2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Results Track

# Production-Driven Patch Generation

Thomas Durieux
University of Lille & Inria Lille, France

Youssef Hamadi
Ecole Polytechnique, LIX, France

Martin Monperrus
University of Lille & Inria Lille, France

***Abstract*—We present an original concept for patch generation: we propose to do it directly in production. Our idea is to generate patches on-the-fly based on automated analysis of the failure context. By doing this in production, the repair process has complete access to the system state at the point of failure. We propose to perform live regression testing of the generated patches directly on the production traffic, by feeding a sandboxed version of the application with a copy of the production traffic, the "shadow traffic". Our concept widens the applicability of program repair, because it removes the requirements of having a failing test case.**

## I. Introduction

Program repair requires the presence of a failing test case to reproduce a failure that has happened in production. Writing such a failing test case is a really hard task, because the developer in charge of reproducing a failure has little access to the system state at the point of failure (she basically only has logs). The difficulty of reproducing production failures has a direct impact on applicability of program repair: with no failing test, there is no patch generation. We aim at weakening the requirements of program repair by removing the mandatory presence of a failing test case.

Our intuition is to perform program repair directly in production, so that the repair process has a direct access to the system state at the point of failure. This paper presents an architecture, called Itzal, it generates patches without requiring a failing test case. The process of Itzal is as follows. First, Itzal uses production assertions or runtime exceptions to detect failures. Second, right after the failure is detected in production, a patch is searched in a sandboxed environment that mimics the production one. If a patch fixes the failure, it is a "candidate patch". Third, the patches are tested for regression, directly in production, based on traffic that is an exact copy of the production traffic – we call it shadow traffic. Itzal has been realized in a prototype implementation for Java which focuses on generating source code patches for null dereferences.

This is a new line of research in automatic repair. Compared to classical test-suite based patch generation (e.g. [1]), Itzal does patch generation online, i.e. as soon as the failure happens, with no need for reproducing the failure. Yet, Itzal is not a classical runtime repair technique either (e.g. [2]): while the patches are generated online in production, the system state is never altered. The Itzal patches are applied later, once the developer has validated them.

To sum up, our contributions are:

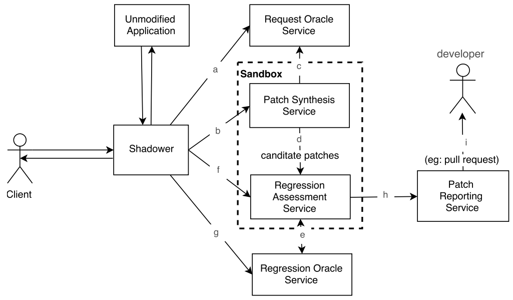

Figure 1: The Architecture of Itzal. The key idea is to duplicate production traffic via a "shadower", the duplicated traffic is used to search for patches and to validate candidate patches.

- Itzal, an architecture for patch generation in production that does not require a failing test case.
- The use of shadow production systems and shadow traffic in the context of automatic repair to generate patches in production.
- The design and implementation of a Java implementation of this vision for null pointer exceptions.

This paper is based on content from Arxiv's document #1609.06848 [3] and is structured as follows. Section II presents Itzal. Section III presents the related works and Section IV concludes.

## II. Itzal

We now present Itzal, a novel software repair technique for generating patches without requiring failing test cases, directly in the production environment. Itzal is a "testless" patch generation approach.

### A. 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 and implicit contracts such 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 under 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. This has the advantage that the regression exactly corresponds to what actually matters.

978-1-5386-2675-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-NIER.2017.8

### B. Architecture

The Itzal architecture is composed of seven components, as shown in Figure 1.

1) The Unmodified Application (see Section II-B1) is the application onto which automatic patch generation is plugged.
2) The Request Oracle Service (see Section II-B2) is a service that determines whether the application has successfully handle a request.
3) The Patch Synthesis Service (see Section II-B3) is the service that searches for patches that fix a given failure.
4) The Regression Assessment Service (see Section II-B4) performs regression testing on the generated patch. It applies the generated patches on the application and executes the request on it.
5) The Regression Oracle (see Section II-B5) is the component that validates the generated patches by comparing the original output of Unmodified Application to the output of the patched application for live production requests.
6) The Shadower (see Section II-B6) is used to duplicate the requests of the Unmodified Application. The duplicated requests are then sent in parallel to the Patch Synthesis Service and the Regression Assessment Service.
7) And the Patch Reporting Service (see Section II-B7) is the component that selects the best patches and communicates them to the developers.

Algorithm 1 shows the interactions between each component of the Itzal. Itzal 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, yet it is additionally sent to the Request Oracle Service (arrow *a* in Figure 1) which verifies the viability of the output (line 4). If the Request Oracle Service determines that there is a failure, the request is sent to the Patch Synthesis Service (arrow *b* in Figure 1 and line 5). The patches generated by Patch Synthesis Service which pass the Request Oracle Service (i.e. fix the failure at hand) are sent to Regression Assessment Service (line 6). If the request has succeeded (no failure on the original application), the request is also sent to the Regression Assessment Service (line 8) where all the previously generated patches are being validated on-the-fly against the new request. When the Regression Assessment Service has identified valid patches with no regression, it sends them to the Patch Reporting Service.

To sum up, Itzal does patch generation online, i.e. as soon as the failure happens, directly in production. However, while the patches are generated online, they are applied later, once the developer has validated them. The side effects of patch search or regression testing on the production state are completely sandboxed, with no interference with the production environment.

*1) The Unmodified Application:* Itzal augments a production application with automatic patch generation capabilities. The requirement to deploy Itzal is that the application must use requests, ie. must have a message-driven architecture. A web application, or a web REST service are examples of message-driven applications.

**Algorithm 1** The main Itzal algorithm

```
Input: A: the Unmodified Application
Input: G: the Patch Synthesis Service
Input: V: the Regression Assessment Service
Input: O: Request Oracle Service
 1: while new request $r_{client}$ from Client do
 2:     $output$ = A($r_{client}$)
 3:     send $output$ to Client
 4:     if O($output$) is failure then
 5:         $patches$ = send $r_{client}$ to G
 6:         push $patches$ to V
 7:     else
 8:         send $r_{client}$ to V for regression
 9:     end if
10:     if $\exists$ validated patches $\in V$ then
11:         $p \leftarrow$ order the patches
12:         report $p$ to developers
13:     end if
14: end while
```

The type of request may vary between applications, for example a request in a web application will be the request sent by a user's browser to a web-server, in a micro-service application, the request will typically be a REST message, in a mobile application, a request would be a touch event triggered when a user touches a mobile device's screen.

*2) Request Oracle Service:* The responsibility of the Request Oracle Service is to verify whether the application has succeeded to answer the request. For instance, in a web-server, the Request Oracle Service can check the HTTP request return code ("assert response_code != 500 (internal server error)"), of 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). When possible, the Request Oracle Service provide some information about the failure to help the Patch Synthesis Service to search for a patch. The information provided by default is the stack trace when the failure is based on an exception.

The Itzal does not require a perfect Request Oracle Service, i.e. the Request Oracle Service may miss some failures (false negatives). In the case of false negatives, when the Request Oracle Service misses the failure detection, Itzal simply does not generate patches: this is unfortunate but it impacts neither the original application nor the patches generated for the other failures. In the case of false positives, when the Request Oracle Service detects a failure when there is no failure in reality, the Patch Synthesis Service would generate patches, yet they would be lilkely benign if they pass regression testing done by the Regression Assessment Service.

*3) Patch Synthesis Service:* The Patch Synthesis Service is the service that synthesizes patches that fix a failing request. Itzal can work with any patch synthesis approach compatible with the Request Oracle Service. The patches are applied to the failure point, for instance at the line where an exception has occurred. For a given failure point, the Patch Synthesis Service performs an exhaustive application of all possible patches.

For each tentative patch, Patch Synthesis Service calls the Request Oracle Service (arrow *c* in Figure 1) to verify that the request has been correctly handled by the patch template under consideration (the failure has been fixed). Because the Patch Synthesis Service generates the patches only based on one request (the failing one), the patches may break the behavior of the application for other requests, in other word, the may introduce a regression. Thus, if the patch is successful on the failing request, the corresponding patch is transferred to the Regression Assessment Service (arrow *d* in Figure 1) that will further validates its correctness based on other requests.

The application and execution of candidate patches can change the state of the application in runtime. Consequently, each execution is done in a sandboxed environment, this nullifies the potential side effects of the request or of the patch templates. The sandboxed environment contains a shadow state of the application, which is regularly synchronized with the production one. Since the space of patches is sometimes large, Itzal uses a time budget. It explores the patch alternatives sequentially until they are all explored or until the time budget is consumed.

Itzal can work with any patch model, whether domain-specific (such as out-of-bounds exception) or generic (à la Genprog [1]). Similarly, Itzal can be applied to binary code if the patch synthesis technique supports it. Our current prototype generates patches for null dereferences. If the patch model generates too much patches, i.e. the search space is too large, this would be a problem because it would represent a huge computation effort on the Patch Synthesis Service and much more importantly on the Regression Assessment Service.

*4) Regression Assessment Service:* The patches generated by the Patch Synthesis Service can introduce regressions because their generation only involves one request (the failing one). The Regression Assessment Service has the responsibly to check the behavior of the application when the generated patches are injected against other requests. It detects these regressions by comparing the output of the Unmodified Application against the output of the patch-augmented application. If the output is different, it means that the patch has introduced a regression, and the patch is consequently marked as invalid. This comparison is done on-the-fly, directly on production traffic. Doing regression testing “live” has the advantage that there is no need to record the potentially enormous amount of production data.

*5) Regression Oracle:* The Regression Oracle compares the output of the Unmodified Application (arrow *g* in Figure 1) and the output of a patched version in the Regression Assessment Service for the same request. If the outputs are different, the Regression Oracle marks the current patch as invalid. For example, a regression oracle for a web server compares the HTML text of both versions. The comparison is not necessarily a byte-to-byte one, it can include heuristics to discard transient information such as time, cookie identifiers, etc.

*6) 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 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 Synthesis 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*, *b*, and *f* in Figure 1). The response is also shadowed for the regression oracle service (arrow *g* in Figure 1).

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 Synthesis Service asynchronously. Indeed, the goal of Itzal is to perform patch generation, not 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 end regression testing for sending the response back to the client. The Shadower is thus the only component that impacts the performance of the Unmodified Application. The Shadower requires to 1) copy and reroute requests on the fly and 2) maintain an appropriate shadow state of the system under consideration. In a typical HTTP-based setup, the cost of the former is similar to that of classical web proxies and load balancers, which are extensively used in production systems. The latter point is more an open question: very few works study production state shadowing, neither in academia nor in industry. It may be a costly operation if databases are naively copied for instance. However, we envision piggy-backing on the latest advances in efficient online backups and copies of file systems.

*7) Patch Reporting Service:* The Patch Reporting Service is the service that communicates the results of Itzal to the developers (arrow *i* in Figure 1).

It happens that multiple patches (corresponding to multiple patch templates) successfully pass the regression test over production traffic. Consequently the Patch Reporting Service has to sort the patches in order to first propose the most useful ones to the developers. To sort, the Patch Reporting Service uses the number of execution of the patched line in the Regression Assessment Service (the number of requests that execute the patch). The idea is that the more a patch has been executed by the Regression Assessment Service, the less likely it is to introduce a regression.

We now discuss the reporting medium to the human developer. There are several types of communication that can be used in the Patch Reporting Service. In the current prototype, we have a dashboard where the developers follow in real time the failures, the generated patches and the progression of the patch validation.

### C. Prototype Implementation for Java

We have implemented a prototype of Itzal for Java in a tool named Itzal4j, dedicated to reactive applications based on HTTP. Itzal4j generates patches for null dereference failures. In Itzal4j, the Request Oracle Service is based on exceptions. Any uncaught exception happening during the processing of a request is considered as a failure. The Patch Synthesis Service is dedicated to null pointers and uses the NPEFix technique [4] for searching the space of possible patches for null dereferences. Sandboxing of patch search is achieved using Docker, a major software containerization platform which provides powerful sandboxing (both disk and IO based). In our implementation, the Patch Synthesis Service sends all candidate patches to the Regression Assessment Service using an HTTP-based protocol. For the Regression Oracle, we compare the body of the HTTP response of the Unmodified Application against the output produced by the patched application (e.g. the HTML body text). If the outputs match, the patch is considered validated for the current request otherwise the patch is permanently marked as invalid. The Shadower is implemented on top of a HTTP proxy implementation in Java called "Jetty Proxy". The Patch Reporting Service of Itzal4j is a web dashboard, where the developers can access in real time the current patches of Itzal: the ones that have fixed at least one failure and the patch that are under regression testing. For each patch, they can visualize the number of failures of the system detected by the Request Oracle Service, see the actual patch code, and the patch validation metrics such as the number of executions done by the Regression Assessment Service.

## III. Related Work

Our work is much inspired by the classical work on runtime repair. Rinard et al. [2] present a technique called "failure oblivious computing" to avoid illegal memory accesses by adding additional code around each memory operation during the compilation process. Assure [5] is a self-healing system based on error-virtualization. Long et al. [6] proposes the concept of "recovery shepherding" in a system called RCV. Those techniques do not produce patches and do not perform regression testing in production.

Gu et al. [7] presents Ares a runtime error recovery for Java exceptions using JavaPathFinder (JPF). The two majors differences with Itzal4j are: first Itzal4j is safer, it does not modify the production state of the application as Ares does, and secondly, while Ares performs runtime repair, Itzal4j produces source code patches that are then communicated to the developers.

Perkins et al. [8] propose ClearView, a system for automatically repairing errors in production. Itzal and ClearView both perform repair in production, yet they are very different: 1) ClearView does not produce source code patches while Itzal does; 2) ClearView modifies the production state, while Itzal only modifies the sandboxed shadow requests and state (this means that ClearView can mess up the application while Itzal never does so); 3) ClearView works with learned invariant-based oracles, while Itzal uses human designed request oracles.

The concept of shadow traffic is related to the execution of multiple versions of the same software in parallel, called in the literature "multi-version execution" [9], or "parallel execution" [10]. However, none of the related work uses shadow traffic to generate patches.

## IV. Conclusion

In this paper, we have presented Itzal, an approach for generating patches in production. The failure detection that triggers the patch search is achieved with runtime assertions, and the regression assessment is done on live production traffic. In Itzal, the patch search is done in a fully sandboxed environment, with no interference with the production data. Our future work now consists of devising an approach to fully automatically store a shadow of the production state, and to efficiently synchronize the shadow state with the actual production state.

## References

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[2] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, "Enhancing server availability and security through failure-oblivious computing." in *OSDI*, vol. 4, 2004, pp. 21–21.

[3] T. Durieux, Y. Hamadi, and M. Monperrus, "Production-driven patch generation and validation," *arXiv preprint arXiv:1609.06848*, 2016.

[4] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic patch generation for null pointer exceptions using metaprogramming," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2017.

[5] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "Assure: automatic software self-healing using rescue points," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 37–48, 2009.

[6] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic runtime error repair and containment via recovery shepherding," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 227–238.

[7] T. Gu, C. Sun, X. Ma, J. Lü, and Z. Su, "Automatic runtime recovery via error handler synthesis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 684–695.

[8] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 87–102.

[9] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 612–621.

[10] O. Trachsel and T. R. Gross, "Variant-based competitive parallel execution of sequential programs," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, 2010.