Phoenix: A Tool for Automated Data-Driven Synthesis of Repairs for Static Analysis Violations

Hiroaki Yoshida
Fujitsu Laboratories of America, Inc.
hyoshida@us.fujitsu.com

Rohan Bavishi
University of California Berkeley
rbavishi@cs.berkeley.edu

Keisuke Hotta
Fujitsu Laboratories Ltd.
hotta-keisuke@fujitsu.com

Yusuke Nemoto
Fujitsu Laboratories Ltd.
y-nemoto@fujitsu.com

Mukul R. Prasad
Fujitsu Laboratories of America, Inc.
mukul@us.fujitsu.com

Shinji Kikuchi
Fujitsu Laboratories Ltd.
skikuchi@fujitsu.com

ABSTRACT

One of the major drawbacks of traditional automatic program repair (APR) techniques is their dependence on a test suite as a repair specification. In practice, it is often hard to obtain specification-quality test suites. This limits the performance and hence the viability of such test-suite-based approaches. On the other hand, static-analysis-based bug finding tools are increasingly being adopted in industry but still facing challenges since the reported violations are viewed as not easily actionable. In previous work, we proposed a novel technique that solves both these challenges through a technique for automatically generating high-quality patches for static analysis violations by learning from previous repair examples. In this paper, we present a tool Phoenix, implementing this technique. We describe the architecture, user interfaces, and salient features of Phoenix, and specific practical use cases of its technology. A video demonstrating Phoenix is available at https://phoenix-tool.github.io/demo-video.html.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Programming by example**; **Software testing and debugging**.

KEYWORDS

Program Repair, Static Analysis, Programming-by-Example

ACM Reference Format:
Hiroaki Yoshida, Rohan Bavishi, Keisuke Hotta, Yusuke Nemoto, Mukul R. Prasad, and Shinji Kikuchi. 2020. Phoenix: A Tool for Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382150

1 INTRODUCTION

Since software debugging is one of the most time-consuming and labor-intensive phases of the software development lifecycle, its automation has the potential to significantly improve developer productivity. A number of *automatic program repair (APR)* [8, 15] techniques have been proposed to address this challenge. Typical APR techniques rely on a test suite as a repair specification, i.e., an oracle that fails on the buggy program, to validate the correctness of generated patches for the bug. However, in practice, the quality of test suites is often far from satisfactory for this purpose.

Static program analysis tools are increasingly being used as an alternative approach to find software quality issues. However, one of the key barriers to the successful adoption of this technology is the lack of actionable suggestions from the tools on how to fix the quality issues that have been flagged. Recently, development environments such as IntelliJ [12], Eclipse [11], and Visual Studio [14] have started offering a *"quick fix"* suggestion feature to fix simple programming errors automatically, based on a pre-determined set of manually-defined fix templates. Such fix templates are carefully designed to maximize their versatility, i.e., being applicable to many different program contexts. This invariably results in the fix templates being rather simple and generic. In other related work, the FootPatch tool [18] proposed the automatic detection and repair of heap properties using static analysis. However, this technique is specifically developed for a limited class of bugs and non-trivial to extend to more generic bug-fixing.

In [4], we proposed a learning-based automatic repair technique for static analysis violations that is comprised of two key elements: (1) it uses an off-the-shelf static analyzer, such as FindBugs [1], Infer [5], or error-prone [9], as an oracle, to detect potential bugs and also to validate a patch as a viable fix for such a bug, and (2) it learns repair strategies from previous repair examples mined from a large corpus of open-source projects (a resource often referred to as Big Code [6]). This eliminates the dependence on a test suite while leveraging the now widely used static analyzers. Our learning-based approach is quite general in that it is not tied to a specific class of bugs or repair strategies. At the same time, it can learn more elaborate repair strategies very specific to each bug type and program context, potentially increasing the accuracy and the scope of bug fixes.

This paper addresses the implementation and deployment of the Phoenix tool. Specifically, the main contributions of this paper are:

- Two practical usage scenarios of Phoenix (Section 3).
- A detailed description of the software architecture and the user interfaces of Phoenix (Section 4).
- Initial results from an industrial deployment of Phoenix on a financial transaction processing system (Section 6).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3382150

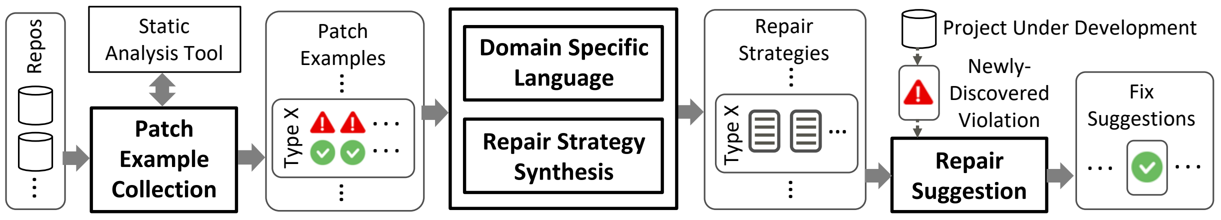


Figure 1: Overview of PHOENIX system.

2 TECHNIQUE

The high-level overview of the PHOENIX system is shown in Figure 1. This fully-automated pipeline consists of three main stages: (1) *collecting* patches for static-analysis violations, (2) *learning* repair strategies from the collected patches, and (3) *suggesting* repairs for unseen violations.

For collecting patches, PHOENIX analyzes repositories from Big Code, and runs the static analyzer on each commit of a given repository, to obtain a list of static analysis violations for that commit. Then, it applies a violation tracking procedure to obtain the violations that were fixed as well as the corresponding patch that fixed them. This yields a collection of patches, which are further cleaned by filtering out edits that are irrelevant to the fix of the violation. These cleaned patches are clustered into collections of patch examples corresponding to each distinct violation type, for the subsequent learning stage.

PHOENIX employs a Programming-by-Example (PbE) technique, specifically domain-specific inductive synthesis [17], to learn generic, high-level repair strategies in a specialized domain-specific language (DSL). Our novel DSL is designed to be expressive enough to capture common edit operations frequently observed in fixes of static analysis violations and at the same time compact enough to efficiently guide the learning procedure to a viable solution without search space explosion. Conceptually, the synthesis algorithm in PHOENIX is comprised of two steps: (1) enumerate all possible strategies for all repair examples and (2) find the fewest number of strategies amongst these that can solve all the repair examples.

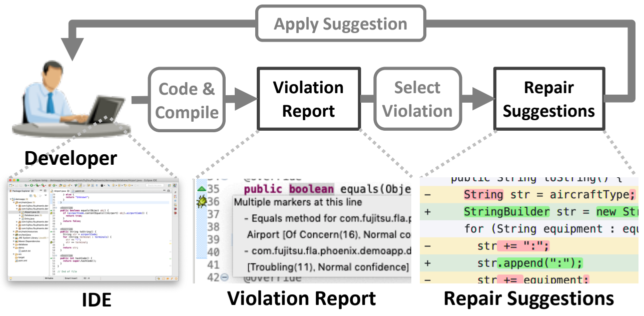
Finally, when an unseen violation is discovered, the strategies for that specific type of violation are selected and applied to the buggy code to generate repair suggestions. For more details, the interested reader is referred to our technical paper [4].

3 USAGE SCENARIOS

3.1 Integration with IDEs

The most straightforward use of PHOENIX is as a plugin installed in an Integrated Development Environment (IDE), as shown in Figure 2. In this use case, the developer writes code, the IDE automatically compiles the code, performs static analysis, and displays a report of violations found, if any. PHOENIX can generate suggestions to repair those violations. Specifically, the developer selects the violation to fix and is shown one or more repair suggestions. The developer can then choose one of the suggestions, and after confirming that it actually repairs the violation correctly, apply it to the actual code.

The advantage of this scenario is that developers can get started by simply installing the plugin, which provides the same UI as the



```
public boolean equals(Obje
Multiple markers at this line
 - Equals method for com.fujitsu.fla.p
 Airport [Of Concern(16), Normal co
 - com.fujitsu.fla.phoenix.demoapp.d
 [Troubling(11), Normal confidence]
```

```
-    String str = aircraftType;
+    StringBuilder str = new Str
     for (String equipment : equ
-      str += ":";
+      str.append(":");
-      str += equipment;
```

Figure 2: Integration with IDEs.

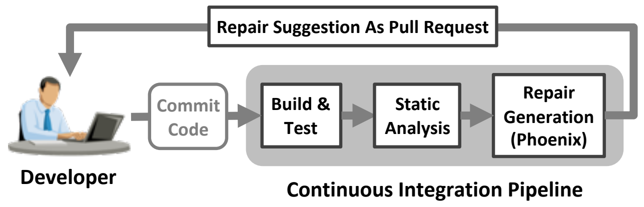

Repair Suggestion As Pull Request
Developer
Commit Code
Build & Test
Static Analysis
Repair Generation (Phoenix)
Continuous Integration Pipeline

Figure 3: Integration with SCM/CI.

existing code suggestion features, minimizing developers' learning costs. On the other hand, an obvious drawback is that it is necessary to provide plugins for multiple IDEs, such as Eclipse or IntelliJ IDEA. We have currently implemented a plugin for Eclipse IDE, one of the widely-used IDEs, on top of the PHOENIX core engine. This is discussed further in Section 4.

3.2 Integration with Source Code Management and Continuous Integration System

Another scenario is to run PHOENIX when developers commit code to a repository. In a typical Continuous Integration (CI) pipeline, when code is committed, build and test are executed, and optionally a static analyzer is executed. As shown in Figure 3, PHOENIX may be used to generate repair suggestions for the violations discovered during the CI pipeline, and the repair patches are fed back to the developers as pull requests.

The advantage of this scenario is that, unlike the IDE scenario, it is not necessary to support multiple IDEs, and hence a generic command line tool that outputs repair patches from buggy source code as input should suffice for such a CI pipeline integration. The downside is that repairs may possibly be delayed or overlooked because the tool cannot make suggestions as soon as violations are discovered, as in the IDE scenario.

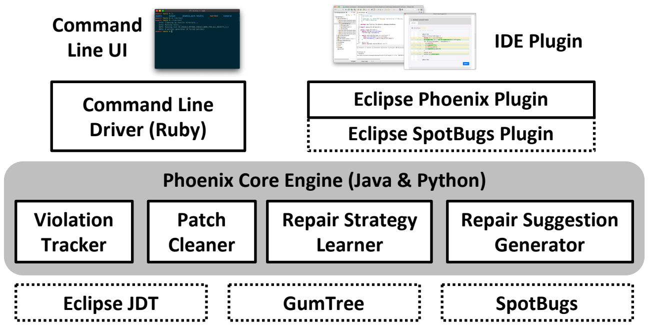


Figure 4: Phoenix system architecture.

4 TOOL DESCRIPTION

Phoenix: We have implemented the fully-automated pipeline, shown in Figure 1, in a tool called Phoenix. As shown in Figure 4, the Phoenix core engine consists of four main components: the violation tracker component in Python, and the patch cleaner, repair strategy learner and suggestion generator components in Java. We use Eclipse JDT to manipulate ASTs, `GumTree` [7] to compute AST differences and employ `SpotBugs`[1] as the static program analyzer.

IDE Plugin: Eclipse provides a standard UI for suggesting code fixes, called Quick Fix. The Phoenix plugin we have developed utilizes the Quick Fix UI to provide repair suggestions for violations. As shown in Figure 5 (a), by right-clicking the SpotBugs indicator of the violation, Phoenix repair suggestions are displayed next to other code suggestions. Clicking on it brings up the Phoenix window where developers can browse the description of the violation and the repair suggestions (Figure 5 (b)). The Phoenix window also shows the previous repair examples that were used to learn the strategies (Figure 5 (c)). In case developers are not confident about the suggestions, these examples serve as an illustration of how other projects have repaired similar violations. Once the developer confirms that the suggestion is correct, simply pressing the APPLY button deploys the suggestion to the actual code.

Command Line UI: The command line interface (CLI) of Phoenix allows for its integration into automated services such as CI/CD pipelines. Given the source code of the target project and its compiled JAR file, the CLI automatically executes SpotBugs, collects all violations, generates repair suggestions for each violation, and outputs HTML reports. The reports include the descriptions of the violations and their repair suggestions.

5 EVALUATION

We summarize the evaluation of Phoenix reported in [4]. The evaluation was conducted on a dataset that consists of 5,389 unique patches extracted from 517 popular Github projects, spanning 234 distinct types of `SpotBugs` violations. Table 1 shows the salient statistics of this dataset.

Effectiveness of Phoenix. For this evaluation we perform a project-level, leave-one-out cross-validation experiment on our dataset. For each patch in a violation category, we learn repair strategies from all the patches in *other* projects, in the same category, and apply them to the buggy source in the patch. We compute **recall** as the percentage of cases where at least one fix is generated, i.e., the violation is removed, as judged using `SpotBugs`. The results are shown in table Table 2. Phoenix achieves a recall of 85% indicating that it is able to generalize across projects.

[1]The results in [4] were reported using the `FindBugs` static analyzer. But our current tool implementation now uses its successor `SpotBugs`, which is substantially similar.

Table 1: Summary of Phoenix dataset.

#Projects	517	#Commits	3,549,436
#Analyzed commits	290,519	#Fixing commits	21,028
#Collected patches(cleaned)	11,865	#Unique patches	5,389
#Violation types	234	Collection time	2 months

Table 2: Leave-one-project-out Cross Validation Results

#Dataset patches	5,389	#Manually-inspected patches	465
#Successful fixes	4,596	#Semantically-equiv. (Top-1)	161 (35%)
		#Semantically-equiv. (Top-5)	203 (44%)
		#Correct (Top-1)	**252 (54%)**
Recall	**85%**	#Correct (Top-5)	299 (64%)

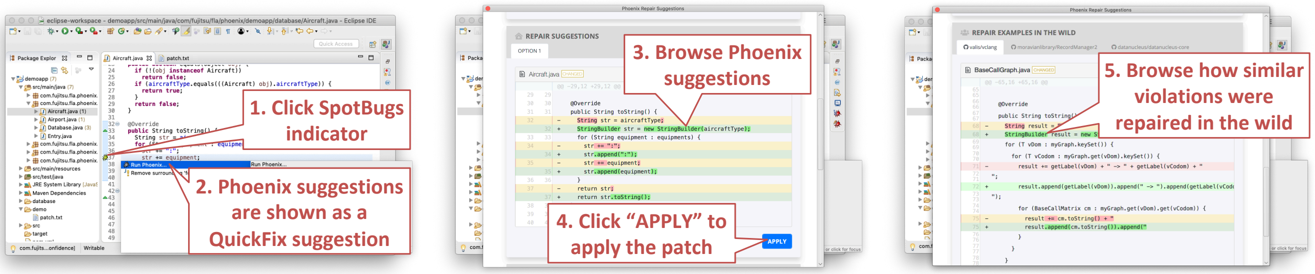
To measure the usefulness of the generated Phoenix fixes, we compute **precision** as the percentage of cases where the generated fix is *semantically equivalent* to the ground-truth i.e., the the developer's patch. However, it is possible that the developer's changes inadvertently removed the violation, thus rendering the ground truth itself inaccurate. Therefore, we separately track the cases where the patch generated by Phoenix is *correct* i.e., that it fixes the violation in the right way for that instance.

The two precision metrics described above are difficult to compute automatically. Therefore we recruited eight researchers *outside* the group of authors to perform a manual evaluation on a sample of 465 (~ 10%) mined patches (proportionately sampling each bug category). For each buggy instance, the participants are presented with the top-5 ranked patches generated by Phoenix and asked to mark the highest-ranked patch, *if any*, that (a) is a *semantically equivalent* fix, and (b) the one that is a *correct* fix. Each bug instance receives *three* independent reviews. We report the majority outcome, or no patch, in case of no majority consensus. As shown in Table 2, 54% of the top-ranked patches are judged *correct*. Given that the only oracle we use is the static analyzer, the correctness metric is more suitable to evaluate Phoenix. Overall, the results suggest that Phoenix indeed learns generic, useful patterns.

Fixing Open Violations. We applied Phoenix to repair open `SpotBugs` violations on *five* large, popular open-source Java projects, *outside* our list of mined repositories, namely *Apache Camel, Flink, Dubbo*, *Spring-Boot* and *Presto-DB*. Phoenix generated patches for 118 violations, out of a total of 141 flagged by `SpotBugs`, 94 of which were manually verified as correct. After applying some common sense filtering criteria (described in [4]), Phoenix-generated patches for 19 bugs were submitted to the respective projects, all of which have been reviewed and merged by the development teams. This study demonstrates the effectiveness of Phoenix in repairing unseen violations in the wild.

6 TOOL DEPLOYMENT

To demonstrate the effectiveness of Phoenix in industrial setting, it was deployed on a real-world financial transaction processing system, comprised of approximately 200 thousand lines of Java code. First, SpotBugs was used to detect all violations in the target


1. Click SpotBugs indicator
2. Phoenix suggestions are shown as a QuickFix suggestion
3. Browse Phoenix suggestions
4. Click "APPLY" to apply the patch
5. Browse how similar violations were repaired in the wild

(a) Launch Phoenix (b) Browse repair suggestions (c) Browse repair examples in the wild

Figure 5: Phoenix Eclipse plugin.

system. Then, the CLI version of Phoenix was applied and successfully generated repair suggestions for 52.7% of the detected violations. Among them, 95.3% of the suggestions were confirmed as valid through manual inspection. In addition, according to the estimates by the stakeholders, the repair time could be shortened by up to 30% compared to traditional manual bug-fixing. Note that the recall and precision numbers are significantly different from those of Section 5 mainly because only repair strategies that were shown to generate correct repairs according to our previous manual inspection (Section 5) were activated in this deployment.

7 RELATED WORK

Static-analysis-based patching. Some tools like FindBugs and Clang provide "quick fix" suggestions to help developers fix certain classes of bugs [3], as do IDEs like IntelliJ, Eclipse, and Visual Studio. However, these recommendations are instantiated from statically defined bug-fix strategies, while Phoenix automatically learns these strategies from examples. FootPatch [18] proposed a static-analysis-based technique for automatically generating patches for violations of heap properties, flagged by static analysis. However, while FootPatch's patch generation is tied to a specific class of violations, Phoenix's approach can target arbitrary violation classes, by learning from their examples. Liu et al. manually extract fix-patterns from a corpus of fixed static analysis (FindBug) violations mined from GitHub, and apply them to manually fix open FindBugs violations [13]. Phoenix proposes a novel, fully-automated PbE-based approach for this problem setting. Getafix [2] uses anti-unification techniques to automatically infer repair templates, similar to [13], and a special patch-ranking technique based on past human-fixes. We believe this ranking function can augment and improve Phoenix's final patch ranking.

Automated program repair (APR). A typical APR technique explores a space S of possible mutations to a buggy program P, for one that allows the mutated program to pass all tests in a given test suite T. This is done either using an explicit search [10] or by synthesizing repairs from an oracular representation derived through symbolic execution [16]. Phoenix uses a fundamentally different mechanism of constructing repairs by synthesizing a repair strategy in a custom DSL from a few examples of the same kind of bug. The DSL is designed around using a static analyzer as the oracle. These key features allow it to construct fairly sophisticated, non-localized repairs while current APR techniques can typically only create one-line, or at most one-hunk, repairs.

A more complete discussion of related work can be found in [4].

8 CONCLUSION

In our previous work [4], we proposed a novel technique for automatically generating high-quality repairs for static analysis violations by learning from repair examples. Our approach benefits from the recent success of static analyzers to completely avoid the use of test suites as a specification. This paper presented our tool Phoenix that implements a fully-automated pipeline that mines and cleans patches for violations from the wild, learns generalized executable repair strategies as programs in a novel DSL, and then instantiates concrete repairs from them on new unseen violations. We also described the architecture, user interfaces and practical usage scenarios of Phoenix. The encouraging results from both the evaluation studies and a real-world tool deployment confirm that Phoenix can actually improve the quality of software while reducing debugging and maintenance efforts.

REFERENCES

[1] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software '08* (2008).
[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. OOPSLA '19 (2019), 159:1–159:27.
[3] T. Barik et al. 2016. From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration. In *ICSME '16*.
[4] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated Data-driven Synthesis of Repairs for Static Analysis Violations. In *ESEC/FSE '19*. 613–624. http://doi.acm.org/10.1145/3338906.3338952
[5] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NFM '11*. 459–465.
[6] DARPA. 2014. Mining big code to improve software reliability and construction. https://www.darpa.mil/news-events/2014-03-06a.
[7] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, et al. 2014. Fine-grained and Accurate Source Code Differencing. In *ASE '14*. 313–324.
[8] L. Gazzola, D. Micucci, and L. Mariani. 2018. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* (2018), 1–1.
[9] Google. 2017. Error Prone. https://errorprone.info/.
[10] Claire Le Goues et al. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *ICSE '12*. 3–13.
[11] Eclipse IDE. 2018. Java Editor Quickfix. https://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.user/reference/ref-java-editor-quickfix.htm.
[12] JetBrains. 2018. IntelliJ Quick Fixes. https://www.jetbrains.com/resharper/features/quick_fixes.html.
[13] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon. 2018. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* (2018).
[14] Microsoft. 2018. Visual Studio - Common Quick Actions. https://docs.microsoft.com/en-us/visualstudio/ide/common-quick-actions.
[15] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1, Article 17 (Jan. 2018), 24 pages.
[16] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *ICSE '13*. 772–781.
[17] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *OOPSLA 2015*.
[18] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *ICSE '18*. 151–162.