PHOENIX: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations

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

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

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

ABSTRACT

Traditional automatic program repair (APR) tools rely on a test-suite as a repair specification. But test suites even when available are not of specification quality, limiting the performance and hence viability of test-suite based repair. On the other hand, static analysis-based bug finding tools are seeing increasing adoption in industry but still face challenges since the reported violations are viewed as not easily actionable. We propose a novel solution that solves both these challenges through a technique for automatically generating high-quality patches for static analysis violations by learning from examples. Our approach uses the static analyzer as an oracle and does not require a test suite. We realize our solution in a system, PHOENIX, that implements a fully-automated pipeline that mines and cleans patches for static analysis violations from the wild, learns generalized executable repair strategies as programs in a novel Domain Specific Language (DSL), and then instantiates concrete repairs from them on new unseen violations. Using PHOENIX we mine a corpus of 5,389 unique violations and patches from 517 Github projects. In a cross-validation study on this corpus PHOENIX successfully produced 4,596 bug-fixes, with a recall of 85% and a precision of 54%. When applied to the latest revisions of a further 5 Github projects, PHOENIX produced 94 correct patches to previously unknown bugs, 19 of which have already been accepted and merged by the development teams. To the best of our knowledge this constitutes, by far the largest application of any automatic patch generation technology to large-scale real-world systems.

CCS CONCEPTS

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

KEYWORDS

program synthesis, program repair, static analysis, programming-by-example

*This work was done when the author was an intern at Fujitsu Labs. of America.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338952

ACM Reference Format:
Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. PHOENIX: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338952

1 INTRODUCTION

Software debugging and patching is a time-consuming and labor-intensive aspect of the software development process, for which an automated solution is highly desirable. This has motivated a significant body of research on *automatic program repair (APR)* [12, 36]. Typical APR techniques use a test suite as a repair specification, *i.e.,* an oracle, that fails on the buggy program, and which the correct patch (for the bug) should satisfy. However, such specification-quality test suites may be hard to obtain in practice. Indeed, recent research has demonstrated that APR using real-life test suites generates a substantial number of incorrect patches, over-fitted to the test suite [8, 44]. Most APR techniques are organized (implicitly or explicitly) as a search problem and consciously limit the repair search space to keep the repair process viable. This inevitably also curtails the scope of potential repairs, with state of the art APR techniques [20, 42, 49, 50] correctly fixing less than 10% of the bugs in publicly available bug datasets such as Defects4J [21]. These limitations present significant challenges to the practical adoption of test suite-based APR techniques.

A noteworthy related trend has been the increasing deployment of static-analysis-based bug finding tools in development practice. Over the last several years, Synopsys' Coverity Scan has identified over 1.1 million defects in over 4,600 active open source software (OSS) projects, with more than 600,000 of these defects actually fixed by developers [45]. This trend is mirrored in industry, as companies such as Google [41], Facebook [6], and Microsoft [5] have actively integrated static analysis tools into their development flows. However, the reported bugs still need to be investigated, diagnosed, and fixed manually. In fact, recent case studies have recognized this aspect as a significant barrier to the widespread adoption of static-analysis-based bug finding [41]. In the sequel we use the term static analyzer (SA) as shorthand for a static-analysis-based bug finding tool.

This landscape has motivated some interesting solutions to *practical* program repair, backed by static analysis. Development environments such as IntelliJ [19], Eclipse [18], and Visual Studio [35] now offer a *"quick fix"* recommendation feature to fix simple programming errors automatically. However, these extensions work off a manually defined set of fix templates and therefore cannot be (automatically) enriched with past bug-fixing experience and/or

customize their bug-fixing to a particular development setting. Further, these tools seem to be unable to generate more comprehensive fixes such as the ones in Figure 1. In recent work, the FootPatch tool [47] proposed the automatic detection and repair of heap properties using static analysis. However, its patch generation is specifically developed for the given class of bugs and non-trivial to extend to other arbitrary bug classes.

Our approach. In this work we propose a learning-based static program repair approach that is comprised of two key elements: (1) it uses an off-the-shelf static analyzer, such as FindBugs [2], Infer [6], or error-prone [13], as an oracle, to identify potential bugs and also to certify a patch as a viable fix for such a bug, and (2) it learns repair strategies from real patches mined from Big Code. This eliminates dependence on a test-suite while leveraging the now widely used static analyzers. The patch generation nicely complements the SA's bug detection, increasing the usability of both technologies. 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 strategies very specific to each bug type reported by the SA tool, potentially increasing repair accuracy.

In order to learn repair strategies from example patches our approach uses a specially-crafted DSL to describe the strategies as programs and a synthesis algorithm to generate the programs. A key distinguishing feature of our approach is that it views each repair as a set of *related edits* emanating from a root cause. It identifies this root cause as a *primary node* in the abstract syntax tree (AST) of the program and describes each of the edits relative to this primary node, using a combination of syntactic and semantic references. Crucially, such a primary node is often directly implicated in the SA violation itself or easily identified from it. This view of repair integrates well with our use-case of repairing SA violations, while giving our approach its expressiveness and generalization ability.

Recently, Liu et al. [30] published an extensive case study also targeting the use case of deriving generic fix-patterns to repair SA violations from open-source projects. However their underlying process is entirely *manual* wherein the authors manually sift through a corpus of mined patches that are clustered using CNNs and try to manually derive fix-patterns that may repair other instances of the same violation. The patterns learned also need to be applied manually. Phoenix addresses the significant technical challenge of automating this entire process.

Our approach falls in the category of Programming by Example (PbE) techniques [29], a sub-field of inductive programming. The objective of PbE is to synthesize a program from an incomplete specification consisting of a (often small) set of input-output examples. PbE was popularized by the FlashFill work [15], and thereafter applied in a variety of domains such as data extraction from text files [25], interactive parser synthesis [28], table transformations [11], synthesizing SQL queries [48] *etc.* The PbE technique closest to our work is Refazer [40] which aims to learn from repetitive code edits and re-apply them in future contexts. The key difference between Refazer and our approach is that while Refazer characterizes (and learns) changes as a set of independent edits, each characterized by its own code context, we learn each patch as edits syntactically and semantically related to a primary node. As shown in Section 4.7 (RQ3) this structure is essential to learning from and correctly reproducing sophisticated patches, for instance the one in Figure 1.

We realize our patch-learning and recommendation solution for SA violations in a system called Phoenix, that includes a fully-automated pipeline that mines and cleans patches for static analysis violations from the wild. Using this mining component we assemble a corpus of 5,389 unique violations and patches from 517 GitHub projects. In a cross-validation study on this corpus Phoenix successfully produced 4382 bug-fixes, with a recall of 85% and a projected precision (the ground truth manually computed on a 10% random sample of the patches) of 54%. We further applied Phoenix to latest revisions of another 5 GitHub projects, where Phoenix produced 94 patches to previously unknown bugs, 19 of which have already been accepted and merged by the development teams. To the best of our knowledge this constitutes, by far the largest application of any automatic patch generation generation technology to large-scale real-world software systems.

The main contributions of this paper are as follows:

- **Technique:** A technique for synthesizing bug-fixes for static analysis violations by learning repair strategies from a few examples, using a novel DSL-based synthesis algorithm.
- **System:** A fully-automated pipeline called Phoenix, for mining and cleaning patches from the wild, as well as learning repair strategies and recommending patches for new violations.
- **Cross-validation study:** A large-scale cross validation study of Phoenix on 5,389 unique violations and patches mined from 517 GitHub projects, along with a user-study of 465 patches generated by Phoenix to quantify the quality of its patch generation.
- **Study on open violations:** A study of fixing unknown bugs from a separate set of 5 GitHub projects.
- **Dataset:** The complete corpus of patches, mined from GitHub, as well as generated by Phoenix, along with the (Phoenix-synthesized) repair strategy used to produce each patch (available at https://figshare.com/s/8ba50b84deee6a826ced).

2 MOTIVATING EXAMPLE

In this section, we motivate our technique using the example in Figure 1. Consider the program in Figure 1a, extracted from `Atomix`, a distributed-systems framework. `FindBugs` reports the violation `WMI_WRONG_MAP_ITERATOR` (described in Figure 1f) at line 9 in Figure 1a (marked by ⊗). It flags the use of the `keySet` iterator in line 1, and the subsequent calls to `get` in lines 9-10, where this additional lookup could be eliminated by using a (key, value) iterator given by `entrySet`. All the relevant fragments are enclosed in a black box.

Figure 1b shows a patch in diff form, derived from commit `0e0f94` in the same project. The high-level repair strategy used in the patch involves four inter-dependent steps, as shown below. The portion of code involved in each step is highlighted with a separate color in both the examples. The line numbers are with respect to figure 1b.

- Change `keySet` to `entrySet` (lines 1-2,)
- Introduce a new iterator variable with a parametrized type `Map.Entry<keyType, valueType>`. (lines 1-2,)
- Revive the original iterator variable at the beginning of the loop body (line 3,)
- Replace all calls to `get` on the map variable with a call to `getValue` on the new iterator (lines 8-11,)

```
1 for (Long segId: segs.keySet()) {
2   Map.Entry<Long,JournalSeg<E>> prevEntry =
        segs.floorEntry(segId-1);
3
4
5   if (!prevEntry != null) {
6     JournalSeg<E> prev = prevEntry.getValue();
7
8
9 ⊗   if (prev.lastIndex() != segs.get(segId).index() - 1) {
10      log.warn("Found misaligned seg {}", segs.get(segId));
11      segs.remove(segId);
12    }
13  }
```

(a) Example 1 (Buggy)

```
- for (Long segId: segs.keySet()) {
+ for (Map.Entry<Long, JournalSeg<E>> entry: segs.entrySet()) {
+   Long segId = entry.getKey();
  Map.Entry<Long,JournalSeg<E>> prevEntry = segs.floorEntry(segId-1);
  if (!prevEntry != null) {
    JournalSeg<E> prev = prevEntry.getValue();

- ⊗ if (prev.lastIndex() != segs.get(segId).index() - 1) {
+ ⊗ if (prev.lastIndex() != entry.getValue().index() - 1) {
-     log.warn("Found misaligned seg {}", segs.get(segId));
+     log.warn("Found misaligned seg {}", entry.getValue());
      segs.remove(segId);
```

(b) Patch Example 1 (Diff)

```
1 RepairStrategy("WMI_WRONG_MAP_ITERATOR", rule) where
2 rule = ApplyFixes(pNode, fix1, fix2, fix3, fix4)
3
4 // Primary Node Filter
5 pNode = N = MatchContext(pCtx, SAnalyzerReport())
6 pCtx = (id.MethodName == get)
7
8 // Get Edit Locations
9 eLoc1 = Visit(N.arg(0).decl().parent().body)
10 eLoc2 = Visit(N.caller().decl().deref().parent()((id.methodName == keySet) &&
11                                                   (id.enclosingLoop.containsSrcLine))
12 eLoc3 = Visit(N.arg(0).decl())
13 eLoc4 = Visit(N.caller().decl().deref().parent()((id.methodName == get) &&
14                                                   (id.enclosingLoop.containsSrcLine))
15 // Apply Edits
16 fix1 = Map(L -> Edit(L, op1), eLoc1)
17 op1 = Insert(0, CNode(VarDeclStmt, N.arg(0).inferredType,
18                                     N.arg(0),
19                                     CNode(Call, CNode(Name, newvar),
20                                                 CNode(Name, getKey))))
21
22 fix2 = Map(L -> Edit(L, op2), eLoc2)
23 op2 = Replace(CNode(Call, L.caller(), CNode(Name, entrySet)))
24 fix3 = Map(L -> Edit(L, op3), eLoc3)
25 op3 = Replace(CNode(VarDecl, CNode(ParameterizedType,
26                                     CNode(Type, CNode(Name, Map.Entry)),
27                                     L.type, N.inferredType),
28                               CNode(Name, newvar)))
29 fix4 = Map(L -> Edit(L, op4), eLoc4)
30 op4 = Replace(CNode(Call, CNode(Name, newvar), CNode(Name, getValue)))
```

(c) Repair Strategy learnt by PHOENIX

```
- for (String url: parsedLinks.keySet()) {
+ for (Map.Entry<String,
+         Map<String, String>> e: parsedLinks.entrySet()) {
+   String url = e.getKey();
- ⊗ RiakLink link = parseOneLink(url, parsedLinks.get(url));
+ ⊗ RiakLink link = parseOneLink(url, e.getValue());
    if (link != null) {
```

(d) Patch Example 2 (Diff)

```
- for (String name: insQueries.keySet()) {
+ for (Map.Entry<String,
+                 String> newvar: insQueries.entrySet()) {
+   String name = newvar.getKey();
- ⊗ String sqlQuery = insQueries.get(name);
+ ⊗ String sqlQuery = newvar.getValue();
    List<List<Object>> params = insParams.get(name);
```

(e) Patched Violation (Diff)

WMI_WRONG_MAP_ITERATOR
Inefficient use of keySet iterator instead of entrySet iterator. This method accesses the value of a Map entry, using a key that was retrieved from a keySet iterator. It is more efficient to use an iterator on the entrySet of the map, to avoid the Map.get(key) lookup.

(f) Findbugs Description for Figure 1

Figure 1: Motivating Example

Figure 1d shows another patch for the same violation that uses the same strategy, extracted from commit `826021` in the `Riak-Java-Client` repository. Automatically learning such a strategy from these two patches involves several challenges. Firstly, `FindBugs` only flags the line containing the call to `get` (marked as ⊗), not all the locations that require an edit. Secondly, the locations are inter-dependent – the strategy should not apply edits at irrelevant locations. For example, not all method calls on `segs` need to be replaced by `getValue` (line 4 in figure 1b). Finally, the required edits are not *local* in the sense that the AST node needing the edit (highlighted in colors) does not contain all the information required to perform the edit. For example, the introduction of the new `entrySet` iterator (line 2 in figure 1b) requires a type that needs to be inferred from the map being iterated on. Such information may not even be present syntactically in the source AST if the map is acquired from a class in a separate file.

PHOENIX automatically learns this strategy from the two patches in 1b and 1d, expressed in its internal DSL as shown in figure 1c. The color codes indicate the code fragments in the examples catered to by the corresponding component in the strategy. The strategy involves three stages as given below. All lines referred to are with respect to figure 1c unless otherwise stated.

Stage 1 (Lines 5-6) : Given the report by the static analyzer, find a *primary node* – an AST node that serves an anchor for locating and applying edit operations. In our example, the primary nodes are the AST nodes corresponding to the code fragments highlighted with a red frame (lines 8, 5 in figures 1b, 1d resp.). The node is found by applying a filter to all AST nodes potentially flagged by the analyzer. In this case, the filter outputs all invocations with method-name as `get`, included in the line flagged by the analyzer.

Stage 2 (Lines 9-14) : Find all the edit locations *relative* to the primary node via AST visitors. Here **N** stands for the primary node.

For example, the visitor in line 12 finds the iterator node to be replaced by visiting the first argument of the method-call corresponding to the primary node (`name`), and then its declaration (in the for-loop). Since visitors can visit multiple nodes, they may also have a context-matcher. For example, the visitor in lines 13-14 goes to all the dereferences of the map, and finally its parent. Since there may be multiple uses of the map in various contexts, the matcher zeroes in on the uses where parent is a call to `get`, and the enclosing loop contains the line flagged by `Findbugs`. The latter condition prevents any replacements of `get` by `getValue` outside the loop.

Stage 3 (Lines 16-30) : Apply edits at all the locations. Edits can be inserts, replacements or deletions. `CNode` indicates a concrete AST node and leaf nodes are in **bold**. Recall that many edits are non-local - they solicit AST elements from elsewhere as well. To represent such edits, the nodes in the edit can be *referential*. For example, in lines 25-28, the parameterized type is constructed using *visitors* `L.type` and `N.inferredType`, which access the declared type of the original iterator and the return type of the call to `get` respectively.

Figure 1e shows the result of applying this strategy to an unseen violation in the `Dari` repository. The strategy finds only one candidate for the primary node - the call to `get` in line 5 enclosed in a (red frame). It then finds all the edit locations and applies edits relative to this node. The strategy fixes the violation correctly, only needing the user to suggest a name for the **newvar** variable.

Refazer is not able to learn such a strategy due to its simpler AST pattern-matching component. It is able to learn the raw edit operations in this case, but over-generalizes and applies edits at irrelevant locations such as replacing the `get` at line 7 in figure 1e. Phoenix learns to avoid this by only going over the dereferences of the map in question (lines 13-14 in Figure 1c). However, Refazer was targeted towards the domain of learning repetitive, purely syntactical code edits, where it works quite well. Phoenix additionally targets the cases where there are multiple edits, each interlinked with the other. This is made possible by our novel master-slave approach of representing edits, wherein the primary node serves as the reference point for all edits.

This patch is also out-of-scope for current Automated Program Repair (APR) techniques as they typically suggest one-line or at most one-hunk repairs. This is largely due to the rather broad transformation schemas that are specified/learned to target the much larger class of functional repairs, where such a patch cannot be generated due to the subsequent combinatorial explosion of repair candidates. Focusing on static analysis violations that often have standard repair strategies allows Phoenix to produce fairly sophisticated and non-localized repairs such as the one in Figure 1e.

3 APPROACH

3.1 Overview

Figure 2 shows the high-level overview of Phoenix. There are three main stages - *mining* patches for static-analysis violations, *learning* repair strategies from the mined patches, and *fixing* open violations.

For mining patches, Phoenix collects open-source Java repositories, and runs the static-analyzer (`FindBugs`) on each commit to obtain a list of static-analysis violations for each analyzed commit. Phoenix then applies a violation-tracking procedure to obtain the violations that were fixed as well as the corresponding patch that fixed them (Section 3.2). This yields a collection of patches, on which it further applies a cleaning procedure (Section 3.3) to filter out edits that are irrelevant to the removal of the violation. These cleaned patches constitute a database of patches.

Phoenix then uses a novel synthesis algorithm (Section 3.5) to learn generic, high-level repair strategies in a special DSL (Section 3.4). Finally, these strategies can now be applied to unseen, buggy source code to generate fix suggestions (Section 3.6).

3.2 Collecting Patch Examples

For collecting examples of static analysis violation fixes, we use exactly the same approach developed in Liu et al.'s work [30]. We first collect Java projects with at least 500 commits, run the static analyzer (Findbugs) and collect violations for all revisions. Note that there are occasional bad commits that leave the code uncompilable or unanalyzable. We ignore such revisions.

Then for every consecutive pair of analyzed revisions, we apply the violation tracking procedure in [1] to match up violations to identify the ones that have disappeared. This is achieved by certain heuristics applied on top of the output of a diffing algorithm [37]. The violations that have disappeared are either because of an edit in the source file (fixed), or because of external reasons. We are interested in the former, as the edits essentially represent a patch for this violation. We collect all such fixed violations to form our patch database. We refer the reader to [1, 30] for further details.

3.3 Cleaning Patches

The patches obtained from Section 3.2 are not quite useful by themselves. This is because they often contain edits irrelevant to the fixing of the violation, these can severely affect the quality of patterns generated by our learning algorithm. Hence we employ a patch-cleaning procedure to remove these irrelevant edits.

The key idea is that we can model this problem as failing test-case minimization problem, where a test-case corresponds to the application of an edit, followed by a run against the oracle (static analyzer). The test-case *fails* if the violation *goes away*. Thus the minimal test-case would correspond to the minimal set of edits required to remove the violation. We can then use a delta-debugging approach [54] to solve this problem instance as follows. We first try to form minimal clusters of edits such that each cluster, when applied to the source, is compilable. This can be done using dependency analysis with the help of the compiler. We then use the delta-debugging algorithm as is, to obtain the minimal set of such clusters the together represent the minimal set of edits that when applied to the source, removes the violation in question.

3.4 Domain Specific Language

We now describe our DSL for representing repair strategies and explain the rationale behind the main components. Figure 3 gives a formal description of our language and its syntax.

A Repair Strategy $\mathcal{S}$ consists of a name, and a *rule* which describes the transformation to be applied. A *rule* consists of a *context-matcher* to locate an AST node we call the *primary node*, and a list $\mathcal{L}$ of fixes. A fix $\mathcal{F}$ in $\mathcal{L}$ is mapper that applies edit operations to a set of locations (AST nodes) obtained by traversing the AST, starting from the primary node through a *visitor*. An edit operation is one of three AST-based operations - inserts, deletes and replacements.

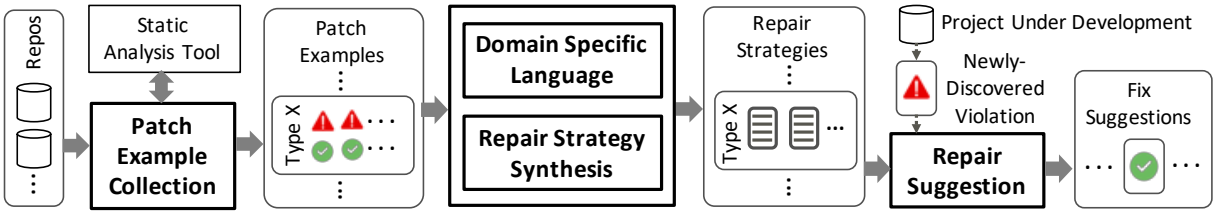


Figure 2: Overview of PHOENIX system.

$$
\begin{aligned}
\textit{strategy} &::= \texttt{Strategy(name, } \textit{rule}\texttt{)} \\
\textit{rule} &::= \texttt{Apply(}\textit{primaryNode}\texttt{, } \textit{fix}_1\texttt{, } \ldots\texttt{, } \textit{fix}_n\texttt{)} \\
\textit{primaryNode} &::= \texttt{MatchCtx(}\textit{ctx}\texttt{, SAnalyzerReport())} \\
\textit{fix} &::= \texttt{Map(}\lambda L \rightarrow \texttt{Edit(}L\texttt{, } \textit{operation}\texttt{)}\texttt{, } \textit{editLoc}\texttt{)} \\
\textit{editLoc} &::= \texttt{Visit(}\textit{primaryNode}\texttt{, } \textit{visitor}\texttt{)} \\
\\
\textit{visitor} &::= \textit{astpath}(\textit{ctx}) \\
\textit{ctx} &::= \textit{astpath}(\textit{pred}) \mid \textit{ctx}_1 \wedge \textit{ctx}_2 \\
\textit{astpath} &::= \textit{vpath}_1 \circ \textit{vpath}_2 \\
&\quad \mid \texttt{N} \mid \texttt{L} \mid \texttt{id} \mid \texttt{child(i)} \mid \texttt{parent} \mid \texttt{decl} \mid \texttt{use} \mid \ldots \\
\textit{pred} &::= \texttt{true} \mid \text{key} = \text{value} \mid \text{key} \neq \text{value} \\
\textit{key} &::= \texttt{nodeType} \mid \texttt{methodName} \mid \texttt{containsSrcLine} \mid \ldots \\
\\
\textit{operation} &::= \texttt{Insert(pos, } \textit{ast}\texttt{)} \mid \texttt{Delete()} \mid \texttt{Replace(}\textit{ast}\texttt{)} \\
\textit{ast} &::= \textit{const} \mid \textit{ref} \\
\textit{const} &::= \texttt{ConstNode(kind, value, } \textit{ast}_1\texttt{, } \ldots\texttt{, } \textit{ast}_n\texttt{)} \\
\textit{ref} &::= \texttt{Visit(}\textit{editLoc}\texttt{, } \textit{visitor}\texttt{)} \\
&\quad \mid \texttt{Visit(}\textit{primaryNode}\texttt{, } \textit{visitor}\texttt{)}
\end{aligned}
$$

Figure 3: Our DSL for representing Repair Strategies

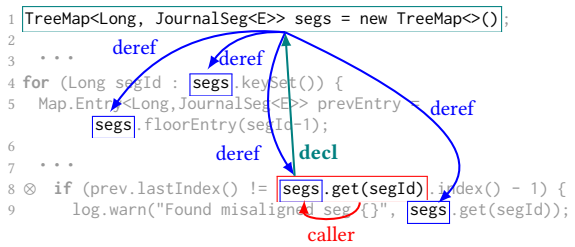


Figure 4: Examples of High-Level AST Edges

Primary Nodes : A primary node $\mathcal{P}$ is an AST node that can be considered as the root cause of the static-analysis violation. For example, the primary node in figure 1a is `segs.get(segId)`. The primary node serves as an anchor point for finding all the necessary repair ingredients such as the edit locations, and AST nodes for replacement/insertion. This is similar to how developers approach such problems. They consult the static analysis report, find the root cause, and devise a *mental strategy* for fixing the violation starting from that point.

In general, this node should be considered as an additional input provided by the static-analyzer, along with the patch (when learning) or the buggy input (when applying). In some cases, such as ours, this input is unavailable as the analyzer only works on byte-code. Hence, we include a context-matcher to further filter out the primary node amongst multiple candidates if any.

Context-Matchers : A Context-matcher C is a boolean function that checks the properties of the surrounding nodes of a target node (including itself). The predicates in C (`pred` in figure 3) check for equality or inequality of key-value pairs defined individually for each AST node. For example, the key-value pairs for a `MethodInvocation` node can be the node type, the declaring class, the return type, type of parameters, method name, etc. Surrounding nodes are visited using AST edges (child, parent, etc.) that are described in the next paragraph. An example of C is `id.parent(nodeType==ForStmt)` which checks if the parent of the current node (represented by `id` is a for-statement.

Visitors : A visitor $\mathcal{V}$ consists of a path in an AST followed by a context-matcher that is applied on the target node. The edges in the patch can be structural(child, parent, etc.) as well as semantic (declaration, use, dereference, writes, type etc.). Examples of such edges for the code fragment in figure 1a are shown in figure 4. Semantic edges enable connections between AST nodes that are far-apart structurally, but share meaningful relationships with each other. Consequently, they enable better generalization across input programs as they bridge gaps between nodes that may have widely varying structural connections across two I/O examples. For example, a path from a variable to its declaration is better described by the `decl` edge, rather than simple structural edges such as `child` and `parent`.

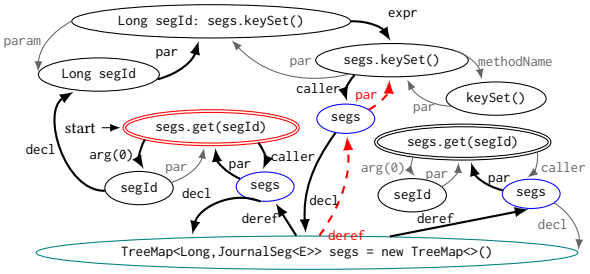
The context-matcher offers further precision if necessary in cases where multiple nodes may be visited by the same path. For example, a path containing the `deref` edge in figure 4 which connects a variable declaration to all its dereferences may need further disambiguation in its usage, as was necessary in the motivating example.

Edit Operations : An edit operation can be one of three kinds -

- `Insert(`*ast*, pos`)`- Insert *ast* at position *pos* in the current sub-tree
- `Delete()`- Delete the current sub-tree
- `Replace(`*ast*`)`- Replace the current sub-tree by *ast*

The concrete new/replacement ASTs used in the insert/replace operations respectively may be overly specific to an input. Therefore, to enable generalization, these ASTs can be a mix of concrete nodes as well as referential nodes. Referential nodes are obtained by traversing the AST starting from the edit location or the primary node using *visitors*, and selecting the node(s) to use to build the AST.

Our core contribution is the use of high-level, semantic AST edges and their use in visitors and context-matchers in defining expressive repair strategies, along with a synthesis algorithm to handle the consequent space explosion. Our novel combination of visitor paths and context-matchers enables powerful *higher-order pattern matching* wherein the pattern itself contains references to nodes in the target AST. For example, PHOENIX can learn patterns such as - "All calls to `get` where the caller is the same as the one in the invocation flagged by the analyzer" (the




```
Path 1 : caller.decl.deref.par
Path 2 : arg(0).decl.par.expr.caller.decl.deref.par
```

Figure 5: Partial NFA representing multiple paths from primary node to calls to get that need editing for I/O example in figure 1a.

reference lies in the underlined portion). This is exactly the pattern encoded in the visitor in line 13 in figure 1c and is out-of-scope for Refazer, which although employs a similar DSL to Phoenix, only supports patterns containing concrete node types or strings to match against.

3.5 Learning Repair Strategies from Examples

We now describe our algorithm for learning repair strategies. Formally, the input to Phoenix is a list $\mathcal{I} = \{(\mathcal{P}_{i_1}, \mathcal{P}_{o_1}, \mathcal{B}_1), \ldots, (\mathcal{P}_{i_n}, \mathcal{P}_{o_n}, \mathcal{B}_n)\}$ where $\mathcal{P}_i$ is the buggy input, $\mathcal{P}_o$ is the patched output, and $\mathcal{B}$ is the bug-report by the static-analyzer. $\mathcal{B}$ should contain the violation type, and the line-number in the source corresponding to the buggy location. Phoenix then learns a set $\mathcal{R} = \{r_1, \ldots, r_k\}$ of repair strategies that collectively covers the transformations required by the all the examples in the input, with an aim to minimize k.

The synthesis algorithm in Phoenix goes through two broad stages - (1) Compute and represent all possible strategies for all I/O examples individually and (2) Pick the fewest number of strategies amongst these that can solve all the I/O examples. Figure 6 formally describes all the steps involved. We start from the procedure Synth.

Locating Primary Nodes and Constructing Edits : In line 3, we locate the primary node candidates based on the analysis report for each input $(\mathcal{P}_i, \mathcal{P}_o, \mathcal{B})$ tuple. We then instantiate a training example for Phoenix for each primary node individually along with the edits involved in the transformation described by $\mathcal{P}_i$ and $\mathcal{P}_o$. An edit contains a visitor from the primary node to its location, and the operation itself, which in case of replacements and inserts, can contain visitors to nodes. GetEdits describes this construction.

At this point, we need to store *all possible visitors* as we are not certain which one would generalize best across all examples (line 33). As there can be arbitrarily many, we represent the collection compactly as an NFA where states are AST nodes, the initial state is the source node, and the accepting states are the destination nodes. An example is shown in figure 5 which encodes paths from the primary node to all the invocations of `get` that need to be replaced in figure 1a in the motivating example. It contains paths apart from the visitor used in line 13 in figure 1c. The dark, bold edges are used in the paths, while the red dashes demonstrate how a path can lead to a non-accepting state. These NFAs are similar in spirit to the VSAs used in [15] to represent all possible programs.

In order to compute common paths between two sets of visitor paths, we convert these NFAs to DFAs using the standard construction [17]. However, such a conversion may be too expensive given the large number of nodes and edges in the AST. However the key insight is that truly useful visitors are not usually very long - therefore, during the construction, we only add sets of states that are reachable within a `maxLen` (we use 8) number of steps (line 47).

We add references to the edit operations in a similar fashion (lines 38, 40-43). If the involved edit is an insert or a replace, we go over all the nodes in the subtree of the node being added, and store a reference in the form of a DFA containing all paths from the primary node or the edit location to all the *syntactic matches* of that node in the same AST. This is the step that allows our strategies to fetch repair ingredients from elsewhere in the AST. This construction concludes the first stage of the algorithm.

Clustering Edits and Training Examples : We now generalize edits across multiple training examples. Edits are grouped together using a greedy pair-wise clustering algorithm (line 7). At each iteration, we select the pair of edits with the highest non-zero *compatibility score*, and *combine* them into one. The procedure stops when all pairs are incompatible (score is zero).

Two edits are considered compatible (lines 48-51), if they have a common path from the primary node *i.e.,* their location visitor DFAs have a non-empty intersection, and the AST components (if any) are either equal, or the unequal nodes share a reference *i.e.,* their corresponding DFAs have common strings. The score captures the degree of similarity - sharing more number of paths, and more concrete AST nodes leads to a higher score.

Once the edits are clustered, the training examples are grouped together if they share the same edits (line 8). Training examples that are grouped together share the same repair strategy.

Assembly : For each group of clustered training examples, we extract the best visitors from the DFAs in each edit by treating the DFAs as graphs and employing Dijkstra's algorithm [9] with the cost metric as a function of the length, the edges involved, and the context-matchers required, if any, in the visitor (lines 28-32). In essence, visitors that are short, use semantic edges, and employ none or small context matchers are preferred.

Finally, we get all the primary nodes, and all the other nodes which also belong to the region flagged by the analyzer, and synthesize a context-matcher for disambiguating the primary nodes.

Synthesis of Context-Matchers : Recall that a context-matcher is a conjunction of positive and negative predicates which use attributes of AST nodes to match against a node. Its synthesis (lines 60-63) involves taking as input a set of *positive* and *negative* examples (nodes). The desired output is a conjunction of predicates that are true for the positive but false for the negative examples.

The problem can be reduced to finding the least-cost subset of the predicates that together evaluate to false for all the negative examples, and thus can be modeled as a set-cover problem for which we use the $log(n)$-optimal greedy approximate algorithm [7].

3.6 Suggesting Fixes

Applying the learned strategies is relatively straightforward. Given a learned strategy, the buggy source, and the corresponding static-analysis report, we first find k primary node candidates using the

```
SYNTH (I ≡{(P_i1, P_o1, R_1), ..., (P_in, P_on, R_n)})
trainingSet ← empty list
for each (P_i, P_o, R) ∈ I do
    pNodeCands ← GETPRIMARYNODECANDS(P_i, R)
    for each pNode ∈ pNodeCands do
        edits ← GETEDITS(P_i, P_o, pNode)
        trainingSet.append((P_i, pNode, edits))
editClusters ← CLUSTEREDITS(trainingSet)
clusters ← CLUSTERTRAININGSET(trainingSet, editClusters)
for each cluster ∈ clusters do
    edits ← GETALLEDITS(cluster, editClusters)
    FINALIZEEDITS(edits)
    pNodes, nonPNodes ← PARTITIONNODES(cluster)
    primCtx ← SYNTHESIZECTX(pNodes, nonPNodes)
    output ASSEMBLESTRATEGY(primCtx, edits)

CLUSTEREDITS (trainingSet)
worklist ← all edits in all points in the trainingSet
e_1, e_2 ← GETMOSTCOMPATIBLEPAIR(worklist)
while e_1 is valid ∧ e_2 is valid do
    COMBINE(e_1, e_2)
    worklist ← worklist − e_2
    e_1, e_2 ← GETMOSTCOMPATIBLEPAIR(worklist)
return worklist

COMBINE (e_1, e_2)
e_1.locDFA ← DFAINTERSECT(e_1.locDFA, e_2.locDFA, maxLen)
if e_1 and e_2 are Inserts or Replacements then
    COMBINEASTS(e_1.ast, e_2.ast)

COMBINEASTS (a_1, a_2)
a_1.ref ← DFAINTERSECT(a_1.ref, a_2.ref, maxLen)
for each child_1, child_2 of a_1, a_2 do
    COMBINEASTS(child_1, child_2)

FINALIZEEDITS (edits)
costMetric = λv → F(cost of edges in v, cost of ctx-matcher of v, len(v))
for each edit ∈ edits do
    edit.loc = RUNDIJKSTRA(edit.locDFA, costMetric)
    for each node n with refs in edit.ast do
        n.ref = RUNDIJKSTRA(n.ref, costMetric)

GETEDITS (P_i, P_o, pNode)
srcAst ← GETAUGMENTEDAST(P_i)
for each edit ∈ ASTDIFF(P_i, P_o) do
    loc ← GETROOTLOCATION(rawEdit)
    locDFA ← GETALLPATHS(pNode, loc, srcAst, maxLen)
    if edit is Insert or Replace then
        ADDREFERENCES(edit, pNode, loc, srcAst)
return all collected edits

ADDREFERENCES (edit, pNode, loc, srcAst)
for each node ∈ GETREPLACEMENTORINSERTEDNODE(edit) do
    matches ← GETASTMATCHES(node, srcAst)
    if matches ≠ φ then
        node.ref ← GETALLPATHS({pNode, loc}, matches, srcAst)

GETALLPATHS (srcs, dsts, augmentedAst)
nfa ← augmentedAst.copy()
nfa.setInitial(srcs)
nfa.setAccepting(dsts);
return NFA-TO-DFA(nfa, maxLen)

COMPATIBILITYSCOREEDIT (e_1, e_2)
if e_1, e_2 not of the same kind return 0
locationScore ← NUMCOMMONSTRINGS(e_1.locDFA, e_2.locDFA, maxLen)
astScore = COMPATIBILITYSCOREAST(e_1.ast.root, e_2.ast.root)
return astScore * locationScore

COMPATIBILITYSCOREAST (n_1, n_2)
score ← (type(n_1) = type(n_2)) + NUMCOMMONSTRINGS(n_1.ref, n_2.ref, maxLen)
if score = 0 return 0
if number of children of n_1 and n_2 differ return 0
for each child_1, child_2 of a_1, a_2 do
    cScore ← COMPATIBILITYSCOREAST(child_1, child_2)
    if cScore = 0 return 0
    score ← score + cScore
return score

SYNTHESIZECTX (posEx, negEx)
posPredicates ← GETPOSPREDICATES(posEx)
negPredicates ← GETNEGPREDICATES(negEx)
preds ← FILTER(λp → p(posEx), posPredicates ∪ negPredicates)
return GREEDYSETCOVER(preds, negEx)
```

Figure 6: Learning Algorithm

primary-node context-matcher. We then generate k fix candidates by considering each of them as the primary node separately. This is followed by a filtering step that removes uncompilable fixes or those that failed to remove the static-analysis violation.

The remaining suggestions are ranked based on the rank of the repair strategies employed to produce them. The rank of a repair strategy is the size of the union of the sets of input examples used to learn each of the edits contained in the strategy, with a higher value indicating a higher rank. The rationale is that more the number of examples used, better is the *support* for that strategy, which indicates a higher chance of it being useful.

4 EVALUATION

4.1 Implementation

We have implemented our patch-collection, cleaning and synthesis algorithm in a tool called PHOENIX. The patch collection component is written in Python and the cleaner and synthesizer components are written in Java. We use `GumTree` [10] to compute AST differences and employ `FindBugs` [2] as the static program analyzer.

4.2 Dataset

We use the same set of repositories as in Liu et al.'s study [30] assembled using `GHTorrent` [14]. The project should (1) have at least 500 commits, (2) have the main language as Java, (3) be an original project (*i.e.,* not forked from another project), (4) use Apache Maven, and (5) have at least two revisions in which `FindBugs` can execute. In the original study, there are 730 projects in total. We could process 517 of these in our time-budget of 2 months.

Table 1: Summary of benchmark subjects.

#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

For each project, we use the tracking procedure described in Section 3.2 to collect 21,028 patches that fix violations identified by `FindBugs`. Applying the cleaning procedure in Section 3.3 yields 11,865 patches out of which 5,512 are unique, covering 234 distinct violation types. The cleaning discards multi-file patches, since currently PHOENIX can only learn from and generate single-file patches. Also, the number of unique patches is lower as merge commits can cause duplication of patches across revision pairs. Finally we discard patches whose violation category only contains one project. This yields a final set of 5,389 patches. Table 1 contains all the statistics.

4.3 Experimental Setup

Data-collection for this paper was performed on two 64-core machines, each with Intel Xeon 2.60 GHz processors with 128GB of memory running Ubuntu 16.04 LTS 64-bit. All other experiments

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%)

were performed on an 8-core machine with Intel i7-4790 3.60GHz CPU, 16GB of memory running Ubuntu 16.04 LTS 64-bit.

4.4 Results

Our evaluation addresses the following research questions:

RQ1: How effective is our synthesis algorithm?

RQ2: Can Phoenix fix open static-analysis violations in the wild?

RQ3: How effective is Phoenix versus the current state of the art?

4.5 RQ1: Synthesis Algorithm Effectiveness

To evaluate the learning ability of Phoenix, we perform a project-level, leave-one-out cross-validation experiment on our database of 5,389 patches. 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 then compute **recall** as the percentage of cases where at least one fix is generated. Note that Phoenix *only* generates a fix if the violation is removed, as judged using `FindBugs`.

The results are shown in table Table 2. Phoenix achieves a recall of 85% indicating that it is able to generalize across projects. However, there can be multiple possible ways to fix the violation, the most trivial being deletion of the offending code fragment. Such fixes may be generated as Phoenix only checks if the violation is removed, but does not check if source semantics have changed. Hence to measure the usefulness of generated fixes, we compute **precision** by computing the percentage of cases where the generated fix is *semantically equivalent* to the ground-truth *i.e.,* the fix applied by the developer. However, static-analyzers are not used widely, and it's quite probable that the developer did not intend to fix the violation and hence the ground-truth itself may not be accurate. 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, and therefore we perform a manual evaluation by picking a sample of 465 (~ 10%) mined patches (proportionately sample each bug category retaining at least 1 instance). We recruited eight researchers *outside* the group of authors for this purpose. The participants are presented with the top-5 ranked patches generated by Phoenix for each buggy instance 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 is independently reviewed by *three* participants and the majority outcome accepted. In case of no majority view, *i.e.,* the three reviews identified different patches, no patch is reported for that instance. We find that in 35% and 44% of cases, the top-ranked and top-5 ranked patches respectively contain a semantically equivalent patch. Further, in 54% and 64% of the cases, the top-ranked and the top-5 ranked patches contain a correct patch. 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.

4.6 RQ2: Fixing Open Violations in the Wild

The true test of a tool like Phoenix is to produce patches for open, previously unseen violations that developers are willing to integrate into their code. To this end, we apply Phoenix to repair open `FindBugs` violations on *five* large, popular open-source Java projects, namely *Apache Camel, Flink, Dubbo*, *Spring-Boot* and *Presto-DB*). Note that these projects are *outside* our list of mined repositories. The results are presented in Table 3. Our procedure for applying Phoenix on these projects involves four steps - (1) We run `FindBugs` and get violations having a rank ≤ 15, priority ≤ 2 and from a category belonging to the 50 categories having the highest precision score in our manual study. (2) We apply Phoenix to generate patches for all of them (3) We manually examine each patch to check its correctness (4) We submit the true-positive, correctly-fixed instances to the development team.

The first two columns show the sizes of the projects, indicating their complexity. The *#Total* column shows the number of violations obtained by running `FindBugs` that satisfy our criteria (141) and the *#Patched* column shows the number of violations for which at least one patch was generated (118), indicating a recall of 84%. The *#Correct* column shows the number of violations for which the patch was correct (94), indicating a precision of 80%. The category-wise distribution of these correctly fixed violations is given in Table 5. We chose to ignore five categories for our final submission of patches as the respective violations were either hard to justify to developers without concrete test-cases, which are themselves difficult to obtain (specifically concurrency and serialization issues), or a large fraction of violations in these categories were indicated as false positives during our manual precision study. We then submitted the remaining 19 patches, all of which have been reviewed and merged by the core teams. These patches covered a wide range of issues including incorrect string processing, performance problems, random number generation as well as infinite loops. These results indicate that Phoenix learns effective strategies capable of repairing unseen violations in the wild.

4.7 RQ3: Comparison against State of the Art

A direct comparison against the current state-of-the-art technique for learning AST transformations, Refazer[40] is unfortunately not feasible as the target languages differ (C# vs Java) and it is highly non-trivial to instantiate, for Java, the Prose framework [39] for program synthesis that Refazer is based on. Instead, since our DSL subsumes the one introduced in Refazer, we modify it and the corresponding synthesis algorithm to approximate the expected operation of Refazer in our use case. We call the modified version Phoenix-Baseline.

In Phoenix-Baseline we remove semantic edges from the AST, eliminate visitor paths completely and only allow positive predicates in context-matchers. We also limit the key-value pairs to the node-type and concrete node value. However, for a fairer comparison, we also incorporate bug report information in the form of additional node properties indicating whether they are mentioned

Table 3: Results of fixing open violations in the wild

Project	#Files	SLOC	#Total	#Patched	#Correct	#Submitted (Merged)
Apache Camel	17,518	1.1m	61	59	49	7
Apache Flink	9,021	1.1m	32	16	11	3
Presto DB	5,174	600k	30	27	25	4
Spring Boot	4,330	250k	5	4	3	2
Apache Dubbo	1,558	110k	13	12	5	3
Total	37,601	3.2m	141	118	94	19

Table 4: Phoenix vs Phoenix-Baseline

	Precision	
	Phoenix	Phoenix-Baseline
Overall	60%	33%
Simple	50%	38%
Complex	70%	28%

Table 5: Correctly-fixed Open Violations by Bug Category

Category	Count	Submitted
JLM_JSR166_UTILCONCURRENT_MONITORENTER	12	✗
NP_BOOLEAN_RETURN_NULL	44	✗
SE_BAD_FIELD	13	✗
RpC_REPEATED_CONDITIONAL_TEST	4	✗
STCAL_INVOKE_ON_STATIC_DATE_FORMAT_INSTANCE	2	✗
DMI_INVOKING_TOSTRING_ON_ARRAY	6	6
WMI_WRONG_MAP_ITERATOR	3	3
ICAST_INT_CAST_TO_FLOAT_PASSED_TO_ROUND	2	2
RV_EXCEPTION_NOT_THROWN	1	1
IM_BAD_CHECK_FOR_ODD	1	1
RC_REF_COMPARISON	2	2
VA_FORMAT_STRING_BAD_CONVERSION_FROM_ARRAY	1	1
DMI_RANDOM_USED_ONLY_ONCE	1	1
SBSC_USE_STRINGBUFFER_CONCATENATION	1	1
IL_INFINITE_RECURSIVE_LOOP	1	1
Total	94	19

```
- String allmsgs = "";
+ StringBuilder allmsgs = new StringBuilder();
  for (int i = 0; i < msgs.length; i++) {
      msgs[i] = msgs[i].replace('\n', ' ');
-     allmsgs += msgs[i] + "\n\n";
+     allmsgs.append(msgs[i]).append("\n\n");
  }
- messagesTextPane.setText(allmsgs);
+ messagesTextPane.setText(allmsgs.toString());
```

Figure 7: Complex fix generated by Phoenix

```
-     writer.printf("%d\n", value);
+     writer.printf("%d%n", value);
```

Figure 8: Patch out of scope for Phoenix

in the report. Overall. this mimics the fairly basic pattern-matching and independent treatment of edits in Refazer. We modify the algorithm to now compute compatibility scores of edits by only tracking the number of common context-matchers between edits.

We repeat the cross-validation study for Phoenix-Baseline and the find its recall is 72%, moderately lower than Phoenix's 85%. This is because a number of violations reported by `FindBugs` involve simple, local one-line fixes. To further scrutinize patch quality, for both simple, single-line instances, as well as those with complex, inter-dependent edits we compute precision manually for Phoenix and Phoenix-Baseline on two sets of 50 patches each. The first set (Complex) is constructed by sampling 5 patches each from 10 violation types in the database which we identified as involving the most number of edits. The second set (Simple) is constructed by sampling 50 patches from the rest of the types. Table 4 presents these results. Compared to Phoenix, Phoenix-Baseline demonstrates significantly lower precision on the complex cases, and even modestly lower on simple instances. Our novel use of primary nodes as anchor points, and the powerful pattern-matching enabled by the combination of visitors and context-matchers allows Phoenix to generate much more sophisticated patches.

5 DISCUSSION

Scope of repair strategies: Phoenix is able to learn a variety of repair strategies. Figure 7 shows an example of a patch generated by Phoenix for the expensive in-loop string concatenation flagged by `FindBugs`. The patch involves multiple inter-dependent edit locations, and cannot be generated by existing work like Refazer.

However, there are certain violation types for which Phoenix currently cannot learn a useful strategy. For example, Figure 8 shows a patch for a violation which flags the use of '\n' instead of the platform-independent '%n'. This involves string operations, which are not present in the DSL. However, if the AST-diff tool produces special operations to capture these edits, it is relatively straightforward to add this capability to Phoenix.

Incorrect variable/method naming styles comprise another category of violations Phoenix is ineffective at. Since any patch would involve project-specific changes, there does not exist a generic repair-strategy that can be learned by Phoenix.

Efficiency: In the cross-validation study, Phoenix took 40s on average to learn strategies from all the examples. Further, it took 50s on average to fully process a violation and generate a ranked list of patch suggestions. The bulk of this time (85%) is spent in compilation and running `FindBugs` and only 15% to actually synthesize edits. We believe this can be significantly improved through tighter integration with the static analyzer. Overall, Phoenix presents a viable solution for real-time patch generation.

Phoenix vs. [30]: Although Phoenix and [30] both share the goal of patching static analysis violations, we believe a direct comparison between the two would not be very meaningful. This is because the *manual* inference of fix-patterns and creation of patches, performed in [30] allows them to work with fairly abstract fix-patterns like: *"Replace variable by some other variable"*. Phoenix would never learn such patterns because of the large search space of concrete instantiations they include.

6 LIMITATIONS

Repair ranking & scope. Phoenix's current implementation can only generate fixes limited to a single file, although we can generalize the learning algorithm to take sets of patches as a single input, and introduce edges across ASTs. The ranking procedure for suggestions in Phoenix is also quite rudimentary, relying solely on a single oracle (static analyzer) and the number of examples a repair strategy is learned from. Thus, a less frequently used repair strategy will always be ranked lower even if it is more appropriate in the

current context. Conversely, an unsuitable repair strategy may be ranked high if it is widely used and the static analyzer certifies a violation fix. An effective ranking procedure should rank repairs in the context of the code being patched.

Patch mining & cleaning. Phoenix's mining component is driven by heuristics, and may be imprecise, especially in violation-tracking. Its cleaning module is also quite simple, at present. This, in some cases, results in low-quality patches because of non-removal of irrelevant edits, as well as loss of good patches because no compilable set of edits could be found. We plan to fix this in the future by using better program analyses.

Construct validity. Our calculation of precision of patches has a subjective component due to reliance on manual evaluation. In particular, judging whether a patch is semantically equivalent or correct requires manual inspection which may be imprecise. We try to minimize the effect of this issue by having three reviewers independently study each patch, and reach consensus by using the majority view.

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 [19], Eclipse [18], and Visual Studio [35]. However, these recommendations are instantiated from statically defined bug-fix strategies, while Phoenix automatically learns these strategies from examples.

FootPatch [47] proposed a static-analysis-based technique for automatically generating patches for violations of heap properties, flagged by a static analysis, namely Infer, in this case. Phoenix shares FootPatch's goal of generating patches for static analysis violations, without the use of test-cases. However, FootPatch's patch generation is tied to a specific class of violations, while Phoenix's approach can, in principle, target arbitrary violation classes, by learning from their examples. In other related work, Liu et al. describe a large-scale empirical study on fixing static analysis violations [30]. Specifically, they manually extract fix-patterns from a corpus of fixed FindBug violations mined from GitHub, and apply them to manually fix open FindBugs violations. Phoenix proposes a novel, fully-automated PbE approach for this problem setting, based on a custom DSL for specifying repair strategies and a synthesis algorithm to learn strategies from repair examples.

More recently, Getafix [43], a tool concurrently under development at Facebook, uses anti-unification techniques to automatically infer repair templates similar to [30] and use a special ranking technique that takes past human-fixes into account to derive the most plausible patches. We believe their ranking function can augment our final ranking to further improve our results.

Automated program repair (APR). A typical APR technique (implicitly or explicitly) 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. Some of these approaches, such as GenProg [27], ACS [51], SearchRepair [23], μScalpel [4], ssFix [50], and SimFix [20] mine concrete program snippets from existing code to construct the repairs. Others, such as PAR [24], Relifix [46], HDrepair [26], Genesis [31], and CapGen [49], extract abstract transformation schemas defining the space S, from existing corpora of patches. Yet others, such as Prophet [32] and Elixir [42] use features of such a corpus to train a classifier to rank the space S. Techniques such as SemFix [38], MintHint [22], DirectFix [33], NOPOL [52], and Angelix [34] use techniques like symbolic execution, to generate an oracular representation of the repair which is then synthesized into a concrete repair. 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. This allows it to construct fairly sophisticated, non-localized repairs while current APR techniques can typically only create one-line, or at most one-hunk, repairs. Further, these techniques rely on a test-suite while Phoenix's DSL is designed around using a static analyzer as the oracle.

Machine-learning based approaches such as [16] use generative-models to produce repairs and therefore can work without the aforementioned oracles. However, since it is trained on unlabeled and unpaired data, it cannot specialize on the various static analysis violation categories, something which contributes greatly to the efficacy of tools like ours.

Programming by Example (PbE). Recently PbE has been applied to a number of different domains [11, 15, 25, 28, 48, 53], each work proposing a DSL and program synthesis algorithm customized to its respective application. Refazer [40], the PbE technique closest to our work, learns from repetitive code edits. Refazer was applied to provide feedback (*i.e.,* patches) for student submissions of programming assignments in MOOC courses as well as in a *systematic edit* scenario, *i.e.,* where a user-provided set of examples for a target edit is used to replicate the edit at appropriate locations in a subject system. Refazer characterizes (and learns) changes as a set of independent edits, each rooted in its own code context. By contrast, Phoenix views each repair in terms of a primary node, to which each of the edits is related in a specific way. Phoenix also employs advanced higher-order pattern matching to achieve far better generalization. These core differences are reflected in the DSLs and synthesis methods of the two techniques and for Phoenix turn out to be crucial to its ability to learn sophisticated patches correctly (RQ3, Section 4.7).

8 CONCLUSION

In this work we proposed a novel technique for automatically generating high-quality patches for static analysis violations by learning from examples. This approach builds on the recent success of static analyzers, while completely avoiding the use of test-suites as a specification, which has hampered the progress of traditional APR tools. We implemented our solution in a system, Phoenix that includes a fully-automated pipeline that mines and cleans patches for static analysis violations from the wild, learns generalized executable repair strategies as programs in a novel Domain Specific Language (DSL), and then instantiates concrete repairs from them on new unseen violations. Using Phoenix we mined a corpus of 5,389 unique patches from 517 GitHub projects. In a cross-validation study on this corpus Phoenix successfully produced 4382 bug-fixes, with a recall of 85% and a precision of 54%. When applied to the latest revisions of a further 5 GitHub projects, Phoenix produced 94 correct patches to previously unknown bugs, 19 of which have already been accepted and merged by the development teams. In future work we propose to conduct a large-scale user study to further quantify the practical utility of Phoenix.

REFERENCES

[1] Pavel Avgustinov, Arthur I. Baars, Anders S. Henriksen, Greg Lavender, Galen Menzel, Oege de Moor, Max Schäfer, and Julian Tibble. 2015. Tracking Static Analysis Violations over Time to Capture Developer Characteristics. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 437–447. http://dl.acm.org/citation.cfm?id=2818754.2818809

[2] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Softw.* 25, 5 (Sept. 2008), 22–29. https://doi.org/10.1109/MS.2008.130

[3] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill. 2016. From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 211–221. https://doi.org/10.1109/ICSME.2016.63

[4] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 257–269.

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[6] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*. 3–11.

[7] V. Chvatal. 1979. A Greedy Heuristic for the Set-Covering Problem. *Math. Oper. Res.* 4, 3 (Aug. 1979), 233–235. https://doi.org/10.1287/moor.4.3.233

[8] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23 (03 2018).

[9] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[11] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

[12] L. Gazzola, D. Micucci, and L. Mariani. 2018. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* (2018), 1–1.

[13] Google. 2017. Error Prone. https://errorprone.info/. (2017).

[14] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[15] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[16] Jacob A. Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Peter Chin. 2018. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., USA, 7944–7954. http://dl.acm.org/citation.cfm?id=3327757.3327890

[17] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[18] Eclipse IDE. 2018. Java Editor Quickfix. https://help.eclipse.org/neon/topic/org.eclipse.jdt.doc.user/reference/ref-java-editor-quickfix.htm. (2018).

[19] JetBrains. 2018. IntelliJ Quick Fixes. https://www.jetbrains.com/resharper/features/quick_fixes.html. (2018).

[20] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 298–309.

[21] 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 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.

[22] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. MintHint: Automated Synthesis of Repair Hints. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 266–276.

[23] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 295–306.

[24] 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 (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 802–811.

[25] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 542–553.

[26] X. B. D. Le, D. Lo, and C. L. Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE Press, Piscataway, NJ, USA, 213–224.

[27] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 3–13.

[28] Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 565–574.

[29] Henry Lieberman. 2001. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[30] 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), 1–1. https://doi.org/10.1109/TSE.2018.2884955

[31] 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 (ESEC/FSE 2017)*. ACM, New York, NY, USA, 727–739.

[32] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 298–312.

[33] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 448–458.

[34] 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.

[35] Microsoft. 2018. Visual Studio - Common Quick Actions. https://docs.microsoft.com/en-us/visualstudio/ide/common-quick-actions. (2018).

[36] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1, Article 17 (Jan. 2018), 24 pages.

[37] Eugene W. Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1 (01 Nov 1986), 251–266. https://doi.org/10.1007/BF01840446

[38] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781.

[39] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

[40] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415.

[41] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66.

[42] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 648–659.

[43] Andrew Scott, Johannes Bader, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *CoRR* abs/1902.06111 (2019). arXiv:1902.06111 http://arxiv.org/abs/1902.06111

[44] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 532–543.

[45] Synopsys. 2018. 2017 Coverity Scan Report. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/SCAN-Report-2017.pdf. (2018).

[46] Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated Repair of Software Regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 471–482.

[47] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[48] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

[49] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1–11.

[50] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-related Code for Automated Program Repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 660–670.

[51] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 416–426.

[52] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (Jan 2017), 34–55.

[53] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 580–593. https://doi.org/10.1145/3177732.3177735

[54] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200.