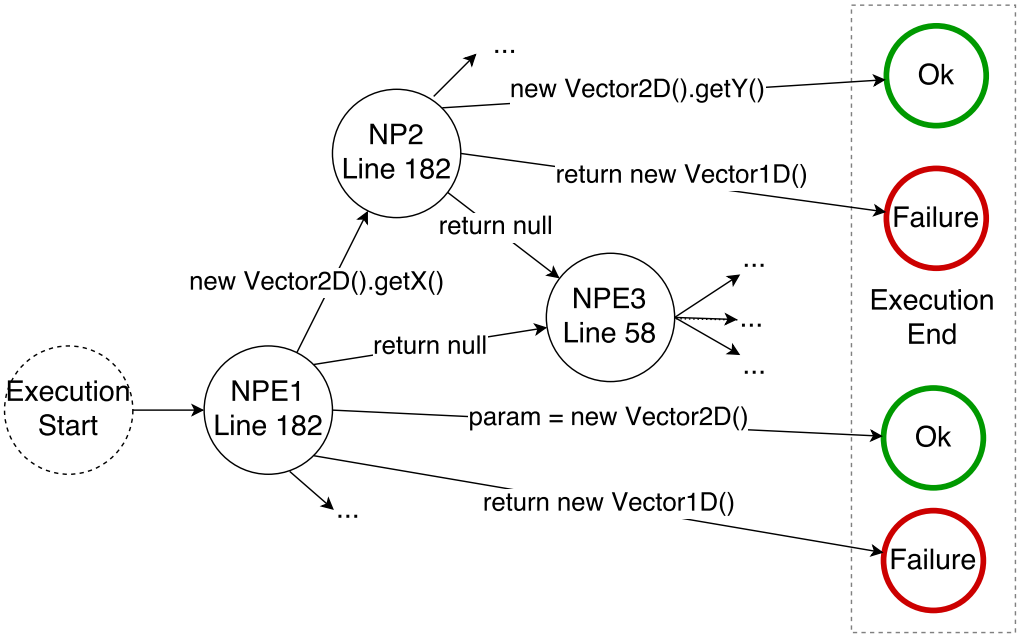

Figure 4.3 Excerpt of the decision tree of Math-988A. One path in this tree is a "decision sequence"'. This figure clearly shows the presence of paths with multiple steps, i.e., the presence of compound decision sequences.

```
   @@ SubLine.java

116  // compute the intersection on infinite line
117  Vector2D v2D = line1.intersection(line2);
118 +if (v2D == null) {
119 +   return null;
120 +}
121
122  // check location of point with respect to first sub-line
123  Location loc1 = getRemainingRegion().checkPoint(line1.toSubSpace(v2D));
```

Listing 4.2 The human patch for Math-988A.

```
   @@ Line.java
181 Vector1D toSubSpace(Vector2D point) {
182   return new Vector1D(cos * point.getX() /* NPE1*/ + sin * p2.getY() /* NPE2*/
        );
183 }

   @@ OrientedPoint.java
 57 double getOffset(Vector2D point) {
 58   double delta = ((Vector1D) point).getX() /* NPE3*/ - location.getX();
 59   return direct ? delta : -delta;
 60 }
```

Listing 4.3 The decision points of Math-988A.

```
   @@ Line.java
181 Vector1D toSubSpace(Vector2D point) {
182 + if (point == null) {
183 +     return Vector1D.NaN;
184 + }
185   return new Vector1D(cos * point.getX() + sin * p2.getY());
186 }
```

Listing 4.4 One valid generated patch for Math-988A.

and methods `getX` and `getY` are called on it. The Vector2D parameter is computed by the line 117 in Listing 4.2 and then passes as argument to the method `toSubSpace` at line 123 in Listing 4.2. If two lines do not have an intersection, the geometrical computation of the intersection of them in line 117 of Listing 4.2 is null and will then cause the null pointer exception.

There are three failure points for this bug as shown in Table 4.2, which means that there are potentially between one and three null pointer exceptions happening during the execution of the failure input depending on the selected decisions. The source code of these failure points are shown in Listing 4.3, and the different decision sequences are illustrated in the decision tree in Figure 4.3. In this figure, each node represents a failure point, each arrow represents a decision, and each path between the Execution Start and the Execution End represents a decision sequence.

For Math-988A, there are different kinds of failure-oblivious decision:

1. initialize the null parameter with a new instance (1 decision point),
2. use a new and disposable instance of Vector2D at both places where the null parameter is used (2 decision points),
3. return null either at the first NPE location or at the second one, triggering another decision in the caller (between 2 and 3 decision points), and
4. return a new instance Vector1D (1 decision point).

For Math-988A, the reproducing test contains JUnit assertions for checking the expected correct behavior, which should return null when no intersection exists. In case the failure-oblivious decisions pass those assertions, it means that failure-oblivious computing achieves full correctness and these decisions are illustrated with “OK” in Figure 4.3. Otherwise, if the failure-oblivious decisions fail on the assertions or trigger another failure, then these decisions do not fully achieve the purpose of failure-oblivious computing and are illustrated with “Failure” in Figure 4.3. As an example, Listing 4.4 shows a generated patch by NPEfix.

Answer to RQ3. For 5/16 failures of our benchmark, the search space contains composite failure-oblivious decision sequences that have more than one decision. For 3/16 failures, the possible failure-oblivious decision sequences have disparate sizes, and our protocol enables us to identify all valid failure-oblivious decision sequences.

4.4 Threats to Validity

We now discuss the threats to validity of our experiment. First, let us discuss internal validity. Our experiment is of computational nature, and consequently, a bug in our code may threaten the validity of our results. However, since all our experiment code is publicly available for the sake of open-science[3], future researchers will be available to identify these potential bugs.

Second, a threat to the external validity relates to the number of failures considered. Our experiment has considered 16 failures from 6 different software projects. Recall that reproducing field failures is a very costly task and consequently, there is a research trade-off between cost and external validity. However, our experiment considers as many or more failures than the related work on failure-oblivious computing. For external validity, it may also be asked whether our results are specific to Java and whether the search space of failure-oblivious computing search space has the same structure in other programming languages and runtime environments. In our opinion, this is an interesting threat to external validity that calls for more research in this area.

4.5 Conclusion

In this chapter, our goal was to characterize the repair search space of NPEfix. We focused on understanding the size and the proportion of valid patches in the repair search space. We performed an empirical study on 16 real Java bugs to draw a precise picture on the nature of the repair search space on real bugs. The important takes away of this chapter are: 1) there are several possible execution modifications to handle a failure; 2) several execution modifications have to be taken in a row to handle some specific failures; and 3) a valid patch can be generated in an acceptable amount of time, even for multi-point patches.

[3] https://github.com/Spirals-Team/runtime-repair-experiments

These results confirm our findings presented in the previous chapter on the viability to generate patches without failing test cases. The results also provide important knowledge towards handling the second problem of this thesis, the automation of patch validation, such as the proportion of valid patches in the repair search space, number of cascading failures and the importance of the context of the failure in the repair.

Chapter 5

Contributions to Automatic Patch Generation in Production

This chapter presents two novel automatic patch generation techniques for the production environment. We create these techniques based on the acquired knowledge from the two previous chapters, and they address our three initial problems: 1) they generate patches without a failing test case; 2) they provide an automatic validation of the generated patches; and 3) they provide solutions to mitigate the side-effects of patch generation techniques.

The two techniques are designed for two different environments. The first technique is a JavaScript client-side patch generation technique. It uses the same principle as the template-based generation technique presented in Section 3.1.2. Instead of targeting null pointer exceptions, the templates of this technique address failures from the JavaScript environment, such as missing libraries or missing HTML element. It uses a novel concept of "repair proxy", which automatically fix the JavaScript code that passes through it. This proxy has three goals: the first goal is to monitor the web applications to detect incorrect behaviors; the second goal is to repair the incorrect behaviors; and finally to monitor the repaired applications to ensure the quality of the generated patches.

The second patch generation technique is designed to generate patches in server-side applications. It also uses a proxy, but in this case, the proxy is used to duplicate the traffic to a sandboxed environment. The sandboxed environment contains the same application as the production environment, but the patch generation technique uses it. The sandboxed environment is also used to validate the generated patches. The duplicated application is patched, and its observable behavior is compared to the original

application. The characteristics of this technique allow the patch generation in the production environment without the potential side-effect of patch generation techniques.

The remainder of this chapter is organized as follows. Section 5.1 presents BikiniProxy, a patch generation technique for JavaScript. Section 5.2 presents Itzal, a patch generation technique for the server-side that provides sandboxing between the production environment and the patch generation technique. Finally, Section 5.3 synthesizes our finding on patch generation in production.

This chapter contains material from papers under submission [16], [18], and a paper available in the proceedings of ICSE NIER'17 [17].

5.1 Production Patch Generation for Client-side Applications

According to [115], at least 76% of all websites on the Internet use JavaScript. The JavaScript code used in today's web page is essential: it is used for social media interaction, dynamic user-interface, usage monitoring, advertisement, content recommendation, fingerprinting, etc., all of this being entirely part of the "web experience". For example, when a user browses the website cnn.com, she is loading more 125 than JavaScript files, which represent a total of 2.8 megabytes of code.

The drawback of this complexity is the growing number of errors in web pages. For instance, a common JavaScript error is due to uninitialized variable, resulting in an error message such as `cannot read property X of null`. Ocariza et al. [116] have performed a systematic study showing that the majority of the most visited websites contain JavaScript errors.

In this contribution, we propose a novel technique to provide patch generation for the web. It is along the line of previous work on automatic repair [34], self-healing software [117, 118], also called failure-oblivious computing (e.g. [7, 13]), automated recovery and remediation (e.g. [119]). The majority of the automatic repair literature focuses on the C and Java runtime. On the contrary, we are interested in the JavaScript/browser runtime, which is arguably much different. Indeed, the topic of automatic repair for the web is a very little researched area [84, 85].

Our novel technique is founded on two insights. Our first key insight is that a proxy between the client browser and the web server can be used for providing repair capabilities

to the web application. Our second key insight is that the most common JavaScript errors can be handled by rewriting the HTML or JavaScript code automatically.

In this contribution, we present a novel repair proxy for the web, called BikiniProxy. It consists of five repair strategies, which are designed explicitly for JavaScript errors. Those strategies are based on rewriting, defined as an automated modification of the code. BikiniProxy automatically modifies the Document Object Model (DOM) of HTML pages or automatically transforms JavaScript abstract syntax trees (AST).

Our approach does not make any assumption on the architecture or libraries of web applications. First, proxy servers are used in most web architectures. Second, our approach does not require a single change to existing web pages and applications. As such, BikiniProxy is highly applicable and could be embedded in a service like Amazon/CloudFlare.

The remainder of this section is organized as follows. Section 5.1.1 explains the background of BikiniProxy. Section 5.1.2 details the BikiniProxy approach. Section 5.1.3 details the evaluation. Section 5.1.4 details the threats to validity of the contribution. Section 5.1.5 concludes this contribution.

5.1.1 Background

5.1.1.1 The Complexity of Today's Web

A web page today is a complex computational object. A modern web page executes code and depends on many different resources. Those resources range from CSS styles of thousands of lines, external fonts, media objects, and last but not least, JavaScript code.

For example, when a user browses the website cnn.com, he is loading more than 400 resources, and 125 of them are JavaScript, which represents a total of 2.8mb of code. Anecdotally, back in 2010, the same web page cnn.com contained 890kb of JavaScript code [116] (68% less code!)

Today's web page JavaScript is essential: it is used for social media interaction, usage monitoring, advertisement, content recommendation, fingerprinting, etc., all of this being entirely part of the "web experience". Consequently, 76% of all websites on the Internet use JavaScript [115]. To this extent, a web page today is a program, and as such, suffers from errors.

5.1.1.2 JavaScript Errors

Web pages and applications load and execute a lot of JavaScript code [120]. This code can be buggy, in fact, the top 100 of the most visited websites contains JavaScript errors [116]. One kind of JavaScript error is an uncaught exception, which is similar to uncaught exceptions in modern runtimes (Java, C#, Python). Those errors are thrown during execution if the browser state is invalid like when accessing a property on a null element (a null dereference).

If the developer does not catch an error, the execution of the current script is stopped. In JavaScript, there is a different "execution scope" for each loaded scripts (i.e., for each HTML script tag) and each asynchronous call. Consequently, contrary to classical sequential execution, in a browser, only the current execution scope is stopped, and the main thread continues running. It means that one can observe several uncaught exceptions for a single page.

The uncaught errors are logged in the browser console that is accessible with the developer tool. Most browsers provide an API to access all the errors that are logged in the browser console. It means that it is relatively easy to monitor JavaScript errors in web applications.

5.1.1.3 Web Proxies

A web proxy is to be an intermediate component between a client browser and web server. In essence, a proxy transmits the client request to the server, receives the response from the server, and forwards it to the client browser. On the web, proxies are massively used for different purposes.

1. A `Network Proxy` is used to expose a service that is not directly accessible, because of network restrictions [121]. The proxy, in this case, is a bridge between two networks and its only task is to redirect requests and responses. For example, a popular network proxy is nginx, it is used to expose websites on port 80 which are indeed served on other ports > 1024, this avoids granting root access to web applications.

2. A `Cache Proxy` is a proxy that is used to cache some resources directly in the proxy in order to improve the serving time to an external resource [122]. The cache proxy stores the response of the server locally and if a request is made for the same resource, the local version is directly sent to the client without sending the request

to the server. A wildly used cache proxy is, for example, a content-delivery network (CDN) that provides optimized access to static resources on the Internet.

3. A `Security Proxy` is used to verify whether a client browser is legitimate to access a server [123]. This type of proxy can be used for example to protect a server against Denial-of-service attacks.

4. A `Load-balancer Proxy` is used on popular applications to distribute the load of users on different backend servers [124]. A load balancer can be as simple as a round-robin, but can also be more sophisticated. For instance, a load-balancer can try to find the least loaded server available in the pool.

5.1.2 Patch Generation for Client-side Applications

We now present BikiniProxy, a novel approach to fully automate repair of HTML and JavaScript in production.

The intuition behind BikiniProxy is that a proxy between web applications and the end-users could provide the required monitoring and intercession interface for automatic error handling. This is the concept of `"repair proxy"` that we explore in this contribution.

5.1.2.1 Architecture

Figure 5.1 presents the architecture of BikiniProxy. BikiniProxy is composed of three main parts.

1. BikiniProxy (see Section 5.1.2.1) is a stateless HTTP proxy that intercepts the HTML and JavaScript requests between the browser (also called "client" in this contribution) and the web server.

2. The JS/HTML Rewriter service (see Section 5.1.2.1) that contains the repair strategies to handle JavaScript errors.

3. The Monitoring and Repair Backend (see Section 5.1.2.1) stores information about the known errors that have happened and the success statistic of each repair strategies for each error.

Let us start with a concrete example. Bob, a user of the website http://foo.com browses the page `gallery.html` and uses BikiniProxy to improve his web experience. Since BikiniProxy is a proxy, when Bob opens `gallery.html`, the request goes through the

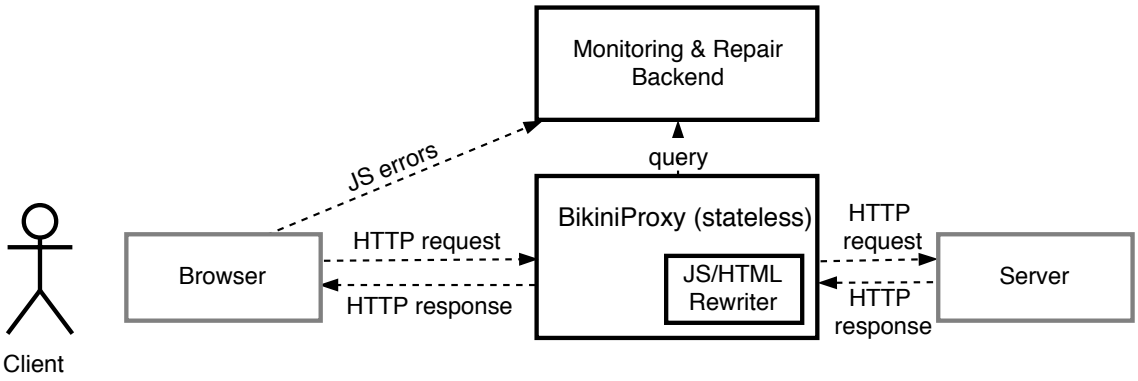

Figure 5.1 The Architecture of BikiniProxy. The key idea is that all requests are proxied by "BikiniProxy". Then, if an error is detected, a repair strategy based on HTML and/or JavaScript rewriting is automatically applied.

proxy. When the request is made, BikiniProxy queries the backend to know whether another, say Alice, has experienced errors on `gallery.html`. Indeed, Alice's browser got a `jQuery is not defined` error two days before. The backend sends this error to the proxy, which consequently launches the JS/HTML Rewriter service to handle the error. For `gallery.html`, the rewriting is HTML-based and consists of by injecting the library jQuery in the HTML response. BikiniProxy also injects its error monitoring framework before sending the rewritten response to Bob's browser. Bob's browser executes the rewritten page, BikiniProxy's monitoring tells the proxy that Alice's error does not appear anymore, meaning that the rewriting strategy made it disappeared.

Algorithm 5.1 shows the workflow of BikiniProxy. BikiniProxy receives the HTTP request from the browser (line 1). Then it redirects the request to the Web Server (line 2) like any proxy. For each HTML response, BikiniProxy injects a framework (line 4) to monitors the JavaScript errors in the client browser. When an error happens on the client browser, it is sent to BikiniProxy's backend for being saved in a database.

BikiniProxy uses the backend to know which JavaScript resource has thrown an error in the past: for each HTML and JavaScript resource, BikiniProxy queries the backend service with the URL of the requested resource to list all the known errors (line 6). If there is at last one known error, BikiniProxy triggers the JS/HTML Rewriter service to apply the repair strategies on the fly to the requested content (line 10). Then the response is sent to the client (line 15) with a unique id to monitor the effectiveness of the applied repair strategy.

Algorithm 5.1 The main BikiniProxy algorithm.

Input: B: the client browser
Input: W: the Web Server
Input: R: the rewriting services
Input: D: BikiniProxy Backend

```
1:  while new HTTP request from B do
2:    response ← W(request)
3:    if request is html page then
4:      response ← inject_bikiniproxy_code(response)
5:    end if
6:    errors ← D.previous_errors_from(request_url)
7:    if errors is not empty then
8:      for r in R do
9:        if isToApply(r, errors, request, response) then
10:         response ← r.rewrite(response, request, errors)
11:         response ← response + uuid
12:       end if
13:     end for
14:   end if
15:   send(response)
16: end while
```

Definition 8 (Resource)**.** A web **resource** is a content on which a web page is dependent. For instance, an image or a JavaScript script is a web resource. In this contribution, a web resource is defined by 1) an URL to address the resource; 2) its content (whether text-based or binary-based) and 3) the HTTP headers that are used to serve the resource. The resource can be used as an attribute of an HTML tag (<script>, <img>, <link>, <iframe>, etc.) or used as an AJAX content.[1]

The Proxy A proxy intercepts the HTML code and the JavaScript code that is sent by the web server to the client browser. By intercepting this content, the proxy can modify the production code and therefore change the behavior of the web application. One well-known example of such a change is to minimize the HTML and JavaScript code to increase the download speed.

In BikiniProxy, the proxy automatically changes the JavaScript code of the web application to handle known errors. BikiniProxy is configured with what we call "repair strategies". A repair strategy is a way to repair a certain class of errors automatically,

[1] AJAX means requested programmatically in JavaScript code

see Section 5.1.2.1. We have carefully designed the following five strategies with the goal of achieving maximum impact, i.e., repair the greatest proportion of pages. All strategies were identified by analyzing the DeadClick benchmark that we use in the evaluation.

JS/HTML Rewriter The role of the JS/HTML Rewriter is to rewrite the content of the JavaScript and HTML resources in order to: 1) monitor the JavaScript errors that happen in the field 2) change the behavior when a JavaScript resource has been involved in an error in the past. In this contribution, a "known error" is an error that has been thrown in the browser of a previous client, that has been detected by the monitoring feature of BikiniProxy and that has been saved in the BikiniProxy backend (see Section 5.1.2.1).

We design for BikiniProxy, five repair strategies that target the most frequent JavaScript errors that we observe when we craw the Internet.

1. HTTP/HTTPS Redirector that changes HTTP URLs to HTTPS URLs.
2. HTML Element Creator that creates missing HTML elements.
3. Library Injector injects missing libraries in the page.
4. Line Skipper wraps a statement with an if to prevent invalid object access.
5. Object Creator initializes a variable with an empty object to prevent further null dereferences.

Besides, the JS/HTML Rewriter is plugin-based, it can be easily extended with new repair strategies to follow the fast evolution of the web environment. All the rewriting actions can also be presented to the developers in a format of a patch. This patch can then be used by the developers to permanently fix their applications.

HTTP/HTTPS Redirector The modern browsers have a policy to block unsecured content (i.e. HTTP resources) in secured web pages (HTTPS). For example, `https://foo.com` cannot load `http://foo.com/f.js` (no https). The rationale is that the unsecured requests can be easily intercepted and modified to inject scripts that will steal information from the secure page, for example, your banking information.

As of 2018, we still are in a period of transition where both HTTP and HTTPS web pages exist, and some websites provide access to their content with both HTTP and HTTPS protocol. Consequently, it happens that the developers forget to change some

URL in their HTTPS version and those resources are blocked, resulting in incomplete web pages or JavaScript errors.

The repair strategy of HTTP/HTTPS Redirector is to change all the HTTP URL by HTTPS URL in HTTPS pages. By doing this, all resources are loaded, and the unwanted behavior due to blocked resources is fixed.

HTML Element Creator As shown by Ocariza et al. [120], most of the JavaScript errors are related to the DOM. This is especially true when the developers try to dynamically change the content of a specific HTML element using getElmentById, like:

```
1 document.getElmentById("elementID").innerText = "Dynamic content";
```

Since the HTML and the JavaScript are provided in different files, it is not rare that an HTML element with an ID is removed or changed without changing the associated JavaScript code. For example, if DOM element `"elementID"` is removed, `document.getElmentById(...)` returns `null` and the execution results on a null dereference when the property `innerText` is set.

When an error happens, BikiniProxy determines if the error is related to the access of a missing element in the DOM: by looking at the JavaScript code at the failure point. If it is the case, "HTML Element Creator" extracts the query that the JavaScript used to access the element and create an empty and invisible HTML element in the DOM. The JavaScript code then runs without error, and the execution continues without affecting the client.

Library Injector In JavaScript, it is a common practice to rely on external libraries to facilitate the development of the web applications. Some of these libraries are extremely popular and are used by millions of users every day, like jQuery or AngularJS. Sometimes, these libraries are not correctly loaded into web pages, and this produces a very characteristic error: for example, `jQuery is not defined` BikiniProxy parses those errors and determines which library is missing.

To do so, BikiniProxy has an initial offline training phase, missing libraries are simulated on a test website, and reference errors are collected for the top 10 libraries presented in [115]. Based on these reference errors, error parsing rules have been manually written to determine which library is missing. When Library Injector detects that a web page contains an error related to a missing library, it injects the related library in the web page.

Algorithm 5.2 Algorithm to rewrite JavaScript code with "Line Skipper" strategy.

Input: E: error
Input: R: resource
1: (line, column, resource_url) = extractFailurePoint(E, R.body)
2: **if** resource_url is R.url **then**
3: ast = getAST(R.body)
4: elem = getElementFromAst(ast, line, column)
5: wrapElemWithIf(elem)
6: R.body = writeAST(ast)
7: **end if**

The rewritten page contains the missing library, and the web page can be completely loaded.

Line Skipper The errors `XXX is not defined` and `XXX is not a function` in JavaScript are errors that are related to invalid access to a variable or a property. `XXX is not defined` is triggered when an identifier (name of variable / function) is used but was never defined in the current scope. For example if we call `if(m){}` without defining `m`, the execution ends with the error `'m' is not defined`. tool toolfunction is called. For example, the code `var func = null; func()` will trigger the error `func is not a function`.

To avoid these errors, Line Skipper wraps the statement that contains the invalid code with an if that verifies that the element is correctly defined for the first error, `if (typeof m != 'undefined' && m) {if(m){}}`. And to verify that a variable contains a reference to function, BikiniProxy rewrites the JavaScript code as follows `if (typeof func === 'function') {func()}`.

Algorithm 5.2 presents the algorithm of Line Skipper. First Line Skipper extracts the line, column, and URL of the resource from the failure point of the error. Line Skipper verifies that the current request is the resource that contains the error. Then Line Skipper extracts the AST from the JavaScript code and looks for the element that is not defined. Finally, the AST is transformed back to a textual form to be sent back to the client.

Object Creator One of the most frequent error types at runtime is null dereference [125], and this also occurs much in JavaScript [120]. Since JavaScript is an untyped language, all null dereferences happening when setting properties can be handled with a generic empty object. For example, the code `var m = null; m.test = ";` will trigger

the error `Cannot set property test of null` and can be avoided by adding `if (m == null) {m = {};}` before setting `m.test`.

The strategy of Object Creator is to initialize all null variables with a generic JavaScript object before setting a property. When the execution continues, all the properties set in the generic object are readable later in the execution.

Monitoring and Repair Backend The Monitoring and Repair Backend fulfills three tasks. The first task is to receive and store all the JavaScript errors happening on client browsers. The Backend provides an API for BikiniProxy to query if a specific resource (URL) contains known errors.

The second task of the backend is to monitor the effectiveness of the different repair strategies. Each time that the repaired JavaScript code is executed, an event is sent to BikiniProxy backend to keep track of the activation of the different strategies. BikiniProxy also keeps track of the number of errors before and after the repair. This way BikiniProxy can disable a repair strategy on a specific page if it increases the number of errors. The validity of the repair strategy is crowdsourced between the different users of the web application. More user uses the repaired application more the confidence on the patch increases.

The third task is to provide a layout of communication with the developers about the monitored errors and the effectiveness of all repair strategies. For example, the following message can be given to the developer: "The strategy `Library Injector` has injected jQuery 22 times in the page `gallery.html` to handle the error `jQuery is not defined` ". This is valuable information to assist the developers in designing a permanent fix. And BikiniProxy backend also provides a visual interface that lists all the errors that the end-users face during the browsing of the web page.

5.1.2.2 Safety Analysis

BikiniProxy is founded on the core failure-oblivious computing principle [7]: any execution happening after the avoided failure is in essence speculative. This speculative execution must be sandboxed.

The safety guarantees of BikiniProxy are provided by the sandboxing in the browser and on the server side. First, all browsers contain very carefully engineered code to sandbox the execution of JavaScript code. This sandboxing means that 1) the JavaScript

code cannot access or transfer data to other tabs and windows (aka tab sandboxing) 2) the JavaScript code cannot access or transfer data to other websites (cross-domain restrictions) 3) the JavaScript code cannot access to the file-system.

Second, in distributed Internet applications with code running on the server-side and on the client-side, it is known that one cannot trust the execution of the client code. Consequently, the best practice is to protect the server-side state with appropriate checks in the REST API accessed by client-side JavaScript. Those checks form the second sandboxing of speculative execution of BikiniProxy: the unwanted side-effects are confined to the current browser window.

5.1.2.3 Applicability Analysis

The usage of BikiniProxy is practically zero cost, and as such, it is widely applicable. First, it requires no change to the original web pages or applications. Second, the usage of an HTTP proxy in web applications is very common. A BikiniProxy can be set up by: 1) a company in front of their web content; 2) a SaaS-based provider 3) a hosting service. Thus, BikiniProxy is an highly-applicable solution.

5.1.2.4 Implementation

In this section, we present the prototype implementation of BikiniProxy. The source code and the usage examples are publicly available in [126]. The implementation of BikiniProxy's proxy is made in JavaScript, based on a popular open-source proxy, anyproxy, by Alibaba.[2] The repair strategies based on HTML rewriting use htmlparser2[3]. The repair strategies based on rewriting the JavaScript abstract syntax tree (AST) use the library babel.js[4].

5.1.3 Evaluation

In our evaluation, we answer the following research questions.

RQ1. [Effectiveness] How is BikiniProxy effective at automatically fixing JavaScript errors in production, without any user or developer involvement? The first research

[2]anyproxy Github repository: https://github.com/Alibaba/anyproxy
[3]htmlparser2 Github repository: https://github.com/fb55/htmlparser2
[4]babel.js Github repository: https://github.com/babel/babel

question studies if it is possible to handle field JavaScript errors with our proxy based approach. We will answer this question by showing how real-world errors have been handled with one of our implemented repair strategies.

RQ2. [Outcome] What is the outcome of BikiniProxy's repair strategies on the page beyond making the error disappear? In this research question, we explore what are the possible outcomes of BikiniProxy on buggy web pages. We will answer this research question by presenting real-world case studies of different possible outcomes.

RQ3. [Comparison] Do the different repair strategies perform equivalently? Finally, we present to which extent the different repair strategies are used: Which type of errors are handled by the five repair strategies?

5.1.3.1 Protocol

We set up the following experimentation protocol to evaluate BikiniProxy. Our idea is to compare the behavior of an erroneous web page, against the behavior of the same, but repaired page using BikiniProxy. The comparison is made at the level of "web trace", a concept we introduce in this contribution, defined as follows.

Definition 9 (web trace)**.** A web trace is the loading sequence and rendering result of a web page. A web trace contains 1) the URL of the page 2) all the resources (URL, content, see Definition 8) 3) all the JavaScript errors that are triggered when executing the JavaScript resources and 4) a screenshot of the page at the end of loading.

Given a benchmark of web pages with JavaScript errors, the following steps are made. The first step is to collect the web trace of each erroneous web page.

The second step is to collect the new web trace of each erroneous web page with BikiniProxy (Recall that all resources are rewritten by BikiniProxy repair strategies). In addition to the web trace, we also collect data about the repair process: the strategies that have been activated, defined by the tuple (initial error, strategy type).

The third step is to compare for each web page the original web trace against the repaired web trace. The goal of the comparison is to identify whether BikiniProxy was able to heal the JavaScript errors. For instance, the comparison may yield that all errors have disappeared, that is a full repair.

Table 5.1 Descriptive statistics of DeadClick

Crawling stats	Value
# Visited Pages	96174
# Pages with Error	4282 (4.5%)
Benchmarks stats	**Value**
# Pages with Reproduced Errors	555
# Domains	466
# Average # resources per page	102.55
# Average scripts per page	35.51
# Min errors per page	1
# Average errors per page	1.49
# Max errors per page	10
# Average pages size	1.98mb

5.1.3.2 Construction of a Benchmark of JavaScript Field Errors

To evaluate BikiniProxy, we need real-world JavaScript that are reproducible. For each reproducible errors, we want to compare the behavior of the web page with and without BikiniProxy. To our knowledge, there is no publicly available benchmark of reproducible JavaScript errors. Consequently, we create a new benchmark, we call it the DeadClick benchmark. The creation of our benchmark is composed of the following steps:

1. randomly browse the web to discover web pages on Internet that have errors (see Section 5.1.3.2)
2. collect the errors and their execution traces (see Section 5.1.3.2)
3. ensure that one is able to reproduce the errors in a closed environment(see Section 5.1.3.2)

Web Page Finder The first step of the creation of DeadClick is finding web pages that contain errors. In order to have a representative picture of errors on the Internet, we use a random approach. Our methodology is to take randomly two words from the English dictionary and to combine those two words in a Google search request. A fake crawler then opens the first link that Google provides. If an error is detected on this page, the page URL is kept as tentative for the next step. The pros and cons of this methodology are discussed in Section 5.1.4

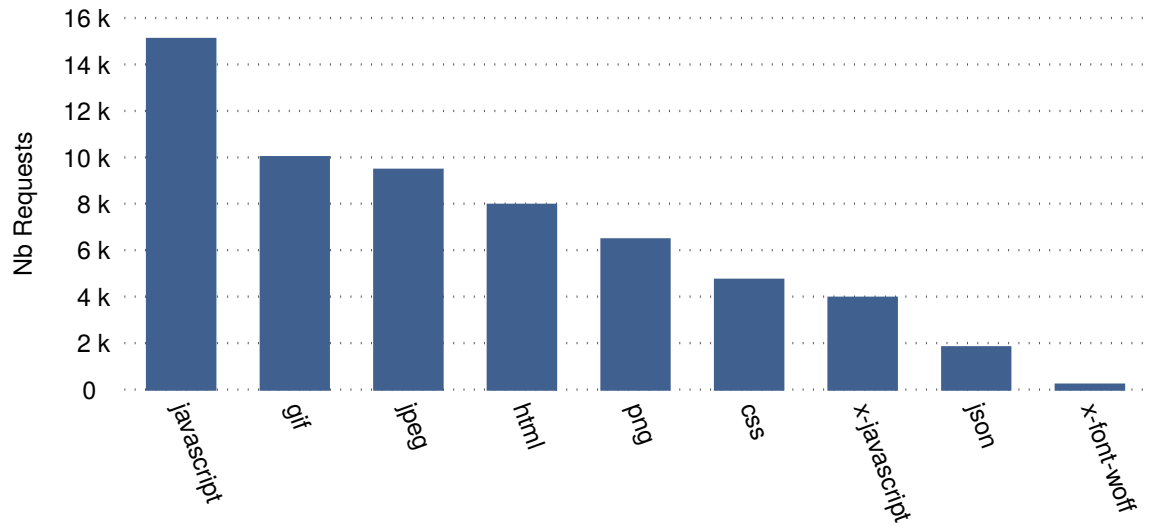

Figure 5.2 Bar plot of the number of requests by content-type.

Web Trace Collector The JavaScript environment is highly dynamic and asynchronous. It means that many errors are transient and as such are not reproducible in the future, even in a very short period after their observation.

For identifying reproducing errors, our idea is to collect the web trace of the erroneous page and to try to reproduce the same web trace in a controlled environment, see Section 5.1.3.2

We implement the trace collection using the library puppeteer from Google[5], which provides an API to control Chrome programmatically. The significant advantage of this library is that it uses the same browser engine as Chrome end-users, meaning that, by construction, DeadClick is only composed of errors that really happen on user browsers.

Since JavaScript is mostly asynchronous, the Web Trace Collector waits for the end of loading where loading is defined as follows: 1) it opens the URL, 2) it waits for seven seconds, in order to load and execute all resources, in particular, JavaScript files. 3) it scrolls the page to the bottom, in order to trigger additional initialization and JavaScript execution. 4) it waits again for one second.

During this process, the Web Trace Collector logs 1) all errors that occur in the browser console and 2) all the requests (including the HTTP headers and the body) made from the browser. When the page is completely loaded, a screenshot of the page is taken, it

[5]puppeteer repository https://github.com/google/puppeteer

Table 5.2 The top 10 error types in DeadClick (left-hand side). The effectiveness of BikiniProxy (right-hand side).

#	Error messages	# Web Pages	# Domains	# Initial Errors	#Healed Errors with BikiniProxy	Improvement
1	XXX is not defined	200	166	307	184	59.93%
2	Cannot read property XXX of null	156	126	176	42	23.86%
3	XXX is not a function	92	86	111	11	9.9%
4	Unexpected token X	54	51	61	2	3.27%
5	Cannot set property XXX of null	21	17	24	11	45.83%
6	Invalid or unexpected token	18	12	21	0	0%
7	Unexpected identifier	13	11	15	0	0%
8	Script error for: XXX	8	3	10	2	20%
9	The manifest specifies content that cannot be displayed on this browser/platform.	5	5	7	0	0%
10	adsbygoogle.push() error: No slot	4	4	7	0	0%
	53 different errors	555	466	826	248	30.02%

provides a visual representation of the page. At the end of this process, for each page, the collected data is stored on disk if at least one error has been logged during the page browsing.

Web Page Reproduction The last step of the benchmark creation consists of verifying that the collected errors can be reproduced. We consider that we succeed to reproduce the behavior of the web page when the observed errors during reproduction are identical to the ones in the originally collected web trace.

The reproduction of the error is done by browsing the erroneous page again but instead of using the resources from the Internet, the Web Page Reproduction is cut from the Internet and only serves the resources stored on disk. In addition, it denies all the requests that have not been observed during the initial collection of the page.

5.1.3.3 Description of DeadClick

Table 5.1 gives the main statistics of DeadClick. The Web Page Finder visited a total of 96174 pages, and 4282 of the pages contains at least one error (4.5%), out of which

555 errors have been successfully reproduced. The final dataset contains errors from 466 different URL domains representing a large diversity of websites. There is on average 1.49 error per page, and each page has between one and ten errors.

Table 5.2 presents the top 10 of the errors present in DeadClick. In total DeadClick contains 53 different error types for a total of 826 collected errors. 69% of the JavaScript errors are the first three error types: `XXX is not defined`, `Cannot read property XXX of null` and `XXX is not a function`. Figure 5.2 presents the number of requests for the top 9 resource types. The external resources are mostly JavaScript files. The rest of the distribution illustrates how complex modern web pages are. For sake open of open-science, DeadClick and its mining framework are available on GitHub [126].

5.1.3.4 RQ1: Effectiveness of Repairing Web Applications

We now present the results of this first research question. Table 5.2 shows the top 10 types of errors in the considered benchmark and how they are handled by BikiniProxy. The first column contains the rank of the error type. The second column contains the error type, represented by the message of the error. The third column shows the number of web pages that contain this errors. The fourth column presents the number of different domains where the error is present. The fifth column contains the initial number of occurrences of the error in the DeadClick. The sixth column contains the number of healed error with BikiniProxy. The seventh column contains the percentage of errors fixed with BikiniProxy.

The first major result lies in the first row. It presents the error "XXX is not defined", which is the most common on the web according to our sampling. This error is present in 200 web pages across 166 different domains. It is thrown 307 times (meaning that some web pages throw it several times). With BikiniProxy, this error is healed 184/307 times, which represents a major improvement of 59.93%.

A second major result is that BikiniProxy is able to handle at least one error for the five most frequent JavaScript errors. It succeeds to heal between 3.27% and 59.93% of the five most frequent JavaScript errors in our benchmark. Overall, 248 errors are automatically healed, meaning that BikiniProxy reduces by 30.02% the number of errors in the benchmark;

Now we discuss the categories of healed errors. We identify whether:

Table 5.3 Analysis of the healing effectiveness per page.

Metric Name	# Pages	Percent
All Errors Disappeared	176/555	31.76%
Some Errors Disappeared	42/555	7.58%
Different/Additional Errors	140/555	25.27%
No Strategy Applied	196/555	35.31%

1. *All errors disappeared:* no error happens anymore in the page loaded with BikiniProxy, meaning that one or a combination of rewriting strategies have removed the errors.
2. *Some errors disappear:*, there are fewer errors than in the original web trace.
3. *Different errors appear:* at least one error still, and it is a new error (new error type or new error location) that has never been seen before.
4. *No strategy applied:* the error type is not handled by any of the strategies and thus there are the same errors than in the original web trace.

Table 5.3 presents the number of web pages per category. The first line of Table 5.3 shows that the number of web pages that have all the JavaScript errors healed by BikiniProxy: BikiniProxy is able to handle all errors for 176/555 (31.76%) of the DeadClick benchmark. The second line shows the number of web pages that have been partially repaired, by partially, we mean that the number of JavaScript errors decrease with BikiniProxy but are still not zero: with BikiniProxy, 42 web pages contain fewer errors than before. The third line shows the number of web pages that have new errors than before: in 140/555 (25.27%) of the faulty pages, BikiniProxy is able to handle some errors but the rest of the execution produce new errors. The last line shows the number of web pages where none of the strategies has been applied: in 196/555 (35.13%) the errors are of a type that is not considered by BikiniProxy. To better understand the case where no strategy can be applied, we perform a manual qualitative analysis.

Case study: Unhandled Errors The errors are unhandled when none of the five BikiniProxy rewriting strategies succeed to heal the errors. In our experiment, 196/555 of web pages have errors not healed, which represents 35.13% of the erroneous web pages of DeadClick. The first cause of non-healed errors is that the error type is not supported. For example, the web page http://dnd.wizards.com/articles/unearthed-arcana/artificer is loading a JSON file. However, the JSON file is invalid, and the browser does not succeed to parse it which produces an `Unexpected token <` error. None of the

five strategies is able to handle malformed JSON errors. The second cause of non-healed errors is that the repair strategies have not enough information to rewrite the resource. For example, the web page http://moreas.blog.lemonde.fr/2007/02/28/le-pistolet-sig-sauer-est-il-adapte-a-la-police/ contains the error `Cannot read property 'parents' of undefined`, this error should be healed with "Object Creator" rewriting. However, the trace of the error does not contain the URL of the resource that triggers this error, because the JavaScript code has been unloaded. Consequently, "Object Creator" is not able to know which resource has to be rewritten to handle the error.

In summary, Table 5.3 shows that BikiniProxy is almost able to heal all the errors from a third of DeadClick. The second third of the benchmark is pages that cannot be healed with BikiniProxy. The last third contains web pages that are partially headed or that the repair strategies produce new errors.

Answer to RQ1. BikiniProxy is effective the handle the five most frequent JavaScript errors present in our benchmark. With the currently implemented rewriting strategies, BikiniProxy can fully heal 248/826 (30.02%) of all errors, representing 196/555 (31.76%) of all buggy web pages of our benchmark.

5.1.3.5 RQ2: Outcome

In this second research question, we focus on category "All Errors Disappeared", and further refines the classification as follows:

1. The errors have disappeared, but the end-user can see no behavioral change.
2. The errors have disappeared, and new UI features (e.g., new buttons) are available to the end-user.
3. The errors have disappeared, and new content is available for the end-user.

Contrary to RQ1, it is not possible to automatically classify all pages with this refined category, because it requires human-based assessment of what is new content or new features. For this reason, we answer this RQ with a qualitative case study analysis.

Case study: Error Handled but No Behavior Change A healed error does not automatically result in a behavior change in the application. For example, this is the case for the website https://cheapbotsdonequick.com/source/bethebot, which triggers the error `"module" is not defined`. This error is triggered by line `module.exports =`

```
Uncaught TypeError:
  Cannot read property 'id' of null
    at bluecava.js?v=1.6:284 ...
    at post (bluecava.js?v=1.6:40)
    at identify (bluecava.js?v=1.6:156) ...
```

Listing 5.1 Error on the web page https://bluecava.com/.

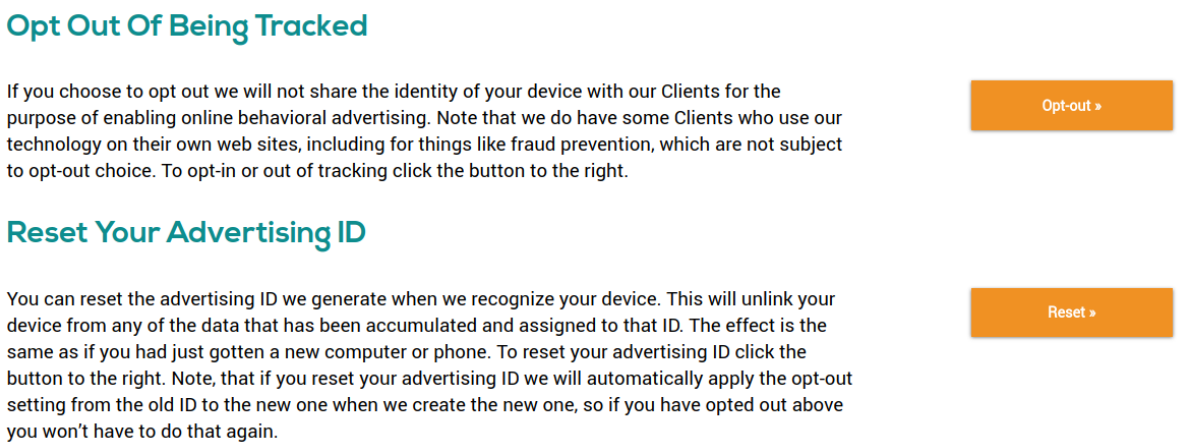

Figure 5.3 The two buttons in orange are missing in the original buggy page. When BikiniProxy is enabled, the two orange buttons provide the user with new user-interface features.

`tracery;`. This type of line is used to make a library usable by another file in a Node.js environment. However, Node.js has a different runtime from a browser, and the module object is not present, resulting in the error. With BikiniProxy, the repair strategy "Object Creator" automatically initializes the variable `module`, however, since this line is the last line of the executed JavaScript file, this has absolutely no further consequence on the execution or the page rendering. This means that the error was irrelevant. However, from a repair perspective, this cannot be known in advance. From a repair engineering perspective, the takeaway is that it is more straightforward to heal irrelevant errors than to try to predict their severity in advance.

Case study: new Feature Available One possible outcome of BikiniProxy is that the repair strategy unlocks new features. For example, this the case of https://bluecava.com/. This page has an error, shown in Listing 5.1, which is triggered because the developer directly accesses the content of Ajax requests without checking the status of the request. However, there are requests that are denied due to cross-domain access restrictions implemented in all browsers. Since the developer did not verify if there is an error before accessing the property 'id' on a null variable, the JavaScript event loop crashes.

With BikiniProxy, the repair strategy "Object Creator" ensures the initialization of the variable if it is null. This execution modification allows the execution to continue and to finally enter into an error handling block written by the developer, meaning that the event loop does not crash anymore. The execution of the page continues and results in two buttons being displayed and enabled for the end-user. Figure 5.3 presents the two buttons that are now available for the user.

Case study: new content available One other outcome of BikiniProxy is that additional content is displayed to the end-user.

Let us consider the web page http://personal.lse.ac.uk/birchj1/ that is the personal page of a researcher. This page triggers the following error: `$ is not defined at (index):20`

This error is thrown because a script in the HTML page calls the jQuery library before the library is loaded. The script that throws the error is responsible for changing the visibility of some content in the page. Consequently, because of the error, this content stays hidden for all visitors of the page.

Using BikiniProxy, the error is detected as being caused by a missing jQuery library. This error is healed by rewriting strategy “Library Injector” Consequently, BikiniProxy rewrites the content of the page by injecting the jQuery library and sends back the rewritten page to the browser. When the rewritten script is executed, jQuery is available, and consequently, the script is able to change the visibility of hidden HTML elements, resulting in newly visible content.

Figure 5.4 presents the visual difference between the page without BikiniProxy (left side) and loaded with BikiniProxy (right side). All the elements in a gray box on the right-hand side are missing on the left image, they have appeared thanks to BikiniProxy.

Finally, we have manually checked the presence of potentially harmful effects (see 5.1.2.2). By manually analyzing a random sample of 25 repaired pages, we did not find a single harmful effect.

Answer to RQ2. We observe three outcomes in our benchmark: (1) no visible change; (2) new features; and (3) new content. BikiniProxy is able to restore broken features or broken content automatically. We have not observed any harmful effect of speculative execution.

(a) Without BikiniProxy, some content is missing

(b) Using BikiniProxy, the loaded page is repaired.

Figure 5.4 A real web page suffering from a JavaScript bug. With BikiniProxy, the bug is automatically healed, resulting in additional information provided to the web page visitor.

Table 5.4 The number of activations of each repair strategy and the number of error types that the strategy can handle.

repair strategies	# Activations	# Supported Error Types
Line Skipper	233	4
Object Creator	109	2
Library Injector	75	3
HTTP/HTTPS Redirector	18	NA
HTML Element Creator	14	2

5.1.3.6 RQ3: Strategies

In this research question, we compare the five different repair strategies of BikiniProxy. For each strategy, Table 5.4 shows the number of times it has been activated to heal errors of DeadClick, and the last column presents the number of different error types for which the strategy has been selected. For example, the first row of Table 5.4 shows that "Line Skipper" has been selected to handle 233 errors, and it has healed four different error types.

The most used strategy is "Line Skipper" with 233 activations. It is also the strategy that supports the highest number of different error type: 1) "XXX is not defined", 2) "XXX is not a function", 3) "Cannot read property XXX of null", 4) "Cannot set property XXX of null". The second most used strategy is "Object Creator" with 109 errors for which it has initialized a null variable. This strategy handles two different error types: "Cannot set property XXX of null" and "Cannot read property XXX of null". These two strategies have something in common, they target the failure point, the symptom, and not the root cause (the root cause is actually unknown). For example, the error `CitedRefBlocks is`

`not defined` is triggered in the web page https://www.ncbi.nlm.nih.gov/pmc/articles/PMC504719/, because the function `CitedRefBlocks` is not defined. Line Skipper strategy avoids the error by skipping the method call, it is a typical example of a fix at the failure point and not at the root cause of the absence of `CitedRefBlocks`.

On the contrary, "Library Injector" addresses the root cause of the problem: the missing library is extracted from the error message, and it is used to rewrite the content of the request. In this case, BikiniProxy exploits the fact that we have a direct relation between root cause (no included library) and symptom (unknown used library name) for this error type.

The case of "HTTP/HTTPS Redirector" is the opposite. Recall that "HTTP/HTTPS Redirector" directly looks in the HTML body of the resource if there are scripts that will be blocked. This means that the rewriting addresses the root cause of potential future problems. For example, the page https://corporate.parrot.com/en/documents tries to load the resource http://www.google-analytics.com/urchin.js, but the request is blocked by the browser (HTTP request in an HTTPS page). Consequently, the Google tracking library is not loaded and function `urchinTracker` is not defined, resulting in the error `urchinTracker is not defined`. “HTTP/HTTPS Redirector” strategy rewrites the URL of the resource in the <SCRIPT>tag to https://www.google-analytics.com/urchin.js, and this fixes the error of the page. This strategy can potentially fix error types that we cannot envision, hence, we do not know the exact number of handled error types, so we put “NA” in Table 5.4.

Finally, strategy "HTML Element Creator" is applied to more rare errors happening only 14 times in our benchmark.

Answer to RQ3. In our experiment, the most used strategy is “Line Skipper” because is able to heal from four common error types with the same strategy. Other repair strategies can be designed and added to BikiniProxy in order to address the rare error types in the long tail of field errors.

5.1.4 Threat to Validity

We now discuss the threats to the validity of our experiment. It is unknown whether the errors of our benchmark are representative of all errors in the web, and whether 96174 visited pages is enough compared to the trillions of pages of the Internet. To our knowledge, there is no work on the representativity of JavaScript bugs.

Our approach has been carefully designed to maximize representativity: 1) the randomness of keyword choice allows us to discover websites about many different topics, done by a variety of persons, with different backgrounds (a website on CSS done by a web developer is likely to have fewer errors than a website on banana culture done by a hobbyist). 2) the ranking of Google provides us with a filter which favors popular websites. If errors are detected on those websites, they likely affect many users. It means that if BikiniProxy heals those errors, it would have a large impact.

5.1.5 Conclusion

In this contribution, we presented BikiniProxy, a novel technique generate patches for the web, focusing on client-side JavaScript errors. We have evaluated our technique on 555 web pages with JavaScript errors, randomly collected on the Internet. Our qualitative and quantitative evaluation has shown that BikiniProxy is effective and can generate patches that fix features and recover missing content.

5.2 Production Patch Generation for Server-side Applications

In modern distributed systems running on the cloud, software failures happen constantly [127]. The leading company in the business of production failure monitoring, called OverOps, has reported that a popular Java web application suffers from 9.2 million exceptions per month on average, due to an average of 53 unique root causes [125].

What about automatically generating source-code patches that would prevent production failures from happening again? We dream of a world where the developer would receive each morning in her GitHub dashboard a list of potential patches that fix certain production failures. This is the blue-sky vision we elaborate in this patch generation technique.

This is fundamentally different from traditional program repair (e.g. [1, 2]). Indeed, traditional program repair is built on premises that are not adequate to fix production failures. First, most repair systems require one or several failing test cases to guide the repair process. But it has been shown that it is extremely difficult to reproduce production failures and translate them into failing test cases [128–130]. Second, traditional program

repair uses a regression test suite to verify that the generated patch has not introduced regressions, with no guarantee whatsoever that the regression test suite covers all the behaviors used in production, resulting in incorrect patches [131, 132, 9].

There is a fundamental gap between the vision of automatically generating patches for production failures and the state-of-the-art of program repair. This is what we address here – we bring program repair to production failures. In this contribution, we specify a novel scheme for program repair in production, and we present the design and implementation of a prototype system for Java, called Itzal.

Itzal works as follows. First, it uses production oracles (such as uncaught exceptions) to detect failures and trigger patch search. Second, right after the failure is detected in production, a patch is searched in a parallel environment that mimics the production one. This search is asynchronous so that patch synthesis has a negligible overhead on the production system. Third, the synthesized patches are validated based on traffic that is an exact copy of the production user traffic – we call it shadow traffic. Patches that fix a production failure and do not introduce regressions that are visible on the end-to-end user traffic are proposed to the developer.

The remainder of this section is organized as follows. Section 5.2.1 presents the patch generation technique: Itzal. Section 5.2.2 presents the evaluation of Itzal. And finally, Section 5.2.3 presents our conclusion.

5.2.1 Patch Generation on Production Failures

We now present Itzal, a novel program repair technique for generating patches directly in the production environment.

Intuition: The intuition behind Itzal is twofold. First, one can use production runtime contracts to drive the generation of source code patches. This includes classical pre- and post-conditions as well as implicit contracts such as that an accessed variable must not be null. The latter is important because the violations of those implicit contracts come for free in any modern runtime, usually in the form of runtime exceptions. The second intuition is that one can use the diversity of the production inputs to perform in-the-field regression testing on the synthesized patches.

Applicability: The requirement to deploy Itzal is that the application has a message-driven architecture [133], i.e., must use requests. The type of requests may vary between applications, it can be for example: 1) a request sent by a user's browser to a web

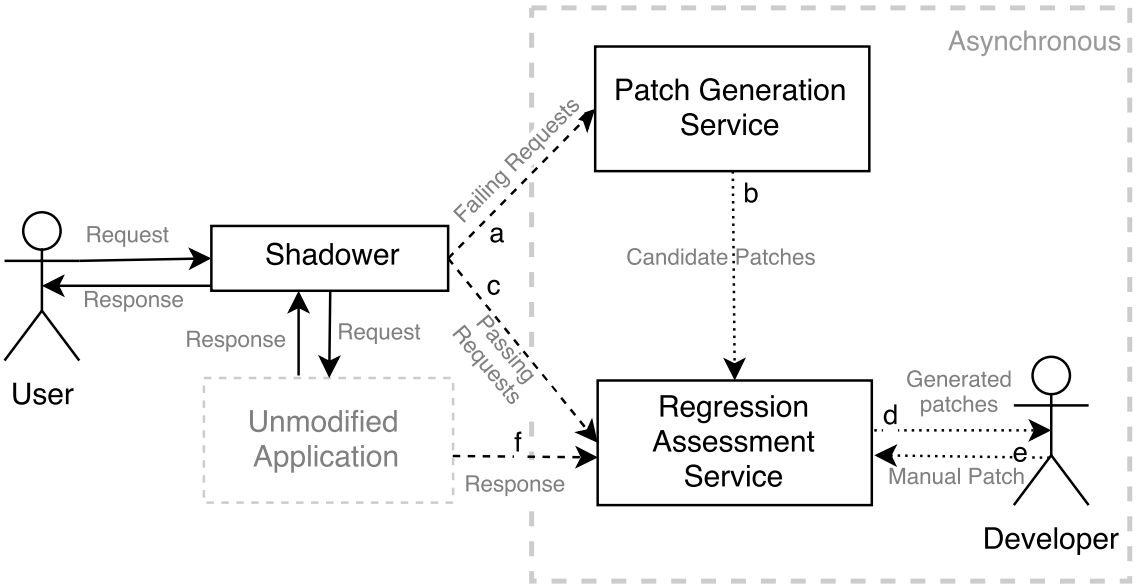

Figure 5.5 The blueprint of Itzal. The key idea is to duplicate user traffic via a "shadower", the duplicated traffic is used to search for patches and to validate candidate patches.

server, 2) a REST message for a micro-service application, 3) for a mobile application, a touch event triggered when a user touches a mobile device's screen. An extreme case of message-driven software is serverless computing, also known as Function-as-a-Service [134], such as Amazon Lambda, where there is no state between requests. One may consider that Function-as-a-Service is a killer application domain of Itzal.

5.2.1.1 Blueprint Architecture

The Itzal architecture is composed of three main components that are set up around an existing unmodified production application, as shown in Figure 5.5.

1. The Shadower is used to duplicate the requests of the Unmodified Application. The duplicated requests are then sent in parallel to the Patch Generation Service and the Regression Assessment Service.

2. The Patch Generation Service is the service that searches for patches that fix a given failure. It uses a failure detection mechanism called "Request-oracle" in this contribution to determine whether the application has successfully handled a request.

3. The Regression Assessment Service performs regression testing on the patches based on user traffic. It applies the generated patches on a copy of the applica-

Algorithm 5.3 The main Itzal algorithm.

Input: A: the Unmodified Application
Input: G: the Patch Generation Service
Input: V: the Regression Assessment Service
1: **while** new request r from Client **do**
2: $output \leftarrow \mathrm{A}(r)$
3: send *output* to Client
4: **if** r produces a failure **then**
5: $patches \leftarrow \mathrm{G}(r)$
6: $failureCount_r++$
7: push *patches* to V
8: **else**
9: send (r,*output*) to V for regression analysis
10: **end if**
11: **if** $\exists$ not "regressive patches" $\in V$ **then**
12: report patches to developers
13: **end if**
14: **end while**

tion – the shadow application – and Shadower duplicates the user traffic to the shadow application in order to observe the behavior of the patched application and potentially detect regressions.

Eventually, the patches generated by Itzal are communicated to the developers, for instance using automated pull requests on GitHub. The developers can directly merge them or further improve them.

Algorithm: Algorithm 5.3 shows the workflow of Itzal. Shadower receives the request from the client (line 1). Then it redirects the request to the Unmodified Application (line 2). Once the request has been handled by the Unmodified Application, the response is sent back to the client (line 3). If the Request-oracle has determined that there is a failure, the request is sent to the Patch Generation Service (arrow a in Figure 5.5 and line 5). The patches generated by Patch Generation Service that pass the Request-oracle (i.e., that fix the failure at hand) are sent to Regression Assessment Service (line 7). If the request has succeeded (i.e., no failure on the original application), the request is also sent to the Regression Assessment Service (line 9) where all the previously generated patches are being regressed on-the-fly against the new requests. When the Regression Assessment Service has identified patches with no regressions, it sends them to the developers.

Patch Generation Service For every request, Itzal verifies whether the application has succeeded to answer the request using a Request-oracle. For instance, in a web server, one can check the return code of HTTP request ("assert response_code != 5XX (internal server error)") or check the presence or not of an exception. Itzal works with generic oracles such as checking the absence of exceptions (e.g., in a web request container or in a thread monitor), and it can also work with domain-specific oracles written by software engineers on top of domain concepts and data (e.g., the returned XML must comply with a specific schema).

For each failing request, a Patch Generation Service searches for patches that prevent the failure or any other ones from happening according to a patch model. On this, Itzal piggy-backs on existing research [7, 13]. Itzal is agnostic to the patch service, it supports several patch models: we have implemented two of them, both used in our evaluation.

Definition 10. A "patch model" enables one to enumerate all patches according to a specification of the search space.

For each explored candidate patch, the Patch Generation Service calls Request-oracle to verify that the request has been correctly handled by the patch under consideration, i.e., the failure has been fixed. As the Patch Generation Service generates the patches based on only one request (the failing one), the patches may break the behavior of the application for other requests, i.e., they may introduce regressions. Thus, if the patch is successful on the failing request, it is transferred to the Regression Assessment Service (arrow *b* in Figure 5.5) which will further regress it on incoming requests.

The execution of candidate patches can change the state of the application in runtime. To nullify the potential side effects of the request or the new behavior introduced in synthesized patches, each execution is done in a completely sandboxed environment. In other words, the side-effects of the execution of the patch candidate never propagate to the production application by construction.

Beyond null dereferences, Itzal can work with any patch model, whether domain-specific (e.g., for out-of-bounds exception [135]) or generic (e.g., Genprog [1]). Note if the patch model generates too many patches (i.e., the search space is too large), it can possibly be a problem as it can incur a huge computation overhead on the Patch Generation Service and much more importantly on the Regression Assessment Service.

Regression Assessment Service The patches generated by the Patch Generation Service can introduce regressions as their generation involves only the failing request. The Regression Assessment Service has the responsibility to check the behaviors of the application when the generated patches are injected on other requests.

The Regression Assessment Service uses an "execution comparison oracle" to compare the output of the Unmodified Application with that of a patched version for the same request. If the outputs are different, the Regression Assessment Service discards the patch and marks it as a "regressive patch". For example, an execution comparison oracle for a web server can compare the HTML texts of both versions.

Definition 11. An "execution comparison oracle" is a mechanism to detect changed behaviors in production.

The comparison is not necessarily a byte-to-byte one, it can include heuristics to discard transient information such as time, cookie identifiers, etc. To increase the accuracy of the regression evaluation, each generated patch is evaluated against a large number of requests, say for example 1 million if there are a large number of users.

The comparison is made on-the-fly, directly on user traffic. Doing regression testing "live" has the advantage that there is no need to record the potentially enormous amount of production data.

Algorithm 5.4 is the main algorithm of the Regression Assessment Service. The Regression Assessment Service requires a copy of the Unmodified Application, the response of the Unmodified Application, an execution comparison oracle, and a list of patches to regress (sent previously by the Patch Generation Service).

For each successful request received from the Shadower (arrow *c* in Figure 5.5 and line 1 in Algorithm 5.4), the Regression Assessment Service iterates over each patch to detect regressions (lines 3-12) in Algorithm 5.4). Finally, the patches that pass user traffic based regression testing are sent to the developers (arrow *d* in Figure 5.5 and line 13 in Algorithm 5.4).

There is a major advantage of doing regression validation on user traffic: the user traffic contains far more usage scenarios and far more diverse values than a regression test suite. Consequently, it reduces the risk of overfitting, i.e. it reduces the risk of suggesting an incorrect patch to the developer.

Algorithm 5.4 The Regression Assessment Service algorithm.

Input: A: Unmodified Application
Input: R: Execution Comparison Oracle
Input: Q: patches from Patch Generation Service
1: **while** new request r from Shadower **do**
2: $output_{ref} \leftarrow$ the output of A for r (from Shadower)
3: **for** patch p in Q **do**
4: $A' \leftarrow$ apply p to A
5: $(output_{A'}) \leftarrow r$ send to A'
6: S $\leftarrow$ R($output_{ref}$, $output_{A'}$)
7: **if** S = false **then**
8: remove p from Q
9: **else**
10: $regressionSuccessCount_p++$
11: **end if**
12: **end for**
13: send {Q, $regressionSuccessCount_p$} to the developer
14: **end while**

We note that the Regression Assessment Service can also be used to validate a human patch with the production traffic, as shown in arrow e of Figure 5.5. In this case, it means that the Regression Assessment Service is used for live testing of code on user traffic.

Shadower The role of the Shadower is to create shadow traffic from actual end-user traffic coming into the application. The "shadow traffic" is made up of production requests that are duplicated one or several times and sent to sandboxed shadow applications. In our case, the shadow applications are the Patch Generation Service and Regression Assessment Service.

In Itzal, the Shadower receives the requests from the clients, duplicates them and sends one duplicate to each service of the architecture (arrows a, c in Figure 5.5). The response is also shadowed for the Regression Assessment Service (arrow f in Figure 5.5).

Definition 12. A "shadower" is a system to duplicate requests of message-driven application.

Definition 13. A "shadow application" is a duplicate and sandboxed copy of a production application, which receives the same requests.

If the production application has a state (typically stored in a database), the shadow application accesses to the production data through a read-only database connection[6]. This guarantees that the shadow application never corrupts the production state, and that patch synthesis remains transparent and safe for the unmodified, deployed application. The drawback is that it prevents repair of code related to state modification. There are sophisticated ways for overcoming this limitation, but this is a hard and unresearched problem which is left to future work.

In the context of web applications, the concept of running multiple instances of an application is well known and heavily used: this is done for load balancing and rolling deployment. The difference between a load balancer and a Shadower is twofold: first, a load balancer does not duplicate the traffic; second, a load balancer does not send requests to sandboxed "sinks" as Itzal does.

Since Itzal is a production technique, it must have a reasonable impact on the performance of the application. In order to minimize the impact on the Unmodified Application, Itzal computes the Regression Assessment Service and the Patch Generation Service asynchronously. Indeed, the goal of Itzal is to perform patch generation, it is not an automatic error recovery system. Hence, the Shadower directly sends the output as soon as the Unmodified Application has handled a request (even if there is a failure). Itzal does not have to wait for the end of the patch search or the regression testing for sending the response back to the client. Consequently, the Shadower is the only component that impacts the performance of the Unmodified Application.

In a typical HTTP-based setup, the cost of copying and rerouting requests on-the-fly is similar to that of classical web proxies and load balancers, which are extensively used in production systems.

5.2.1.2 Prototype Implementation for Java

We have implemented a prototype of Itzal for Java in a tool named Itzal, which is dedicated to message-driven applications based on HTTP. For sake of open-science, Itzal is publicly available on https://github.com/Spirals-Team/itzal-runtime-repair.

Itzal has a default Request-oracle based on unhandled exceptions. Any unhandled exception happening during the processing of a request is considered as a failure. Also, for the case studies in the domain of web applications that will be presented later in

[6]this is supported by all major databases, whether relational or NoSQL

Section 5.2.2, we have also implemented a Request-oracle based on HTTP return codes. According to the specification of the HTTP status codes, the HTTP status code that begins with the digit "5" indicates that the server is aware that it has encountered an error. Failure detection is achieved by checking whether the HTTP status code begins with the digit "5". If it is the case, the request is considered as failing. Otherwise, it is considered as successful.

Implementation of the Patch Generation Service In our implementation, the Patch Generation Service uses two different patch enumeration techniques. First, our prototype system uses the NPEFix model [13] which addresses null dereferences in Java. Second, our prototype system also implements the exception-stop model [102], which prevents the failure from happening by adding try-catch blocks at the method level.

Implementation of the Regression Assessment Service The Regression Assessment Service first receives and stores a list of patches from the Patch Generation Service. Then, it will apply the requests that it receives from the Shadower on each patch. Finally, the observable behavior of the patched application is compared with that of the Unmodified Application, and the results of the comparison are stored to provide statistics to the developers.

In Itzal, the HTTP body of the response of the Unmodified Application is compared against that of the patched application (e.g., the HTML body text for web apps). The comparison discards transient information (e.g., IP addresses and dates). It can further be configured with domain-specific heuristics. If the outputs do not match, the patch is discarded and is permanently marked as a "regressive patch".

Implementation of the Shadower The Shadower is implemented with the Jetty Proxy Library.[7] The major implementation challenge is to maintain a list of session identifiers (e.g., cookies) for each shadowed service. To achieve this, when a session-enabled request arrives with the session ID of the end user's browser, the Shadower translates on-the-fly the session ID to each of the shadowed services (and vice-versa for the response).

[7] Jetty Proxy Servlet http://www.eclipse.org/jetty/documentation/9.4.x/proxy-servlet.html

Implementation of the Sanboxing Sandboxing is achieved using the Docker container system, a major software containerization platform which provides powerful sandboxing capability (including both disk and network sandboxing) [136]. The Patch Generation Service and the Regression Assessment Service are encapsulated in their respective docker images, with disk and network sandboxing enabled, so that it is impossible for them to impact the production state.

Communication with the Developer Now we discuss the communication of the patches with the developers (arrow d, e in Figure 5.5). In the current prototype, we use a web dashboard where the developers follow in real time the failures, the generated patches and the progression of the regression testing of the patches on user traffic. We also imagine an approach integrated into the versioning system (Git/GitHub) where patches are communicated to the developers with automated pull requests.

If patches exist for multiple failures, we use the failure count $failureCount_r$ from Algorithm 5.3 to order the patches. The idea is that the developer would prefer to spend time firstly to the most frequent failures. It also happens that, for the same failure kind, multiple patches successfully pass the regression testing done over the user traffic. Consequently, we also sort the patches in order to first propose the most useful ones to the developers. We prioritize the patches according to the regression success count ($regressionSuccessCount_p$ from Algorithm 5.4). The idea is that the more a patch has been executed by the Regression Assessment Service, the more confidence we have in it.

5.2.2 Evaluation

In this section, we demonstrate the feasibility of our novel and disrupting scheme of patch synthesis in production. Our blueprint architecture addresses many different aspects: patch generation, regression detection based on shadow traffic, and shadower.

We devise a research protocol that aims at: 1) studying each aspect one by one in isolation, and 2) studying the Itzal from an end-to-end perspective. Figure 5.6 depicts the evaluation approach.

RQ1. [Live Patch Synthesis Feasibility] To what extent is it possible to generate patches in production, directly from failing requests triggered by user traffic? The research question aims at evaluating the Patch Generation Service to verify that it is possible

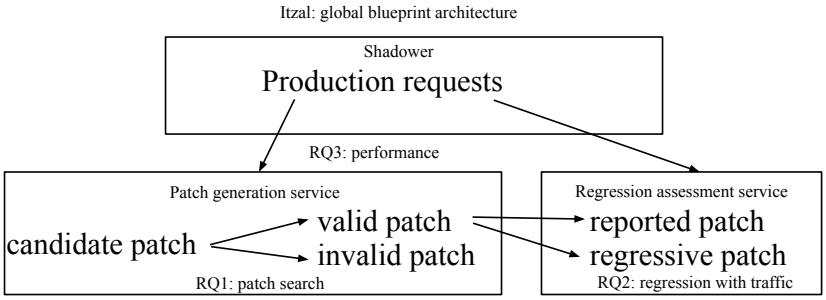

Figure 5.6 Our research questions target each component in isolation as well as the global end-to-end approach.

to generate patches directly from failures in a production environment, where failing requests replace failing test cases.

RQ2. [User Traffic Effectiveness for Regression] What is the effectiveness of using user-traffic to perform live regression testing? The research question aims to evaluate the Regression Assessment Service. We want to see whether one can use user traffic to discard incorrect patches. We study the effectiveness of four execution comparison oracles which are used to compare the behaviors of the patched application against that of the original application.

RQ3. [Performance] What is the performance overhead of Itzal? The research question aims to evaluate the performance overhead of Itzal. We measure the performance overhead of our blueprint architecture on a production system, and compute the required time needed by the Patch Generation Service to generate the patches.

RQ4. [End-to-end Effectiveness] How does Itzal work in a production-like setting? While RQ1, RQ2, and RQ3 specifically concentrate on Patch Generation Service, Regression Assessment Service, and Shadower, the research question aims to evaluate Itzal from an end-to-end perspective by considering two real bugs that are reproducible in a live server environment with emulated traffic.

Benchmarks: Note that the subjects required to answer each of the four research questions are not identical. They all share the characteristics of being really hard to obtain. For instance, it is really difficult to collect and reproduce production failures in the laboratory. Consequently, we build one specific evaluation benchmark per research question. However, we have managed to have one common case across all research questions: Mayocat-231 is used as a real bug in RQ1, as a regression subject for RQ2, in the overhead measurement of RQ3, and in the end-to-end evaluation of RQ4.

5.2.2.1 RQ1. Live Patch Generation Feasibility

Benchmark In order to evaluate whether patch generation can be made directly on production failures, we need to identify real reproducible failures. To collect as many production bugs as possible, we build a benchmark based on the failures used in five different papers from the literature: [63], [5], [92], [13] and [137].

Our inclusion criteria are as follows. First, we select the exception bugs. An exception bug is an unhandled exception in production which makes a request crash. Second, we only keep the bugs for which we are able to replay user traffic or setup that triggers the bug. Third, we discard the failures that happen during initialization or shutdown of an application.

By applying the inclusion criteria, we eventually come up with 34 real production failures from 14 different applications. The benchmark contains 33 null pointer exceptions and one invalid argument exception. For sake of open-science, this benchmark is publicly available on GitHub [138].

In Table 5.6, the first five columns present the dataset. The first column contains a simple bug identity, the second column contains the origin of the bug, the third column contains the type of production oracle considered, the fourth column contains the type of the failure (NPE for Null Pointer Exception, IAE for IllegalArgumentException), and the fifth column contains the number of lines of Java code of the buggy application under consideration, which is computed with the CLOC tool.

Experimental Protocol for RQ1 To evaluate the patch generation of Itzal, we set up the following experimental protocol. The main idea of this experimental protocol is to execute an HTTP request that triggers a failure. Based on this failure the Patch Generation Service search for patches. The main novelty of this setup compared to test suite based program repair is the following: while previous experiments assume a manually written failing test case, Itzal only assumes a failing request. This greatly widens the applicability of the approach.

For the bug of our benchmark that uses HTTP status as failure oracle, we simulate a server that runs the buggy version. This server waits for requests, as a production server would do. Then, we send a request that triggers the failure. We check whether the failure is well detected by the Request-oracle, the HTTP status in this case. Then, we put the failure-triggering request in an infinite loop to simulate arriving user requests that trigger

Table 5.5 The benchmark used in our experiments.

Bug	Origin	Request Oracle	Bug Type	LOC
BroadleafCommerce 1282	[137]	HTTP status	NPE	161 428
Collection 360	[13]	Exception	NPE	21 650
DataflowJavaSDK c06125d	[5]	Exception	NPE	50 655
Felix 4960	[13]	Exception	NPE	33 057
Javapoet 70b38e5	[5]	Exception	NPE	3 884
Jetty 335500	[92]	HTTP status	NPE	153 789
Jongo f46f658	[5]	Exception	NPE	7 384
Lang 20	[63]	Exception	NPE	49 637
Lang 304	[13]	Exception	NPE	17 277
Lang 33	[63]	Exception	NPE	45 444
Lang 39	[63]	Exception	NPE	45 143
Lang 587	[13]	Exception	NPE	17 319
Lang 703	[13]	Exception	NPE	19 047
Math 1115	[13]	Exception	NPE	90 782
Math 1117	[13]	Exception	NPE	90 794
Math 290	[13]	Exception	NPE	38 728
Math 305	[13]	Exception	NPE	38 893
Math 369	[13]	Exception	NPE	41 082
Math 4	[63]	Exception	NPE	164 667
Math 70	[63]	Exception	NPE	83 720
Math 79	[63]	Exception	NPE	89 611
Math 988A	[13]	Exception	NPE	82 442
Math 988B	[13]	Exception	NPE	82 443
Mayocat 231	[137]	HTTP status	NPE	31231
PDFBox 2812	[13]	Exception	NPE	67 294
PDFBox 2965	[13]	Exception	NPE	64 375
PDFBox 2995	[13]	Exception	NPE	64 821
Sling 4982	[13]	Exception	NPE	583
Tomcat 43758	[92]	HTTP status	NPE	156 480
Tomcat 54703	[92]	HTTP status	NPE	186 301
Tomcat 55454	[92]	HTTP status	NPE	193 648
Tomcat 56010	[92]	HTTP status	IAE	195 130
Tomcat 58232	[92]	HTTP status	NPE	224 194
Webmagic ff2f588	[5]	Exception	NPE	9 239
34 bugs from 14 software applications				2 622 172

Table 5.6 The feasibility of using two patch generation models for production failures. Many patches from the patch models' search space are marked as invalid because they fail to make the runtime exception disappear. The main goal is to have non-zero values in column "# Valid".

Bug	Patch Models			
	NPEFix		Exception-Stopper	
	# Valid	# Invalid	# Valid	# Invalid
BroadleafCommerce 1282	5	8	0	0
Collection 360	16	35	64	44
DataflowJavaSDK c06125d	2	1	0	0
Felix 4960	4	6	0	1
Javapoet 70b38e5	0	133	12	87
Jetty 335500	2	2	2	2
Jongo f46f658	0	1	2	8
Lang 20	78	634	0	15
Lang 304	65	12	32	276
Lang 33	1	27	0	3
Lang 39	4	7	0	8
Lang 587	28	0	3	0
Lang 703	7	12	0	15
Math 1115	5	6	4	1
Math 1117	22 132	29 711	0	0
Math 290	4	10	4	3
Math 305	3	1	9	1
Math 369	23	3	22	2
Math 4	415	95	2	17
Math 70	1	25	0	24
Math 79	0	4	0	10
Math 988A	168	37	8	11
Math 988B	17	15	1	12
Mayocat 231	102	182	18	19
PDFBox 2812	4	21	2	9
PDFBox 2965	3	1	1	0
PDFBox 2995	1	4	1	1
Sling 4982	11	9	6	4
Tomcat 43758	1	9	1	0
Tomcat 54703	10	0	2	1
Tomcat 55454	1	0	1	0
Tomcat 56010	0	0	0	7
Tomcat 58232	3	0	0	0
Webmagic ff2f588	2	49	0	10
34 bugs from 14 software applications	23 118	31 060	198	592

the failure, making the same failure happening again and again, as in production. Each time the failure happens, it triggers a patch search by the Patch Generation Service. Hence, the Patch Generation Service enumerates all candidate patches and identify those that make the failure disappear, i.e., that pass the Request-oracle.

For the other bugs due to unhandled exceptions, we encapsulate a small execution scenario that triggers the unhandled exception into an HTTP request that could be run again and again. This small execution scenario is also put in an infinite loop, as what happens in production with user-generated requests.

Finally, we use the two patch models described in Section 5.2.1.2 to generate patches for the bugs with. We count the number of invalid and valid patches for each failure and for each patch model.

Results We now present the results for this research question. To answer this question, we consider columns *# NPEFix Valid/Invalid* and *# Exception-Stopper Valid/Invalid* of Table 5.6, which show the number of valid and invalid patches generated by our two patch models respectively. Valid means the initial failure does not happen anymore, and no other exceptions happen. Invalid means that the initial failure still happens or other failures happen. The main goal is to have non-zero values in column "# Valid", this shows the feasibility of our vision.

For example, the first row of Table 5.6 presents the result for bug BroadleafCommerce 1282. This bug is caused by a null dereference happening upon a user request. We assert the presence of the bug in the application by using an HTTP-based production oracle: HTTP status. The first repair model, NPEFix, generates 13 candidate patches, including 5 valid patches and 8 invalid patches. The second repair model, Exception-Stop, does not generate any patch for this specific bug.

Overall, we can see from Table 5.6 that it is possible to generate patches for real-life production failures. For all the 34 failures of our benchmark, at least one patch can be generated by the two patch models used by Itzal. The Request-oracle is capable of discarding many invalid patches that are in the search space of the considered synthesis techniques.

The number of generated patches varies significantly between projects and failures, the number of candidate patches ranges from 2 (for Tomcat 55454) to 51843 (for Math 1117) and the number of valid patches varies between 0 (for several failures) and 22132 (for Math 1117). This difference in the number of generated patches from the two patch

models emerges as NPEFix is able to generate more patches than Exception Stopper in general. The underlying reason is that the search space of the Exception Stopper patch model is smaller than that of NPEFix. The search space of Exception Stopper is defined by the number of method calls in the stack and the number of variable/value pairs that are available for returning from the current method. Instead, the search space of NPEFix is defined by 9 repair strategies that contain several variants (different values for the placeholder) depending on the context of execution.

Interestingly, we can see from the table that there are a lot of valid patches for both considered patch models. This is a challenge because one would obviously not report to the developer so many patches. However, this issue is handled later in Itzal because 1) the Regression Assessment Service further removes patches and 2) the patches are displayed to the developers by the order of potential value, as discussed in Section 5.2.1.2.

Meanwhile, we can also see from the table that NPEFix also has proportionally more valid patches than Exception Stopper. This can be explained by two facts. On the one hand, Exception Stopper is a generic repair technique, which works at the coarse-grain level. But NPEfix works at the point-of-failure (statement level), and thus it generates patches that are more likely to be invalid. On the other hand, our benchmark contains mostly null dereferences. NPEFix is thus favored as its strategies are specifically designed to handle such failures. Note NPEFix is unable to handle failures that are not null dereference by design (this is what happens for the failure in Tomcat 56010).

To sum up, for all the 34 failures of our benchmark, we show that it is possible to generate patches using one or both patch models implemented in Itzal. This is a large proof-of-concept that it is possible to generate patches directly from production failures.

Answer to RQ1. This novel experiment on 34 production failures shows that one can replay a failing request to explore the search space of a patch model. Thus, it is possible to generate patches directly in production based on user traffic. Our experimentation with two different patch models shows that Itzal is oblivious to the actual patch generation technique.

5.2.2.2 RQ2. User Traffic Effectiveness for Regression.

We have shown that it is possible to generate patches directly from user traffic. We are now interested in seeing if it is possible to use user traffic to discard regressive patches.

Table 5.7 The Effectiveness of four Execution Comparison Oracles to Detect Regressions based on User Traffic. A green plain circle means that the oracle is effective at detecting the regression.

Projects	Patch Location	Oracles				Is Valid Patch
		HTTP status	HTTP content	# Method	# Block	
Broadleaf	CategoryImpl:835 1	0% ⊗	17% ●	36% ●	40% ●	No
	CategoryImpl:835 2	0% ⊗	17% ●	36% ●	40% ●	No
	CategoryImpl:835 3	0% ⊗	20% ●	36% ●	41% ●	No
	CategoryImpl:835 4	0% ⊗	17% ●	36% ●	40% ●	No
	CategoryImpl:835 5	0% ⊗	20% ●	36% ●	41% ●	No
	CategoryImpl:835 6	0% ⊗	20% ●	36% ●	41% ●	No
	CategoryImpl:835 7	45% ●	48% ●	42% ●	47% ●	No
	OrderItemImpl:418 1 ✓	0% ⊗	1% ●	0% ⊗	0% ⊗	Yes
	OrderItemImpl:418 2	0% ⊗	1% ●	0% ⊗	0% ⊗	No
	OrderItemImpl:418 3 ✓	0% ⊗	1% ●	0% ⊗	0% ⊗	Yes
	RelatedProductsServiceImpl:208 1 ✓	0% ⊗	1% ●	0% ⊗	0% ⊗	Yes
	RelatedProductsServiceImpl:208 2	0% ⊗	7% ●	34% ●	38% ●	No
	RelatedProductsServiceImpl:208 3	0% ⊗	15% ●	3% ●	4% ●	No
	SolrHelperServiceImpl:531 1	0% ⊗	13% ●	0% ⊗	0% ⊗	No
	SolrHelperServiceImpl:531 2	35% ●	37% ●	11% ●	9% ●	No
	SolrHelperServiceImpl:531 3	35% ●	37% ●	11% ●	9% ●	No
	SolrHelperServiceImpl:531 4	35% ●	37% ●	11% ●	9% ●	No
	SolrHelperServiceImpl:531 5	35% ●	37% ●	11% ●	9% ●	No
Mayocat	AbstractScopeCookieContainerFilter:202 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	AbstractScopeCookieContainerFilter:202 2 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	AbstractScopeCookieContainerFilter:202 3	0% ⊗	21% ●	6% ●	6% ●	No
	AbstractScopeCookieContainerFilter:202 4	0% ⊗	21% ●	6% ●	6% ●	No
	AbstractScopeCookieContainerFilter:202 5	0% ⊗	21% ●	6% ●	6% ●	No
	AbstractScopeCookieContainerFilter:256 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	AbstractScopeCookieContainerFilter:256 2	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	AbstractScopeCookieContainerFilter:256 3	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	AbstractScopeCookieContainerFilter:256 4	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	DateAsTimestampArgumentFactory:30 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	DateAsTimestampArgumentFactory:30 2	0% ⊗	0% ⊗	1% ●	1% ●	No
	DateAsTimestampArgumentFactory:30 3 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	DateAsTimestampArgumentFactory:30 4	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	DateAsTimestampArgumentFactory:30 5	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	DefaultCartLoader:88 1	82% ●	82% ●	18% ●	16% ●	No
	DefaultCartLoader:88 2 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	DefaultCartManager:198 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	DefaultCartManager:198 2	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	FlatStrategyPriceCalculator:38 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	FlatStrategyPriceCalculator:38 2	0% ⊗	5% ●	0% ⊗	0% ⊗	No
	FlatStrategyPriceCalculator:38 3 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	FlatStrategyPriceCalculator:38 4	0% ⊗	5% ●	0% ⊗	0% ⊗	No
	FlatStrategyPriceCalculator:38 5	20% ●	20% ●	1% ●	1% ●	No
	FlatStrategyPriceCalculator:38 6 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	FlatStrategyPriceCalculator:38 7	20% ●	20% ●	1% ●	1% ●	No
	MapAsJsonArgumentFactory:30 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	MapAsJsonArgumentFactory:30 2	0% ⊗	0% ⊗	2% ●	2% ●	No
	MapAsJsonArgumentFactory:30 3	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	MapAsJsonArgumentFactory:30 4 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	MapAsJsonArgumentFactory:30 5 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	PostgresUUIDArrayArgumentFactory:30 1 ✓	0% ⊗	0% ⊗	1% ●	2% ●	Yes
	PostgresUUIDArrayArgumentFactory:30 2	0% ⊗	0% ⊗	0% ⊗	1% ●	No
	PostgresUUIDArrayArgumentFactory:30 3 ✓	0% ⊗	0% ⊗	0% ⊗	1% ●	Yes
	PostgresUUIDArrayArgumentFactory:30 4 ✓	0% ⊗	0% ⊗	0% ⊗	1% ●	Yes
	ProductMapper:44 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	ProductMapper:44 2	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	ProductMapper:44 3	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	ProductMapper:44 4	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	ProductMapper:44 5	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
Shopizer	CategoryFacadeImpl:55 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	CategoryFacadeImpl:55 2	0% ⊗	67% ●	6% ●	5% ●	No
	CategoryFacadeImpl:55 3	0% ⊗	67% ●	18% ●	18% ●	No
	CategoryFacadeImpl:55 4	0% ⊗	67% ●	15% ●	14% ●	No
	CategoryFacadeImpl:55 5	0% ⊗	67% ●	18% ●	18% ●	No
	ReadableCategoryPopulator:51 1	0% ⊗	0% ⊗	1% ●	1% ●	No
	ReadableCategoryPopulator:51 2	0% ⊗	67% ●	4% ●	3% ●	No
	ReadableCategoryPopulator:51 3	12% ●	67% ●	10% ●	7% ●	No
	ReadableCategoryPopulator:51 4	0% ⊗	67% ●	6% ●	4% ●	No
	ReadableCategoryPopulator:51 5	0% ⊗	67% ●	6% ●	5% ●	No
	ReadableProductPopulator:94 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	ReadableProductPopulator:94 2	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	ReadableProductPopulator:94 3	26% ●	37% ●	9% ●	6% ●	No
	ReadableProductPopulator:94 4	26% ●	37% ●	11% ●	7% ●	No
	ReadableProductPopulator:94 5	26% ●	37% ●	11% ●	7% ●	No
	ShoppingCategoryController:253 1 ✓	0% ⊗	0% ⊗	0% ⊗	0% ⊗	Yes
	ShoppingCategoryController:253 2	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	ShoppingCategoryController:253 3	0% ⊗	0% ⊗	0% ⊗	0% ⊗	No
	ShoppingCategoryController:253 4	0% ⊗	12% ●	2% ●	1% ●	No
	ShoppingCategoryController:253 5	12% ●	12% ●	2% ●	1% ●	No
	ShoppingCategoryController:253 6	12% ●	12% ●	2% ●	1% ●	No
	ShoppingCategoryController:253 7	12% ●	12% ●	2% ●	1% ●	No
	ShoppingCategoryController:253 8	12% ●	12% ●	2% ●	1% ●	No
	80 patches from 17 locations	16	42	39	42	23 valid patches

Benchmark The benchmark of RQ1 has a single request, i.e., the failure-triggering one. For this second research question, we need several requests for the same application, i.e., a workload. We search for HTTP applications on the GitHub software repository with a focus on e-commerce applications as e-commerce applications are easy to understand and consequently, we can create a meaningful workload.

We identify three e-commerce applications that meet our criteria: Mayocat[8], BroadLeaf Commerce[9], and Shopizer[10]. Mayocat is composed of 31 231 lines of Java code, done over 1 670 commits, and in development since 2012. BroadleafCommerce is bigger, it is composed of 154 309 lines of code, done over 9 779 commits, and in development since 2008. Shopizer is composed of 61 555 lines of Java code, done over 154 commits, and in development since 2015. Similarly, for the sake of open science, this benchmark is made publicly available on GitHub [138].

User traffic: For each of these three e-commerce applications, we create a user traffic by identifying a set of requests that execute the major user features, such as adding an item to the cart. Then we automatically create 25 different user sessions that contain between 3 and 7 requests, selected randomly from our set of requests. For sake of reproducibility, we always use the same random seed. Consequently, we generate a user traffic of 124 requests for each e-commerce application. By keeping the number of requests below 200, the experimental time remains manageable.

Failures: Since we aim at studying the Regression Assessment Service which detects regressions introduced by patches, we need such patches. To achieve this, we first seed faults into the programs and then consider a sample of patches generated by the patch model under consideration for the seeded faults. We seed null dereference faults by removing the "then" block of a randomly sampled not-null check that has been executed. For example, if an executed not-null check is "if (x==null) then A else B", we rewrite it as "B". In other words, we remove the error-handling code which deals with null values. We further check whether the seeded faults really trigger failures. Eventually, we have 16 seeded faults that trigger failures under our emulated user traffic.

Patches: For each seeded fault, we select a random sample of patches that are in the search space of NPEFix, one of the patch models implemented in the prototype implementation of Itzal. We obtain a benchmark of 80 candidate patches to be considered for regression. The first two columns of Table 5.7 present this benchmark.

[8] http://www.mayocat.org/
[9] http://www.broadleafcommerce.com/
[10] http://www.shopizer.com/

Experimental Protocol for RQ2 To evaluate the Regression Assessment Service, we first execute the user traffic as described in Section 5.2.2.2 on each considered patch.

Then, for each request of the user traffic, we compare the execution of the patched application against that of the original application. In this context, it means defining a point of observation, collecting some values at this point, and comparing the values collected on the original application with that collected on the patched application. This enables us to observe differences, called "divergence" in the rest of this contribution, following the terminology introduced in [139]. If there exists a divergence for a normal successful request, the patch is considered as a regression.

In this experiment, we consider four different execution comparison oracles to capture divergences. The first is the HTTP status of the response, the second is the HTTP content of the response, the third is the set of covered methods, and the final one is the set of covered blocks. For HTTP status and HTTP content, we collect the percentage of requests for which we observe differences in HTTP status and content respectively. For method and block coverage, we first collect the coverage divergence for each request and then compute the average value over all requests. Finally, we compare the oracle results against a manual analysis of the generated patches.

Results Table 5.7 contains the data obtained with the experimental protocol described in Section 5.2.2.2, investigating whether the execution comparison oracles considered are able to detect divergences. The first column of Table 5.7 contains the name of the project under consideration. The second column contains the patch id which is composed of the class name, the line number, and a sequential id. The four next columns of Table 5.7 provide the measured divergence between the original program and the seeded programs using the 4 execution comparison oracles presented in Section 5.2.2.2. A green plain circle ● means that the considered comparison oracle is able to detect a regression on at least one request, which is desirable in the context of patch generation in production. A crossed circle ⊗ means that the execution comparison oracle fails to detect a divergence. An ideal oracle would detect all divergences, but this would require to compare the whole execution state which is impossible in practice. The last column, Is valid Patch, indicates if the generate patch is semantically correct according to our manual analysis.

For example, the first row of Table 5.7 describes a patch for Broadleaf at line 835 of file CategoryImpl. For this patch, the HTTP status does not detect a single divergence, the HTTP content detects a divergence for 17% of the requests (i.e., 17% of the requests have different contents compared to the original program), and the average divergences

of method and block coverage across all requests are 36% and 40% respectively. When a line contains 0% ⊗ for all four oracles, it means that either the patch is correct (hence has no regressions) or that the synthetic workload we use is not rich enough to highlight the regressions.

The HTTP status execution comparison oracle is easy to obtain but is at a relatively coarse level. Indeed, we can see from Table 5.7 that the HTTP status oracle discards only 16 patches of the whole 80 patches. Since we do not have a reliable correctness oracle, it is not meaningful to compute information retrieval metrics such as precision and recall. The HTTP status execution comparison oracle has two main advantages. On the one hand, it has virtually no overhead. On the other hand, it is directly generalizable to any HTTP based applications.

The HTTP content execution comparison oracle detects regressions for 42 of the whole 80 patches, which is better than the HTTP status execution comparison oracle. However, it has a drawback: it requires to define some transformations on the output in order to remove transient information. In Itzal, the response is cleaned by removing certain transient information, e.g., date, a session key or dynamic CSS classes. It is not always possible to identify all transient information. For instance, for the patches at line 418 of file OrderItemImpl for project Broadleaf, the HTTP content execution comparison oracle is considered as regressive because random inconsistencies happen in the HTML response (one letter disappears at different locations). In the context of patch generation in production, this means that some patches would be discarded because of transient information but not because of an actual regression.

The method coverage and block coverage execution comparison oracles both detect more regressions and have almost the same behavior. Compared to method coverage execution comparison oracle, the block coverage execution comparison oracle detects regressions for 3 more patches located at file PostgresUUIDArrayArgumentFactory for project Mayocat. While these two execution comparison oracles are effective at detecting behavior changes, we have observed an issue: parallel execution can introduce some randomness and consequently some variance in the observed dynamic coverage for a given request. This can possibly be a reason for having correct patches that are discarded by at least one oracle.

We consider the HTTP content as the best execution comparison oracle for regression detection based on user traffic. The reasons are: 1) it is quite effective at detecting behavioral changes; 2) its sensitivity can be overcome with careful design (on the contrary,

it is virtually impossible to overcome the non-determinism of observed coverage due to concurrency).

Now let us discuss the oracle results against the actual correctness as found by manual analysis. Our manual analysis of the generated patches reveals that 16 patches are incorrect. However, these 16 incorrect patches have not been detected by any of the four oracles. In other words, the rows for these 16 patches are indicated as invalid but with ⊗ for all the 4 oracles in Table 5.7, e.g., AbstractScopeCookieContainerFilter:256. The reason for this phenomenon is that our simulated HTTP workload is not able to produce inputs that trigger the invalid behaviors of the incorrect patches. There are also 6 patches for which the opposite phenomenon occurs: they are correct but they are discarded due to randomness and multithreading as discussed above.

Answer to RQ2. It is possible to employ user traffic to validate patches. To substantiate this claim, our novel experimental methodology compares different execution comparison oracles that are all available in production. In the context of HTTP based applications, we observe that the HTTP status oracle, HTTP content oracle, method coverage oracle and block coverage oracle successfully discard 16, 52, 39 and 42 out of 80 patches respectively. This result shows that 1) the Itzal novel scheme is generic enough to accommodate different execution comparison oracles; 2) using an HTTP content based execution comparison oracle represents a good trade-off to perform live regression testing in message-driven applications.

5.2.2.3 RQ3. Itzal Performance

We now consider the performance of Itzal. We will focus first on the impact of the Shadower on the performance of the application, and then evaluate how much time Itzal requires to generate patches.

Shadower Overhead As we have previously discussed in Section 11, the performance of the application is only impacted by the Shadower. Since the other two services (Patch Generation Service and Regression Assessment Service) are executed asynchronously (no overhead on the response time), the Shadower is the only one that may have a user-visible impact.

In order to evaluate the performance impact of Shadower on the application and further on the clients, we create a workload of 100 000 requests. We compare the performance of those requests with the Shadower and without the Shadower. First, these 100 000

requests are launched sequentially on the Mayocat application without the Shadower. Second, we execute the same 100 000 requests but this time with the Itzal Shadower. We collect the average response time for the two infrastructures.

We observe that it takes on average 104ms to make a request directly to Mayocat. With Itzal Shadower, it takes on average 114ms to make a request through the Shadower. This represents a slowdown of 10ms per request or an overhead of 10.44% on average. The reason for the slowdown is that the Shadower costs some time to copy the request to Patch Generation Service and Regression Assessment Service, redirect the original request to the Mayocat application, and finally copy the response of Mayocat to the Regression Assessment Service.

Patch Generation Service Performance The role of the Patch Generation Service is to generate the patches for the failing request. We now study how much time the Patch Generation Service needs to generate these patches.

We apply Itzal on the bug Mayocat-231 and we execute the request that produces the failure. We measure how much time the Patch Generation Service takes to exhaustively generate the patches with the NPEFix repair model for this bug.

The result shows that the Patch Generation Service takes 4min and 33sec to generate the 286 candidate patches, which means that the Patch Generation Service takes on average 953ms to generate one patch. Recall that this has no impact on the user because the Patch Generation Service is called in an asynchronous manner by the Shadower (see Section 11). In other words, the end-user does not have to wait for 5 minutes in front of the browser.

Answer to RQ3. By design, the only component of Itzal that has an overhead in production is the Shadower (all other components being called asynchronously, with no blocking callbacks). Our experimental evaluation of the Shadower's overhead shows that it adds on average a 10ms latency per client request, which is negligible from a user experience perspective. When a failure is detected, according to our benchmark, Itzal is able to generate one per patch per second, and the time to explore the search space is linear in the number of patches.

```
   @@ FlatStrategyPriceCalculator.java
   @@ -35,7 +35,8 @@
35          return BigDecimal.ZERO;
36      }
37 -    price = price.add(carrier.getPerItem().multiply(BigDecimal.valueOf(numberOfItems)));
38 +    BigDecimal perItem = carrier.getPerItem() != null ? carrier.getPerItem() : BigDecimal.ZERO;
39 +    price = price.add(perItem.multiply(BigDecimal.valueOf(numberOfItems)));
40      return price;
41  }
```

Listing 5.2 The human patch for bug Mayocat 231.

5.2.2.4 RQ4. End-to-end Effectiveness of Itzal

While research questions RQ1, RQ2, and RQ3 concentrate on evaluating the Patch Generation Service, Regression Assessment Service, and Shadower, the fourth research question aims to evaluate Itzal from an end-to-end perspective by considering two real bugs that are reproducible in a live server environment with emulated traffic. We do the end-to-end evaluation on two real bugs from open-source e-commerce applications– Mayocat-231 and BroadleafCommerce-1282. Mayocat-231 is our working case and has already been used in all the above three research questions, and BroadleafCommerce-1282 has been already used in RQ1.

Experimental Protocol for RQ4 To evaluate the end-to-end effectiveness of Itzal, we apply Itzal on the two applications. The infrastructure consists of four different docker images and an instance of Shadower that duplicates the requests and the responses to the different services. The first docker image contains the Unmodified Application with the correct state to reproduce the bug. The second and the third docker images are for the Patch Generation Service (one per patch model NPEFix or Exception-stopper). The last docker image contains the Regression Assessment Service, the infrastructure to identify regressions due to the generated patches.

Once the infrastructure is up and running, we launch the failing HTTP request that triggers the patch generation by the Patch Generation Service. Then, each generated patch will be evaluated by the Regression Assessment Service against the emulated traffic. Note that the emulated traffic here is the same as the traffic use in the experiment for RQ2,

End-to-end Evaluation on Mayocat-231

```
   @@ FlatStrategyPriceCalculator.java
   @@ -37,2 +37,5 @@
37 +   if (carrier.getPerItem() == null) {
38 +      return null;
39 +   }
40     price = price.add(carrier.getPerItem().multiply(BigDecimal.valueOf(numberOfItems)));
```

Listing 5.3 An invalid Itzal patch for bug Mayocat 231.

```
   @@ FlatStrategyPriceCalculator.java
   @@ -37,3 +37,7 @@
37 -   price = price.add(carrier.getPerItem().multiply(BigDecimal.valueOf(numberOfItems)));
38 +   if (carrier.getPerItem() == null) {
39 +      price = price.add( BigDecimal.ZERO.multiply(BigDecimal.valueOf(numberOfItems)));
40 +   } else {
41 +      price = price.add(carrier.getPerItem().multiply(BigDecimal.valueOf(numberOfItems)));
42 +   }
43     return price;
```

Listing 5.4 A patch found by Itzal for bug Mayocat 231.

Description of the bug This bug is an unhandled null pointer exception of the e-commerce application Mayocat (https://github.com/jvelo/mayocat-shop/issues/231). The bug is triggered during the computation of the shipping cost of the current cart. This bug is present only for one specific shipping strategy. When the bug happens, the user is left with a white page. Worse still, the user session becomes completely unusable, which means that the website is completely broken for this particular user. The client is thus unable to further navigate through the product list, buy a product or even click on the "contact the administrator" link to report the issue.

Human patch Listing 5.2 shows the snippet of code written by the human developer to fix the bug. The patch consists of using "BigDecimal.ZERO" when the shipping price per product ("carrier.getPerItem()") is null. It is a classical patch for null dereferences: adding a not-null check, here in the form of a ternary expression.

Patch Generation Service For the failing request, the Patch Generation Service of Itzal generates 284 candidate patches with patch model NPEFix and 37 candidate patches with patch model Exception-Stopper (the complete list of the Itzal patches is available at [138]). Out of the 321 (284 + 37) candidate patches, 201 (182 + 19) fail to make the exception disappear or produce another exception so that they are considered as invalid (as we did in RQ1). For example, let us consider the candidate patch shown in Listing 5.3. This patch is invalid according to the Request-oracle because it produces an HTTP status 500, i.e., it indicates an internal server error. The reason is that when this patch is applied, a null value is returned, which itself produces a new null pointer

exception in the caller method. This new null pointer exception makes the request fail and the server eventually returns an HTTP 500 code. The Request-oracle well detects the HTTP error and the candidate patch is considered as invalid.

Regression Assessment Service Let us now consider the Regression Assessment Service. In our simulation of bug Mayocat-231, the Regression Assessment Service performs regression testing on 80 synthetic requests (as per RQ2). However, it does not reject patches based on this simulated workload. This happens as the production traffic simulator is unable to create an input for this bug that requires regression testing (the bug report only provide us with crashing input). Note this is a limitation of our production traffic generator, not a conceptual limitation of Itzal. We note that designing generators of likely synthetic traffic is an unresearched area yet a very difficult problem.

Comparison against the Human Patch Among the 120 patches synthesized by Itzal which pass all oracles in this setup, none is syntactically equivalent to the patch written by the developer. However, Listing 5.4 shows an example of a semantically equivalent one, which has the same behavior as the human patch. It replaces the null element ("carrier.getPerItem()") by an existing variable "BigDecimal.ZERO"found in the execution context.

Note that, as shown by Long and Rinard [64], it is common to have many equivalently correct yet syntactically different patches in the search space of a patch model.

End-to-end Evaluation on BroadleafCommerce-1282 We now study the case of bug BroadleafCommerce-1282[11], which is still in the domain of e-commerce. We focus on showing an important point that was not highlighted by the first case study: the fact that some aspects of patch optimality, in particular with respect to user experience, are not handled by standard validity oracles.

Description of the bug This bug is a null dereference that happens when the website administrator adds a customer with an email address that already exists in the database (i.e., the email address is already used by another customer). When this failure occurs, the user interface displays a low level debugging stack trace. Contrary to bug Mayocat-231 that completely breaks the website, this bug has a lower severity.

[11] https://github.com/BroadleafCommerce/BroadleafCommerce/issues/1282

```
1 void populateEntityForm(...) {
2     ...
3     String idProperty = adminEntityService.getIdProperty(cmd);
4
5     // null pointer exception here
6     // because entity.findProperty(idProperty) is null when idProperty is not present in "entity
       "
7     ef.setId(entity.findProperty(idProperty).getValue());
8     ...
9 }
```

Listing 5.5 The failure point of bug BroadleafCommerce-1282.

```
1     adminInstance.setUsername(adminInstance.getEmailAddress());
2     if (customerService.readCustomerByUsername(adminInstance.getUsername()) != null) {
3 -       Entity error = new Entity();
4 -       error.addValidationError("username", "nonUniqueUsernameError");
5 -       return error;
6 +       entity.addValidationError("emailAddress", "nonUniqueUsernameError");
7 +       return entity;
8     }
```

Listing 5.6 The human patch for bug BroadleafCommerce-1282 (simplified version).

```
    @@ FormBuilderServiceImpl.java
    @@ -717,2 +717,5 @@
717     String idProperty = adminEntityService.getIdProperty(cmd);
718 +   if (entity.findProperty(idProperty) == null) {
719 +     return;
720 +   }
721     ef.setId(entity.findProperty(idProperty).getValue());
```

Listing 5.7 The Itzal patch for bug BroadleafCommerce-1282.

Listing 5.5 shows the failure point (i.e., where the null pointer exception happens): When idProperty is "emailAddress", "entity.findProperty(idProperty)" returns null as no such property exists in the entity. Consequently, the call to "getValue()" results in a null pointer exception.

Itzal patches Itzal generates 12 different compilable candidate patches with NPEFix patch model, the other patch model did not succeed to generate a patch for this bug (Again, the complete list of Itzal patches is available at [138]). Among the 12 patches, 5 of them avoid the null pointer exception and do not produce any new bad behaviors that are detected by the Request-oracle.

Let us analyze one of them as shown in Listing 5.7. This patch handles the failure by exiting the method when utility method "findProperty" does not find the required property. With this patch, no dirty error message is displayed in the user interface and can be considered as a valid workaround to the problem.

Comparison against the Human Patch When comparing the Itzal patch against the human patch, the surprise is that they are in different methods. The human patch (shown in Listing 5.6) is in method "validateUniqueUsername", and it essentially replaces the error identifier "username" by "emailAddress". Later on, at the failure point, the "emailAddress" property that is looked up is found and no exception is thrown.

From the viewpoint of the Request-oracle (the absence of exceptions in this case study), both patches handle the failure and both are correct. However, the human patch is conceptually better. While the Itzal patch silently skips the action to be done and gives no feedback to the user, the human patch transforms the exception into a clean and explicit warning about duplicate emails. This shows that there are cases where the absence of domain knowledge in the patch model and/or in the oracle results in sub-optimal patches. To overcome this problem, the developer always has the option to improve the patches shown in the Itzal dashboard before merging them in the code base of the application.

Answer to RQ4. This end-to-end experiment shows the feasibility of deploying the novel program repair scheme Itzal on real applications. It also highlights that the main challenge of doing this kind of research in the laboratory is to have good workloads reflecting production traffic.

5.2.3 Conclusion

In this contribution, we have presented Itzal, an approach for synthesizing patches live for production failures. This novel and disrupting scheme for program repair is based on the conjunction of embedding the patch search in production, together with validating the absence of regressions based on the whole, diverse, production usages and values. We have evaluated our novel technique based on 34 failures.

This new line of research in automatic software repair calls for future work. First, there is a need to research how to efficiently synchronize an application and its shadows (the mirror applications fed with the shadow traffic). Second, we envision a feedback loop with developers as follows. When a developer discards or modifies a generated patch, this information should be given back to Itzal, then the Patch Generation Service would automatically refine the patch model, the Regression Assessment Service would automatically synthesize better execution comparison oracles, and finally, the patch prioritization done in the dashboard would be the result of a machine learning approach.

5.3 Summary

In this chapter, we presented two patch generation techniques that are designed for the production environment. The first technique is designed to generate patches for JavaScript client-side code, i.e., to generate patches for websites. The second technique is designed to generate patches for the server-side applications, with focus on the safety of the execution to prevent any side-effect.

These two techniques demonstrated that it is feasible to generate patches in the production environment. It also shows that it is possible to reduce the involvement of the human in the process of patch generation, which could reduce the maintenance cost for companies and improve the general experience of the user in applications.

Chapter 6

Conclusion and Perspectives

This chapter summarizes the contributions of this thesis and discusses perspectives for further research. It is structured as follows. Section 6.1 summarizes the contributions of this thesis. Section 6.2 describes the short-term perspectives of this thesis. Finally, Section 6.3 describes the global perspectives.

6.1 Summary of the Contributions

This thesis presented three contributions that remove the requirement for human intervention during the patch generation. Our general idea is to directly use the production state of the application to generate patches. This design removes the human intervention and consequently speeds up the patch generation.

The first contribution (see Chapter 3) presented three new patch generation techniques that only use a failing execution and its runtime state to generate patches. The first patch generation technique uses a patch-template approach, the templates are inspired by human patches to generated readable and understandable patches. The second technique uses a metaprogramming approach and embeds nine different repair strategies inside the application. It allows to explore the repair search space by compiling only once the application. The last technique is an expression synthesizer that generates patches for buggy condition and missing pre-condition. We evaluated the techniques on a new benchmark of null pointer exceptions collected from open-source applications and on Defects4J [63] a state-of-the-art benchmark of real bugs. The evaluation showed that the

three techniques can generate valid patches by exploiting the running application and its state.

The second contribution (see Chapter 4) examined the search space of automatic patch generation at runtime. We studied its size, the proportion of valid patches it contains and how much time is required to explore it. It is an important study to understand how to explore the repair search space and how to determine which patches are valid automatically. The study consisted in exhaustively exploring the repair search space of NPEFix (see Section 3.1.3). It showed that the search space can contain several thousand patches and that, in some cases, it is required to support multi-point repair.

The last contribution (see Chapter 5) introduced two patch generation techniques for the production environment. The first technique uses an approach to generate patches for JavaScript client-side applications. The second technique targets server-side applications and proposes to use the production traffic to generate and validate the patches. The technique is designed to remove the side-effects of patch generation by executing it in a sandboxed environment. The first technique is evaluated on a new benchmark of JavaScript failures collected by browsing the web randomly. The second technique is evaluated on a benchmark of real bugs from open-source e-commerce applications. The evaluation of the first patch generation technique was able to automatically reduce the number of crashes by 36%. A case study evaluation showed that the repair technique had a positive impact on the users, by fixing missing content or enabling missing features. The evaluation of the second technique showed the practicability of the approach by repairing automatically null pointer exceptions in e-commerce applications and showed that production oracles can detect behavior changes.

6.2 Local Perspectives

This section presents the local perspectives for each problem that we addressed in this thesis. Recall that the three main problems of this thesis were: 1) how to generate patches without a failing test case to remove the dependency on developers; 2) how to automatically validate patched in the production environment to only present interesting patches to the developers; 3) and how to handle the side-effects of the patch generation in the production environment to remove all potential side-effects of patch generation techniques.

6.2.1 Patch Generation without Failing Test Case

This first problem was to create new patch generation techniques that do not rely on failing test case. In this thesis, we developed four patch generation techniques that only use a failing execution and its state to generate patches. Those techniques open the following two research perspectives: 1) handling new types of bug with the metaprogramming approach; 2) supports additional repair strategies for generic JavaScript bugs.

6.2.1.1 Metaprogramming for Other Kinds of Repair

The metaprogramming approach of NPEFix (Section 3.1.3) has been shown to be particularly effective for handling null pointer exceptions. In future work, we will investigate its ability to fix other kinds of bugs. NPEFix can be easily extended to repair other predictable failures such as invalid index access on arrays and lists or to fix illegal arithmetic operations. In a longer term, we will study the possibility to use metaprogramming approaches to generate patches for different types of bugs. More specifically, we plan to include repair strategies that are used in the literature of the automatic program repair and also to extract common patches from human code modifications.

Perspective A: Extend the metaprogramming patch generation technique to other types of bugs.

6.2.1.2 New Repair Model for JavaScript

The second short-term perspective is to improve the repair model for JavaScript applications. The current literature on patch generation focuses only on C and Java applications, but JavaScript is one of the most popular languages. It is even the most popular language, and by far, on GitHub [140]. JavaScript is interesting because it is an interpreted language and it is heavily integrated with other languages such as HTML and CSS. It is also heavily used by people that do not have computer science background which can influence the type of bugs compared to C or Java. These characteristics open new perspectives for patch generation techniques such as creating a new repair model that exploits the characteristic of interpreted languages. Future work could also address the automated fix of bindings between HTML and JavaScript. The repair technique could for instance automatically fix bugs arising when the developer forgets to update an identifier in the JavaScript after having changed it in HTML.

Perspective B: Repair the buggy interactions between HTML and JavaScript.

6.2.2 Automatic Patch Validation in Production

The second problem of this thesis was to validate the generated patches to only present valid patches to developers. We monitored the behavior of the patched application and compared it to the original application to assert the validity of the patches. This work opens the following open perspectives: 1) use the knowledge obtained during the patch validation to generate additional patches; 2) guide the patch generation with developer feedback.

6.2.2.1 Patch Generation Guided by Patch Validity

A perspective of the patch validation would be to use the patch validation during the patch generation. The patch validation is designed in this thesis as a step separated from the patch generation. However, the patch validation could be integrated directly into the patch generation. A simple integration would continue the patch generation once a patch is rejected, instead of restarting the generation from scratch. A more sophisticated technique would extract the inputs that invalidated the generated patch and process them to integrate them as constraints in the patch generation. This approach would help the patch generation to converge faster to a valid patch. This process is entirely automatic and would increase the quality of the patch without the intervention of developers.

Perspective C: Use the production oracle as an additional patch generation constraint.

6.2.2.2 Patch Generation Guided by Patch Developer Feedback

A second perspective for the patch validation would be to use developer feedback during the patch generation. This perspective is similar the previous one, but it uses the developer knowledge instead of the production oracles. During the review of the generated patch, if the developer rejects the generated patch, the patch generation would restart its patch generation where it found the previous adequate patch. This process can be improved if the developer provides a counterexample that invalidates the generated patch. The counterexample can then be encoded as a constraint or as a test case that helps generating better patches. This perspective and the previous one can be combined to converge faster to a correct patch.

Perspective D: Use developer feedback to guide patch generation.

6.2.3 Side-effect of Patch Generation in Production

The final problem of this thesis was to provide the patch generation techniques with access to the production state without introducing side-effects to the production application. To address this problem, we duplicated the production application in a sandboxed environment. The production traffic associated with the invalid behavior was shadowed to the sandboxed environment where the patch generation techniques can exploit it without risking to introduce side-effects. This architecture creates these two new research opportunities: 1) evaluate the patch generation in industrial applications; 2) create patch generation techniques for mobile applications.

6.2.3.1 Evaluation on Industry Applications

In this thesis, we adopted an academic perspective to evaluate our patch generation techniques. To ensure the practicality of our approaches on real systems, we plan to create collaborations with the industry to deploy our techniques in real production systems. The sandboxed environment that we used in Section 5.2 simplifies the deployment since it minimizes the impact on the partner. This collaboration would have two effects, first, it would provide automatic patches and additional knowledge on the software to the industry. Second, it would provide us with feedback on the expectations of developers and with access to a rich and real workload which not easily available to an academical laboratory.

Perspective E: Integrate our patch generations in a real production system.

6.2.3.2 Patch Generation for Mobile Application

The second local perspective that we will investigate is the possibility to use the knowledge and technologies that we create in this thesis to build a patch generation technique for mobile applications like Mahajan et al. did [141]. Indeed, in this thesis, we only focused on Java and JavaScript applications. Mobile applications are now massively used and are more and more important in our day-to-day life. The diversification of operating systems and devices makes it difficult for the developer to think about and handle all cases. The

reproduction of bugs is even harder. In this environment, automatic techniques can provide a valuable help for the developers to identify problems and handle them.

Perspective F: Bring patch generation to mobile applications.

6.3 Global Perspectives

In this thesis, we presented new approaches that integrate the automatic patch generation in the production environment to reduce the dependence on developers. This thesis opens new perspectives in different directions, two of them are detailed next.

6.3.1 Production Oracle for Patch Validity

The current patch generation techniques suffer from patch overfitting. It means that the patch generation technique succeeds in generating a patch that makes the test suite pass, but it is in practice incorrect. In other words, the generated patches overfit the test inputs by making the application work as expected for the input specified in the tests while breaking the behavior of the application for other inputs. A potential solution to address this problem is to verify the behavior of the application with a larger set of inputs. The production state of the application can provide this diversification of inputs.

A production oracle can be used to assess the correct behavior of the application for the production inputs. A production oracle, as defined in Section 10, is an oracle that compares the state of the patched application with its original state. We considered four different production oracles in Section 10: the HTTP status of the two application versions, the HTTP content, the method coverage and the block coverage.

Our initial work has shown that the production oracles are indeed able to detect behavior changes and to provide different granularity level of comparison, i.e., some oracles are more sensitive to changes than others. In light of these initial results, we plan to continue this work by considering new potential production oracles. We hypothesize that several different production oracles must be combined to correctly detect regressions. The ability to compare executions is useful for assessing the validity of generated patches and also for a large range of other applications, from debugging assistants to the detection of incoherent behaviors in a running application.

Perspective G: Monitor behavior changes in applications to detect valid and invalid patches.

6.3.2 Interaction Between the Developers and the Generated Patches

Code review is one of the most important steps in the life cycle of a code change. It is the last verification of a code modification before it is merged in the main repository of an application. It is also at that moment that the contributors of the project interact to understand the modifications and potentially ask for corrections. This step is important to ensure the quality of the application and also to communicate the changes to other members of the project. There is currently a large effort to create techniques that automate the patch generations. However, the research community targets only the patch generation itself. There is currently no work considering the interaction between the patch generation technique and the human members of the project. Even with an automatic approach, developers will still want to continue to ask questions and to ask for code modifications, to understand and improve the generated patch. The practicality of the patch generation techniques would consequently remain limited until the patch generation techniques can interact with developers.

We plan to work on a new concept of interactive bot for code review. The goal is to assist the developers during the review process in order to explain the code change and take into account modification requests. This bot would, for example, be able to do some refactoring in the code change, i.e., rename a variable. It will also be able to provide valuable information that is fastidious to collect by a human. The bot can, for example, provide an insight on the behavior change that the code modification has on the application. This information can help the developers decide if the patch has a positive or negative impact on the application. This can be done, for example, by automatically deploying the code modification in the staging environment. Then, the bot would shadow the production traffic and compare the behavior of the production execution and the staging execution. The comparison could then be presented to the developers.

Perspective H: Create a bot that interacts with the developers during the review of generated patches.

6.4 Final Words

In this thesis, we presented our three-year effort to create a fully automated patch generation technique that has the ambition of reducing the maintenance and improving the user experience. This work opens several interesting research perspectives and new challenges that are summarized in Table 6.1.

Table 6.1 Summary of this thesis' perspectives.

#	Perspectives
A	Extend the metaprogramming patch generation technique to other types of bugs.
B	Repair the buggy interactions between HTML and JavaScript.
C	Use the production oracle as an additional patch generation constraint.
D	Use developer feedback to guide patch generation.
E	Integrate our patch generations in a real production system.
F	Bring patch generation to mobile applications.
G	Monitor behavior changes in applications to detect valid and invalid patches.
H	Create a bot that interacts with the developers during the review of generated patches.

References

[1] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[2] 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, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3.

[3] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *Transactions on Software Engineering (TSE)*, 2016.

[4] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659. IEEE Press, 2017.

[5] 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*, pages 727–739. ACM, 2017.

[6] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A.D. Keromytis. Assure: Automatic software self-healing using rescue points. *ACM Sigplan Notices*, 44(3):37–48, 2009.

[7] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, volume 4, pages 21–21, 2004.

[8] Harold Valdivia Garcia and Emad Shihab. Characterizing and predicting blocking bugs in open source projects. In *Proceedings of the 11th working conference on mining software repositories*, pages 72–81. ACM, 2014.

[9] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering (EMSE)*, 2016.

[10] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Test case generation for program repair: A study of feasibility and effectiveness. *Empirical Software Engineering (EMSE)*, 2017.

[11] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811, 2013.

[12] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[13] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 349–358. IEEE, 2017.

[14] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST) colocated with ICSE*, pages 85–91. ACM, 2016.

[15] Thomas Durieux, Youssef Hamadi, Zhongxing Yu, and Martin Monperrus. Exhaustive exploration of the failure-oblivious computing search space. In *Proceedings of the 11th International Conference on Software Testing, Validation and Verification (ICST)*, pages 139–149. IEEE, 2018.

[16] Thomas Durieux, Youssef Hamadi, and Martin Monperrus. Fully automated HTML and JavaScript rewriting for constructing a self-healing web proxy. *Under submission, arXiv:1803.08725*, 2018.

[17] Thomas Durieux, Youssef Hamadi, and Martin Monperrus. Production-driven patch generation. In *Proceedings of the 37th International Conference on Software Engineering (ICSE), track New Ideas and Emerging Results*, pages 23–26. IEEE, 2017.

[18] Thomas Durieux, Zhongxing Yu, Youssef Hamadi, and Martin Monperrus. Automatic patch synthesis and validation in production. *Under submission, arXiv: 1609.06848*, 2018.

[19] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140. IEEE, 2018.

[20] Nelly Delgado, Ann Q Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on software Engineering*, 30 (12):859–872, 2004.

[21] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. Root cause detection in a service-oriented architecture. In *ACM SIGMETRICS Performance Evaluation Review*, volume 41, pages 93–104. ACM, 2013.

[22] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46. ACM, 2009.

[23] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[24] Jonas Magazinius, Daniel Hedin, and Andrei Sabelfeld. Architectures for inlining security monitors in web applications. In *International Symposium on Engineering Secure Software and Systems*, pages 141–160. Springer, 2014.

[25] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 247–248. ACM, 2012.

[26] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *OSDI*, volume 12, pages 107–120, 2012.

[27] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 17–30. ACM, 2007.

[28] Carla Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *Proceedings of the 2000 workshop on New security paradigms*, pages 101–110. ACM, 2001.

[29] Michael E Locasto, Angelos Stavrou, and Gabriela F Cretu. Life after self-healing: assessing post-repair program behavior. Technical report, Technical report, George Mason University, 2008.

[30] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *ACM SIGARCH Computer Architecture News*, 30(5):45–57, 2002.

[31] Qianqian Wang, Yuriy Brun, and Alessandro Orso. Behavioral execution comparison: Are tests representative of field behavior? In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 321–332. IEEE, 2017.

[32] George Candea and Armando Fox. Crash-only software. In *HotOS*, volume 3, pages 67–72, 2003.

[33] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 2018.

[34] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):17, 2018.

[35] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 162–168, 2008. doi: 10.1109/CEC.2008.4630793.

[36] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of ICST*, pages 65–74, 2010.

[37] Westley Weimer, Zachary P Fry, and Stephen Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.

[38] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[39] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189, 2013. doi: 10.1109/ICSM.2013.29.

[40] Fan Long and Martin C. Rinard. Staged program repair with condition synthesis. In *Proceedings of ESE/FSE*, 2015.

[41] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings. 40th International Conference on Software Engineering, ICSE*, 2018.

[42] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39. ACM, 2014.

[43] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015.

[44] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1.

[45] Shin Hwei Tan and Abhik Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 471–482, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5.

[46] Christian Kern and Javier Esparza. Automatic error correction of Java programs. In *Formal Methods for Industrial Critical Systems*, pages 67–81. 2010.

[47] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages

416–426, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.45.

[48] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *SIGPLAN Not.*, 51(1):298–312, January 2016. ISSN 0362-1340.

[49] Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129. IEEE, 2018.

[50] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449, 2014. doi: 10.1109/TSE.2014.2312918.

[51] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 389–400, 2011.

[52] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, pages 277–287, 2012. doi: 10.1109/ICSE.2012.6227186.

[53] Frolin S Ocariza Jr, Karthik Pattabiraman, and Ali Mesbah. Vejovis: suggesting fixes for javascript faults. In *Proceedings of the 36th International Conference on Software Engineering*, pages 837–847. ACM, 2014.

[54] Divya Gopinath, Sarfraz Khurshid, Diptikalyan Saha, and Satish Chandra. Data-guided repair of selection statements. In *Proceedings of the 36th International Conference on Software Engineering*, pages 243–253. ACM, 2014.

[55] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, Dave Towey, and Zuohua Ding. A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of systems and software*, 126:127–140, 2017.

[56] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 177–187, 2012.

[57] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. Automatically generated patches as debugging aids: a human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 64–74, 2014.

[58] Xianglong Kong, Lingming Zhang, W Eric Wong, and Bixin Li. Experience report: How do techniques, programs, and tests impact automated program repair? In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 194–204. IEEE, 2015.

[59] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[60] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 191–200. IEEE Computer Society Press, 1994.

[61] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, October 2005. ISSN 1382-3256. doi: 10.1007/s10664-005-3861-2.

[62] Matias Martinez and Martin Monperrus. Astor: A program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 441–444, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9.

[63] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, July 23–25 2014.

[64] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 702–713, 2016.

[65] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8.

[66] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Bergamo, Italy*, 2015.

[67] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. 2017.

[68] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. A correlation study between automated program repair and test suite metrics. *Empirical Software Engineering*, pages 1–32, 2017.

[69] Qi Xin and Steven P Reiss. Identifying test suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 226–236. ACM, 2017.

[70] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.

[71] Xinyuan Liu, Muhan Zeng, Yingfei Xiong, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based automatic program repair. *arXiv preprint arXiv:1706.09120*, 2017.

[72] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 433–436, 2007. ISBN 978-1-59593-882-4.

[73] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[74] 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 (TSE), in press*, 2015.

[75] Thomas Durieux and Martin Monperrus. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs*. PhD thesis, Universite Lille 1, 2016.

[76] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182. IEEE Press, 2017.

[77] Codeforces. Codeforces. programming competitions and contests, programming community. https://codeforces.com/, 2018.

[78] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion)*, pages 55–56. ACM, 2017.

[79] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR Data Showcase)*, pages 1–4. To appear, 2018.

[80] Paul E Ammann and John C Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[81] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. *ACM SIGPLAN Notices*, 38(11):78–95, 2003.

[82] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Software self-healing using collaborative application communities. In *Proceedings of NDSS*, 2006.

[83] Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. Avoiding program failures through safe execution perturbations. In *Computer Software and Appli-*

cations, 2008. COMPSAC'08. 32nd Annual IEEE International, pages 152–159. IEEE, 2008.

[84] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 237–246. ACM, 2010.

[85] Antonio Carzaniga, Alessandra Gorla, Nicolo Perino, and Mauro Pezze. Automatic workarounds: Exploiting the intrinsic redundancy of web applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):16, 2015.

[86] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 295–308. USENIX Association, 2008.

[87] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *ACM SIGPLAN Notices*, volume 47, pages 431–450, 2012.

[88] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies – a safe method to survive software failures. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 235–248, 2005.

[89] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, and Nicolo Perino. A self-healing technique for Java applications. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1445–1446. IEEE, 2012.

[90] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 782–791. IEEE Press, 2013.

[91] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[92] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. Automatic runtime recovery via error handler synthesis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 684–695. ACM, 2016.

[93] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 291–302. IEEE, 2005.

[94] Panagiota Papavramidou and Michael Nicolaidis. An iterative diagnosis approach for ecc-based memory repair. In *VLSI Test Symposium (VTS), 2013 IEEE 31st*, pages 1–6. IEEE, 2013.

[95] Stelios Sidiroglou and Angelos D Keromytis. Countering network worms through automatic patch generation. *IEEE Security & Privacy*, 3(6):41–49, 2005.

[96] Jun Yuan and Rob Johnson. Cawdor: compiler assisted worm defense. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 54–63. IEEE, 2012.

[97] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices*, volume 41, pages 158–168. ACM, 2006.

[98] G. Novark, E.D. Berger, and B.G. Zorn. Exterminator: automatically correcting memory errors with high probability. *ACM SIGPLAN Notices*, 42(6):1–11, 2007.

[99] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[100] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[101] Dennis Appelt, Annibale Panichella, and Lionel C. Briand. Automatically repairing web application firewalls based on successful SQL injection attacks. In *28th IEEE International Symposium on Software Reliability Engineering*, pages 339–350, 2017.

[102] Kinga Dobolyi and Westley Weimer. Changing Java's semantics for handling null pointer exceptions. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 47–56. IEEE, 2008.

[103] Stephen W Kent. Dynamic error remediation: A case study with null pointer exceptions. *University of Texas Master's thesis*, 2008.

[104] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 26, 2014.

[105] Manuel Rigger, Daniel Pekarek, and Hanspeter Mössenböck. Context-aware failure-oblivious computing as a means of preventing buffer overflows. Technical Report 1806.09026, Arxiv, 2018.

[106] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 25–33, 2006.

[107] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Conference on Programming Language Design and Implementation,*, 2014.

[108] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. Technical report, 2015.

[109] William B Langdon, Brian Yee Hong Lam, Marc Modat, Justyna Petke, and Mark Harman. Genetic improvement of gpu software. *Genetic Programming and Evolvable Machines*, pages 1–40, 2016.

[110] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[111] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. Codehint: dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663, 2014.

[112] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: an Eclipse plug-in for testing and debugging. In *Proceedings of Automated Software Engineering*, 2012.

[113] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. volume 20, pages 481–494. SAGE Publications, 2006.

[114] Chris Lewis and Jim Whitehead. Runtime repair of software faults using event-driven monitoring. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 275–280. IEEE, 2010.

[115] Uage ranking of JavaScript libraries. https://w3techs.com/technologies/overview/JavaScript_library/all, 2018.

[116] Frolin S Ocariza Jr, Karthik Pattabiraman, and Benjamin Zorn. JavaScript errors in the wild: An empirical study. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 100–109. IEEE, 2011.

[117] Angelos D Keromytis. Characterizing self-healing software systems. In *Fourth International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 22–33, 2007.

[118] Philip Koopman. Elements of the self-healing system problem space. Technical report, Carnegie Mellon University, 2003.

[119] George Candea, Emre Kiciman, Steve Zhang, Pedram Keyani, and Armando Fox. Jagr: An autonomous self-recovering application server. In *Autonomic Computing Workshop. 2003. Proceedings of the*, pages 168–177. IEEE, 2003.

[120] Frolin S Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. A study of causes and consequences of client-side JavaScript bugs. *IEEE Transactions on Software Engineering*, 43(2):128–144, 2017.

[121] Ari Luotonen. *Web proxy servers*. Prentice-Hall, Inc., 1998.

[122] Joseph C Pistriotto and Katrina Montinola. Method and apparatus for configuring a client to redirect requests to a caching proxy server based on a category id with the request, October 24 2000. US Patent 6,138,162.

[123] Tammo Krueger, Christian Gehl, Konrad Rieck, and Pavel Laskov. Tokdoc: A self-healing web application firewall. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1846–1853. ACM, 2010.

[124] Michel K Bowman-Amuah. Load balancer in environment services patterns, June 10 2003. US Patent 6,578,068.

[125] Takipi. We crunched 1 billion Java logged errors - here's what causes 97% of them. http://blog.takipi.com/we-crunched-1-billion-java-logged-errors-heres-what-causes-97-of-them/.

[126] Bikiniproxy repository. https://github.com/Spirals-Team/bikiniproxy/, 2018.

[127] David Oppenheimer, Archana Ganapathi, and David A Patterson. Why do internet services fail, and what can be done about it? In *USENIX symposium on Internet Technologies and Systems*, volume 67. Seattle, WA, 2003.

[128] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-995-1.

[129] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

[130] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 362–371, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3.

[131] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[132] Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015.

[133] The reactive manifesto. http://www.reactivemanifesto.org/. accessed: 8/2016.

[134] Gojko Adzic and Robert Chatley. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 884–889. ACM, 2017.

[135] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI '15, pages 43–54, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6.

[136] Play in a content trust sandbox. https://docs.docker.com/engine/security/trust/trust_sandbox/, accessed: 8/2016.

[137] Thomas Durieux, Youssef Hamadi, and Martin Monperrus. Banditrepair: Speculative exploration of runtime patches. (1603.07631), 2016.

[138] Itzal open source repository. https://github.com/Spirals-Team/itzal-experiments.

[139] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1181–1192. ACM, 2016.

[140] GitHub. The state of the octoverse 2017. https://octoverse.github.com/, accessed: 6/2018.

[141] Sonal Mahajan, Negarsadat Abolhassani, Phil McMinn, and William GJ Halfond. Automated repair of mobile friendly problems in web pages. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018.