Production-Driven Patch Generation and Validation

THOMAS DURIEUX, Inria & University of Lille
ZHONGXING YU, KTH Royal Institute of Technology
YOUSSEF HAMADI, Ecole Polytechnique
MARTIN MONPERRUS, KTH Royal Institute of Technology

We envision a world where the developer would receive each morning in her GitHub dashboard a list of potential patches that fix certain production failures. For this, we propose a novel program repair scheme, with the unique feature of being applicable to production directly. We present the design and implementation of a prototype system for Java, called Itzal, that performs patch generation for uncaught exceptions in production. We have performed two empirical experiments to validate our system: the first one on 34 failures from 14 different software applications, the second one on 16 seeded failures in 3 real open-source e-commerce applications for which we have set up a realistic user traffic. This validates the novel and disruptive idea of using program repair directly in production.

1 INTRODUCTION

In modern distributed systems running on the cloud, software failures happen constantly [Oppenheimer et al. 2003]. 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 [Burton 2016].

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 paper.

This is fundamentally different from traditional program repair (e.g. [Le Goues et al. 2012; Nguyen et al. 2013a]). 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 [Bell et al. 2013; Bettenburg et al. 2008; Jin and Orso 2012]. 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 [Martinez et al. 2016; Qi et al. 2015; Smith et al. 2015].

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 paper, 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

Authors' addresses: Thomas Durieux, Inria & University of Lille, Lille, France; Zhongxing Yu, KTH Royal Institute of Technology, Stockholm, Sweden; Youssef Hamadi, Ecole Polytechnique, Paris, France; Martin Monperrus, KTH Royal Institute of Technology, Stockholm, Sweden.

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.

To show that the Itzal vision is feasible, we have implemented it for Java. To demonstrate that it is generic, we have considered two different patch models: one for null pointer exceptions (#1 exception in production [Burton 2016]), and one for arbitrary uncaught runtime exceptions. We have performed two large scale experiments: the first one on 34 failures from 14 different software applications used in production, the second one on 16 seeded failures in 3 real open-source e-commerce applications for which we have set up a realistic user traffic. Those experiments validate our novel concept of patch synthesis in production.

In summary, our contributions are:

- A novel scheme for program repair in production, it performs patch synthesis and regression testing live based on user traffic. This is a fundamental breakthrough with traditional program repair based on tests or static analysis that may open a new research avenue.
- The design and implementation of a prototype in Java, called Itzal. It is made open-source for fostering follow-up research in that new area.
- The evaluation over 34 production failures from 14 different large scale software applications. It is the first proof-of-concept that our vision of program repair in production is feasible.

This paper is available as working paper on arXiv [Durieux et al. 2016b]. The remainder of this paper is organized as follows. Section 2 presents the current challenges of patch generation. Section 3 presents Itzal. Section 4 presents the evaluation of Itzal. Section 5 presents the related works and Section 6 concludes.

2   CHALLENGES OF AUTOMATIC SOFTWARE REPAIR

Automatic software repair is the process of fixing software bugs automatically. However, this broad definition covers techniques that are fundamentally different. First, "repair" means patching the program code, and the literature [Monperrus 2015] mainly refers to this as "program repair", "patch generation" or colloquially "automatic bug fixing". An archetypal program repair system is GenProg [Le Goues et al. 2012]. Second, repair also means changing the program state in response to field failures at runtime. The literature uses "runtime repair", "self-healing" and "failure-oblivious computing" to refer to this idea. An archetypal self-healing system is ClearView [Perkins et al. 2009].

Note that there is a key difference here: runtime repair does not mean generating source code patch as done in patch generation. Runtime repair in production means modifying the execution. For example, ClearView does not generate source code patches, it restores CPU register invariants at runtime. Reversely, GenProg cannot be used in production, since it takes as input test cases.

2.1   Challenges of Patch Generation

In production, end-users trigger bugs by exercising the system with inputs and sequences of interactions that were unforeseen by the development team. When this happens, they may report the bug to the development team through a support channel or a bug tracking system. The problem is that there is a huge gap between an issue reported by an end-user and a failing test case that is usable by a test-suite based repair system. The main problem is that the issue is reported in natural language by end users, and often the description is incomplete. The extreme difficulty of reproducing production bugs has been confirmed by numerous studies (e.g., [Bell et al. 2013; Bettenburg et al. 2008; Jin and Orso 2012]) and little support is available today for this task. Thus, current patch generation
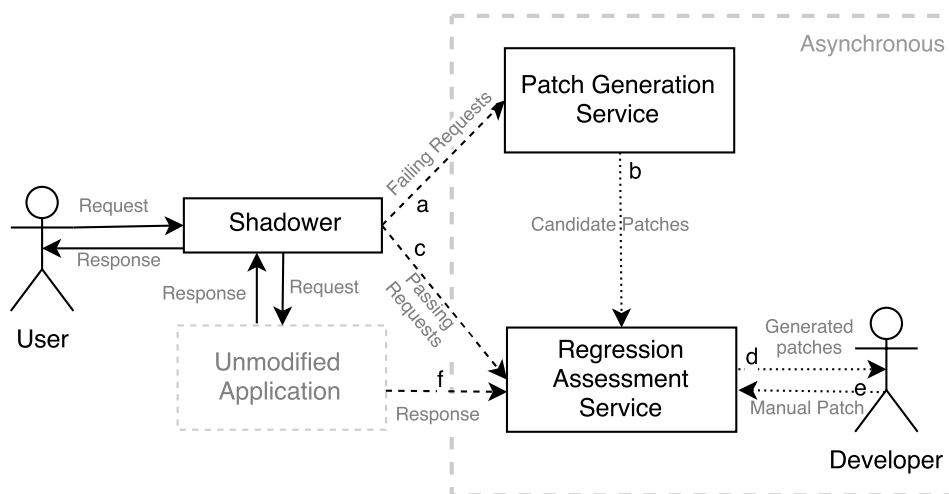
Production-Driven Patch Generation and Validation   1:3

Fig. 1. 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.

systems are inapplicable for production failures, because much manual work is needed to translate an end-user problem into a valid failing test case. Similarly, it is also difficult to convert production traffic into regression tests.

In this paper, we recast patch generation in a unique and novel way: we bring patch generation to production.

2.2   Challenge of Self-Healing Systems

Self-healing software systems have completely different challenges. The core idea of self-healing software is to modify the execution in response to a runtime failure. The first and foremost challenge of self-healing systems is to assess whether the state after healing is viable. Since self-healing software replaces a crashing failure into a continued execution, this execution is speculative in essence and no programmer has designed it upfront. There is very little work on assessing the viability of program states after self-healing [Locasto et al. 2008]. The second challenge is to transform those runtime modifications into valuable knowledge for developers. One solution is to log every execution modification, but this may represent an enormous volume of logs for any production system with large traffic, and this volume would be of little use for developers.

In this paper, the novelty we propose is that developers receive source code patches directly synthesized from production executions.

3   PROGRAM REPAIR 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 [Boner et al. 2014], 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 webserver, 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 [Adzic and Chatley 2017], 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.

3.1 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 1.

(1) The Shadower (see Section 3.1.3) 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 (see Section 3.1.1) is the service that searches for patches that fix a given failure. It uses a failure detection mechanism called "Request-oracle" in this paper to determine whether the application has successfully handled a request.
(3) The Regression Assessment Service (see Section 3.1.2) performs regression testing on the patches based on user traffic. It applies the generated patches on a copy of the application – 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 1 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 1 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.

3.1.1 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 webserver, 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 [Durieux et al. 2017; Rinard et al. 2004]. Itzal is agnostic to the patch service, it supports several patch models: we have implemented two of them, both used in our evaluation.

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

Production-Driven Patch Generation and Validation   1:5

Algorithm 1 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:   $output \leftarrow A(r)$
 3:   send $output$ to Client
 4:   **if** r produces a failure **then**
 5:     $patches \leftarrow G(r)$
 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**

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 1) 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 [Sidiroglou-Douskos et al. 2015]) or generic (e.g., Genprog [Le Goues et al. 2012]). 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.

3.1.2 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: an "execution comparison oracle" is a mechanism to detect changed behaviors in production.

Algorithm 2 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**

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 2 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 1 and line 1 in Algorithm 2), the Regression Assessment Service iterates over each patch to detect regressions (lines 3-12) in Algorithm 2). Finally, the patches that pass user traffic based regression testing are sent to the developers (arrow d in Figure 1 and line 13 in Algorithm 2).

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.

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 1. In this case, it means that the Regression Assessment Service is used for live testing of code on user traffic.

3.1.3 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 1). The response is also shadowed for the Regression Assessment Service (arrow f in Figure 1).

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

Definition: 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[1]. 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.

3.2   Prototype Implementation for Java

We have implemented a prototype of Itzal for Java in a tool named Itzal4j, which is dedicated to message-driven applications based on HTTP. For sake of open-science, Itzal4j is publicly available on http://anonymous.4open.science/repository/f5722a25-5510-4dd7-bd04-664d5f47715a/.

Itzal4j 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 Section 4, 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 succeed.

3.2.1   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 [Durieux et al. 2017] which addresses null dereferences in Java. Second, our prototype system also implements the exception-stop model [Dobolyi and Weimer 2008], which prevents the failure from happening by adding try-catch blocks at the method level.

3.2.2   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

---

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

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 Itzal4j, 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".

3.2.3 Implementation of the Shadower. The Shadower is implemented with the Jetty Proxy Library.[2] 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).

3.2.4 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) [Docker [n. d.]]. 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.

3.2.5 Communication with the Developer. Now we discuss the communication of the patches with the developers (arrow d, e in Figure 1). 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 1 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 2). The idea is that the more a patch has been executed by the Regression Assessment Service, the more confidence we have in it.

4 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 2 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 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.

---

[2]Jetty Proxy Servlet http://www.eclipse.org/jetty/documentation/9.4.x/proxy-servlet.html
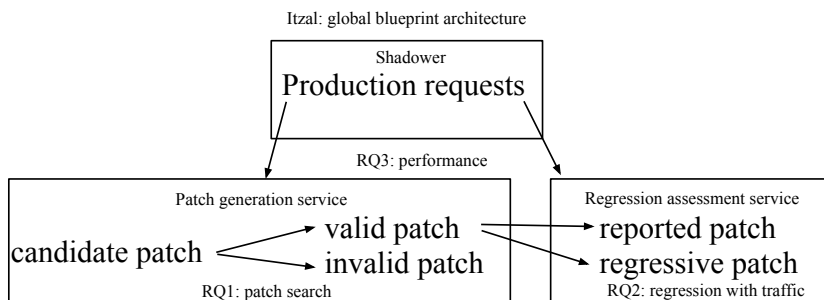
Fig. 2. Our research questions target each component in isolation as well as the global end-to-end approach.

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 4 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.

4.1 RQ1. Live Patch Generation Feasibility

4.1.1 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: [Just et al. 2014], [Long et al. 2017], [Gu et al. 2016], [Durieux et al. 2017] and [Durieux et al. 2016a].

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 [Durieux et al. 2018].

In Table 1, 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.

Table 1. The benchmark used in our experiments

Bug	Origin	Request Oracle	Bug Type	LOC
BroadleafCommerce 1282	[Durieux et al. 2016a]	HTTP status	NPE	161 428
Collection 360	[Durieux et al. 2017]	Exception	NPE	21 650
DataflowJavaSDK c06125d	[Long et al. 2017]	Exception	NPE	50 655
Felix 4960	[Durieux et al. 2017]	Exception	NPE	33 057
Javapoet 70b38e5	[Long et al. 2017]	Exception	NPE	3 884
Jetty 335500	[Gu et al. 2016]	HTTP status	NPE	153 789
Jongo f46f658	[Long et al. 2017]	Exception	NPE	7 384
Lang 20	[Just et al. 2014]	Exception	NPE	49 637
Lang 304	[Durieux et al. 2017]	Exception	NPE	17 277
Lang 33	[Just et al. 2014]	Exception	NPE	45 444
Lang 39	[Just et al. 2014]	Exception	NPE	45 143
Lang 587	[Durieux et al. 2017]	Exception	NPE	17 319
Lang 703	[Durieux et al. 2017]	Exception	NPE	19 047
Math 1115	[Durieux et al. 2017]	Exception	NPE	90 782
Math 1117	[Durieux et al. 2017]	Exception	NPE	90 794
Math 290	[Durieux et al. 2017]	Exception	NPE	38 728
Math 305	[Durieux et al. 2017]	Exception	NPE	38 893
Math 369	[Durieux et al. 2017]	Exception	NPE	41 082
Math 4	[Just et al. 2014]	Exception	NPE	164 667
Math 70	[Just et al. 2014]	Exception	NPE	83 720
Math 79	[Just et al. 2014]	Exception	NPE	89 611
Math 988A	[Durieux et al. 2017]	Exception	NPE	82 442
Math 988B	[Durieux et al. 2017]	Exception	NPE	82 443
Mayocat 231	[Durieux et al. 2016a]	HTTP status	NPE	31 231
PDFBox 2812	[Durieux et al. 2017]	Exception	NPE	67 294
PDFBox 2965	[Durieux et al. 2017]	Exception	NPE	64 375
PDFBox 2995	[Durieux et al. 2017]	Exception	NPE	64 821
Sling 4982	[Durieux et al. 2017]	Exception	NPE	583
Tomcat 43758	[Gu et al. 2016]	HTTP status	NPE	156 480
Tomcat 54703	[Gu et al. 2016]	HTTP status	NPE	186 301
Tomcat 55454	[Gu et al. 2016]	HTTP status	NPE	193 648
Tomcat 56010	[Gu et al. 2016]	HTTP status	IAE	195 130
Tomcat 58232	[Gu et al. 2016]	HTTP status	NPE	224 194
Webmagic ff2f588	[Long et al. 2017]	Exception	NPE	9 239
34 bugs from 14 software applications				2 622 172

4.1.2 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 a HTTP request that triggers a failure. Based on this failure the Patch Generation Service searches for patches. The main novelty of this setup compared to test-suite based program repair is the following: while

Production-Driven Patch Generation and Validation   1:11

Table 2. 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

previous experiments assume a manually written failing test case, Itzal only assumes a failing request. This is 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 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 a 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 3.2.1 to generate patches for the bugs with. We count the number of invalid and valid patches for each failure and for each patch model.

4.1.3 Results. We now present the results for this research question. To answer this question, we consider the columns *# NPEFix Valid/Invalid* and *# Exception-Stopper Valid/Invalid* of Table 2, 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 2 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 2 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 3.2.5.

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.

4.2   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.

4.2.1   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[3], BroadLeaf Commerce[4], and Shopizer[5]. 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 [Durieux et al. 2018].

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 3 present this benchmark.

---

[3]http://www.mayocat.org/
[4]http://www.broadleafcommerce.com/
[5]http://www.shopizer.com/

1:14   Thomas Durieux, Zhongxing Yu, Youssef Hamadi, and Martin Monperrus

Table 3. 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

Production-Driven Patch Generation and Validation   1:15

4.2.2   Experimental Protocol for RQ2. To evaluate the Regression Assessment Service, we first execute the user traffic as described in Section 4.2.1 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 paper, following the terminology introduced in [Palikareva et al. 2016]. 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.

4.2.3   Results. Table 3 contains the data obtained with the experimental protocol described in Section 4.2.2, investigating whether the execution comparison oracles considered are able to detect divergences. The first column of Table 3 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 3 provide the measured divergence between the original program and the seeded programs using the 4 execution comparison oracles presented in Section 4.2.2. 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 3 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 3 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 $\otimes$ for all the 4 oracles in Table 3, e.g., AbstractScopeCookieContainerFilter:256. The reason of 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 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.

4.3 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.

4.3.1 Shadower Overhead. As we have previously discussed in Section 3.1.3, 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 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.

4.3.2 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 3.1.3). 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 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.

4.4 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 ecommerce 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.

4.4.1 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 used in the experiment for RQ2,

4.4.2 End-to-end Evaluation on Mayocat-231.

1:18   Thomas Durieux, Zhongxing Yu, Youssef Hamadi, and Martin Monperrus

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

Fig. 3. The human patch for bug Mayocat 231.

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

Fig. 4. An invalid Itzal patch for bug Mayocat 231.

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

Fig. 5. 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. Figure 3 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 [Durieux et al. 2018]). 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 Figure 4. This patch is invalid according to the Request-oracle,

```
void populateEntityForm(...) {
    ...
    String idProperty = adminEntityService.getIdProperty(cmd);

    // null pointer exception here
    // because entity.findProperty(idProperty) is null when idProperty is not present in "
        entity"
    ef.setId(entity.findProperty(idProperty).getValue());
    ...
}
```

Fig. 6. The failure point of bug BroadleafCommerce-1282.

```
    adminInstance.setUsername(adminInstance.getEmailAddress());
    if (customerService.readCustomerByUsername(adminInstance.getUsername()) != null) {
-       Entity error = new Entity();
-       error.addValidationError("username", "nonUniqueUsernameError");
-       return error;
+       entity.addValidationError("emailAddress", "nonUniqueUsernameError");
+       return entity;
    }
```

Fig. 7. The human patch for bug BroadleafCommerce-1282 (simplified version).

because it produces a 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 a 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, Figure 5 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 [Long and Rinard 2016a], it is common to have many equivalently correct yet syntactically different patches in the search space of a patch model.

4.4.3 End-to-end Evaluation on BroadleafCommerce-1282. We now study the case of bug Broadleaf Commerce-1282 (https://github.com/BroadleafCommerce/BroadleafCommerce/issues/1282), 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.

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

Fig. 8. The Itzal patch for bug BroadleafCommerce-1282.

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.

Figure 6 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 patch for this bug (Again, the complete list of Itzal patches is available at [Durieux et al. 2018]). 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 Figure 8. 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 Figure 7) 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 exception 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 RELATED WORK

5.1 Test Suite Based Patch Generation

The literature on test suite based patch generation is growing very fast, and we here only present a brief overview of notable contributions. GenProg [Le Goues et al. 2012] applies genetic programming to the AST of a buggy program and generates patches by adding, deleting, or replacing AST nodes. Prophet [Long and Rinard 2016b] and Genesis [Long et al. 2017] learn from existing successful human patches to improve the repair success rate. SemFix [Nguyen et al. 2013b] is a constraint based repair approach. It provides patches for assignments and conditions by combining symbolic execution and code synthesis. DirectFix [Mechtaev et al. 2015] and Angelix [Mechtaev et al. 2016] from the same team provide further improvement over SemFix. Nopol [Xuan et al. 2016] is also a constraint based method, which focuses on repairing bugs in if-conditions and missing preconditions. There are also some template-based patch generation approaches. PAR [Kim et al. 2013] uses 10 patch templates for common programming errors and Relifix [Tan and Roychoudhury 2015] defines templates specifically for regression bugs.

Main novelty: Itzal does patch synthesis directly in production, which none of those contributions address. Itzal does not require a failing test case or a strong regression test suite.

5.2 Runtime Repair in Production

Several automatic repair techniques handle failures in production, we now review the notable ones.

Rx [Qin et al. 2005] is a runtime repair system based on changing the environment upon failures. Rx employs checkpoint-and-rollback for re-executing the buggy code when failures happen. Assure [Sidiroglou et al. 2009] is a self-healing system also based on checkpointing. For both of them, there is no patch generation strategy associated with the checkpoint and rollback mechanism.

Rinard et al. [Rinard et al. 2004] present a technique called "failure oblivious computing" to avoid illegal memory accesses by adding additional code to each memory operation during the compilation process. Dobolyi and Weimer [Dobolyi and Weimer 2008] present a technique to tolerate null dereferences by introducing hooks to inject manufactured values. Long et al. [Long et al. 2014] also explore this idea with the concept of "recovery shepherding". Upon certain errors (null dereferences and divide by zero), recovery shepherding consists of returning manufactured values for failure oblivious computing. Perkins et al. [Perkins et al. 2009] propose ClearView, a system for automatically repairing errors in production by monitoring the system execution on low-level registers to learn invariants. Gu et al. [Gu et al. 2016] present Ares, a runtime error recovery technique for Java exceptions using JavaPathFinder (JPF).

Main novelty: these techniques do not directly produce source code patches that are communicated to developers. On the contrary, Itzal provides a fully automated bridge between production and source code patches for developers.

5.3 Shadow Traffic

The concept of shadow traffic is related with the execution of multiple versions of the same software in parallel, called in the literature "multi-version execution" [Hosek and Cadar 2013], "parallel execution" [Trachsel and Gross 2010] or simply "dual execution" [Kwon et al. 2016] when only two versions run. However, none of these work aims to generate patches. Using shadow traffic for repair has only recently been suggested in a short vision paper [Durieux et al. 2016b], which remains at the concept level.

For security. Kwon et al. [Kwon et al. 2016] do dual execution for detecting information leaks and attacks. Salamat et al. [Salamat et al. 2009] do multi-version execution also for the sake of security, and they implement the monitor entirely in userspace. In both cases, patch generation is not

considered. The idea of shadow traffic is closely related to the idea of shadow executions introduced by Capizzi et al. [Capizzi et al. 2008]. However, the goals are completely different. While they aim at isolating all private information, our goal is to perform patch search.

For performance. Trachsel and Gross [Trachsel and Gross 2010] perform parallel execution to speed up programs. The instances that are run in parallel are different implementations of the same algorithm or different binary versions compiled with different optimization options. Compared to Itzal, no actionable feedback is given to the developer.

For reliability. Hosek and Cadar [Hosek and Cadar 2013] do multi-version execution over versions and switch between versions when a bug is detected. This technique can handle failures because some bugs may disappear between different versions. It is not clear how this technique can be used for patch generation.

Main novelty: to our knowledge, Itzal is the first approach to use production traffic for validating automatically synthesized patches.

6 CONCLUSION

In this paper, 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 on 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.

REFERENCES

Gojko Adzic and Robert Chatley. 2017. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 884–889.

Jonathan Bell, Nikhil Sarda, and Gail Kaiser. 2013. Chronicler: Lightweight Recording to Reproduce Field Failures. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 362–371.

Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 308–318. https://doi.org/10.1145/1453101.1453146

Jonas Boner, Dave Farley, Roland Kuhn, and Martin Thompson. 2014. The Reactive Manifesto. http://www.reactivemanifesto.org/. accessed: 4/2018.

Steve Burton. 2016. 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/.

R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. 2008. Preventing Information Leaks through Shadow Executions. In *Computer Security Applications Conference*.

Kinga Dobolyi and Westley Weimer. 2008. Changing java's semantics for handling null pointer exceptions. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*. IEEE, 47–56.

Docker. [n. d.]. Play in a content trust sandbox. https://docs.docker.com/engine/security/trust/trust_sandbox/. accessed: 4/2018.

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

Thomas Durieux, Youssef Hamadi, and Martin Monperrus. 2016a. BanditRepair: Speculative Exploration of Runtime Patches.

Thomas Durieux, Youssef Hamadi, and Martin Monperrus. 2016b. Production-driven patch generation and validation. *arXiv preprint arXiv:1609.06848* (2016).

Production-Driven Patch Generation and Validation 1:23

Thomas Durieux, Youssef Hamadi, and Martin Monperrus. 2018. Open-science companion data. https://github.com/Spirals-Team/itzal-experiments/.

Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. 2016. Automatic runtime recovery via error handler synthesis. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 684–695.

Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 612–621.

Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 474–484.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. 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)*. 437–440.

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering*. 802–811.

Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality Inference by Lightweight Dual Execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*.

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

Michael E Locasto, Angelos Stavrou, and Gabriela F Cretu. 2008. *Life after self-healing: assessing post-repair program behavior*. Technical Report. Technical report, George Mason University.

Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 727–739.

Fan Long and Martin Rinard. 2016a. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering*. 702–713.

Fan Long and Martin Rinard. 2016b. Automatic Patch Generation by Learning Correct Code. *SIGPLAN Not.* 51, 1 (jan 2016), 298–312.

Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 227–238.

Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2016. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering* (2016), 1–29.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 691–701.

Martin Monperrus. 2015. *Automatic Software Repair: a Bibliography*. Technical Report hal-01206501. Inria.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013a. SemFix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. 772–781.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013b. SemFix: Program repair via semantic analysis. In *Proceedings of the International Conference on Software Engineering*. 772–781.

David Oppenheimer, Archana Ganapathi, and David A Patterson. 2003. Why do Internet services fail, and what can be done about it?. In *USENIX symposium on Internet Technologies and Systems*, Vol. 67. Seattle, WA.

Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1181–1192.

Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 87–102.

Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. 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*. ACM, 24–36.

F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. 2005. Rx: treating bugs as allergies – a safe method to survive software failures. In *ACM SIGOPS Operating Systems Review*, Vol. 39. 235–248.

Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing.. In *OSDI*, Vol. 4. 21–21.

Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. 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*. ACM, 33–46.

Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D Keromytis. 2009. Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News* 37, 1 (2009), 37–48.

Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. 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)*. ACM, New York, NY, USA, 43–54.

Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. 2015. 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)* (2–4).

Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated Repair of Software Regressions. In *37th International Conference on Software Engineering*. IEEE.

Oliver Trachsel and Thomas R. Gross. 2010. Variant-based Competitive Parallel Execution of Sequential Programs. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*.

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