AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations

Kui Liu, Anil Koyuncu, Dongsun Kim, Tegawendé F. Bissyandé
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
{kuiliu, anilkoyuncu, dongsun.kim, tegawende.bissyande}@uni.lu

Abstract—Fix pattern-based patch generation is a promising direction in Automated Program Repair (APR). Notably, it has been demonstrated to produce more acceptable and correct patches than the patches obtained with mutation operators through genetic programming. The performance of pattern-based APR systems, however, depends on the fix ingredients mined from fix changes in development histories. Unfortunately, collecting a reliable set of bug fixes in repositories can be challenging. In this paper, we propose to investigate the possibility in an APR scenario of leveraging code changes that address violations by static bug detection tools. To that end, we build the AVATAR APR system, which exploits fix patterns of static analysis violations as ingredients for patch generation. Evaluated on the Defects4J benchmark, we show that, assuming a perfect localization of faults, AVATAR can generate correct patches to fix 34/39 bugs. We further find that AVATAR yields performance metrics that are comparable to that of the closely-related approaches in the literature. While AVATAR outperforms many of the state-of-the-art pattern-based APR systems, it is mostly complementary to current approaches. Overall, our study highlights the relevance of static bug detection tools as indirect contributors of fix ingredients for addressing code defects identified with functional test cases.

Index Terms—Automated program repair, static analysis, fix pattern.

I. INTRODUCTION

The current momentum of Automated Program Repair (APR) has led to the development of various approaches in the literature [1]–[16]. In the software engineering community, the focus is mainly placed on fixing semantic bugs, i.e., those bugs that make the program behavior deviate from developer’s intention [1], [17]. Such bugs are detected by test suites. APR researchers have then developed repair pipelines where program test cases are leveraged not only for localizing the bugs but also as the oracle for validating the generated patches.

Unfortunately, given that test suites can be incomplete, typical APR systems are prone to generate nonsensical patches that might violate the intended program behavior or simply introduce other defects which are not covered by the test suites [4]. A recent study by Smith et al. [18] has thoroughly investigated this issue and found that overfitted patches are actually common: these patches can pass all the available test cases, but are not actually correct.

To address the problem of patch correctness in APR, two research directions are being investigated in the literature. The first direction attempts to develop techniques for automatically augmenting the test suites [19]. The second one focuses on improving the patch generation process to reduce the probability of generating nonsensical patches [20], [21]. The scope of our work is the latter.

Mining fix templates from common patches is a promising approach to achieve patch correctness. As first introduced by Kim et al. [4], patch correctness can be improved by relying on fix templates learned from human-written patches. In their work, the template constructions were performed manually, which is a limiting factor and is further error-prone [22]. Since then, several approaches have been developed towards automating the inference of fix patterns from fix changes in developer code bases [14], [20], [23]–[25]. A key challenging step in the inference of patterns, however, is the identification and collection of a substantial set of relevant bug fix changes to construct the learning dataset. Patterns must further be precise and diverse to actually guarantee repair effectiveness.

There have been approaches to mining fix patterns and exploring the challenges in achieving the diversity and reliability of fix ingredients. Long et al. [14] have relied on only three simple bug types, while Koyuncu et al. [21] have focused on bug linking between bug tracking systems and source code management systems to identify probable bug fixes. Unfortunately, the former approach cannot find patterns to address a variety of bugs, while the latter may include patterns that are irrelevant to bug fixes since developer changes are not atomic [26]; it is thus challenging to extract useful and reliable patterns focusing on fix changes.

Our work proposes a new direction for pattern-based APR to overcome the limitations in finding reliable and diverse fix ingredients. Concretely, we focus on developer patches that are fixing static analysis violations. The advantages of this approach are: (1) the availability of toolsets for assessing whether a code change is actually a fix [27], [28], and (2) the ability to further pre-categorize the changes into groups targeting specific violations, leading to consistent fix patterns [29], [30]. Although static analysis violations (e.g., FindBugs [31] warnings) may appear irrelevant to the problem of semantic bug fixing, there are two findings in the literature, which can support our intuition of leveraging fix patterns from static analysis violation patches to address semantic bugs:

- Locations of semantic bugs (unveiled through dynamic execution of test cases) can sometimes be detected by static analysis tools. In a recent study, Habib et al. [32], have found that some bugs in the Defects4J dataset can be identified by static analysis tools: SpotBugs [33], Infer [34] and ErrorProne [35]. Other studies [36]–[38] have also suggested...
that violations reported by static analysis tools might be smells of more severe defects in software programs.

- **Violation fix patterns have been used to successfully fix bugs in the wild.** In preliminary live studies, Liu et al. [29] and Rolim et al. [30] have shown that it can systematically fix statically detected bugs by using some of their previously-learned fix patterns. They further showed that project developers are even eager to integrate the systematization of such fixes based on the mined patterns.

Our work investigates to what extent fix patterns for static analysis violations can serve as ingredients to the patch generation step in an automated program repair pipeline.

This paper thus makes the following contributions:

1) **We discuss a counterpoint to a recent study in the literature on the usefulness of static analysis tools to address real bugs.** We find that, although static bug finding tools, can detect a relatively small number of real-world semantic bugs from the Defects4J dataset, fix patterns inferred from the patches addressing statically detected bugs can provide relevant ingredients in an APR pipeline that is targeting semantic bugs.

2) **We propose AVATAR (static analysis violation fix pattern-based automated program repair), a novel fix pattern-based approach to automated program repair.** Our approach differs from related work in the dataset of developer patches that is leveraged to extract fix ingredients. We build on patterns extracted from patches that have been verified (with bug detection tools) as true bug fix patches. Given the redundancy of bug types detected by static analysis tools, the associated fixes are intuitively more similar, leading to the inference of reliable common fix patterns. AVATAR is available at: https://github.com/SerVal-Repair/AVATAR.

3) **We report on an empirical assessment of AVATAR on the Defects4J benchmark.** We compare our approach with the state-of-the-art based on different evaluation aspects, including the number of fixed bugs, the exclusivity of fixed bugs, patch correctness, etc. Among several findings, we find that AVATAR is capable of generating correct patches for 34 bugs, and partially-correct patches for 5 bugs, when assuming a perfect fault localization scenario.

II. BACKGROUND

We provide background information on general pattern-based APR, as well as on pattern inference from static analysis violation data.

A. Automated Program Repair with Fix Patterns

Fix pattern-based APR has been widely explored in the literature [4], [10], [11], [14], [39], [40]. The basic idea is to abstract a code change into a pattern (interchangeably referred to as a template) that can be applied to a faulty code. The fixing process thus consists in leveraging context information of faulty code (e.g., abstract syntax tree (AST) nodes) to match context constraints defined in a given fix pattern. For example, the fix template “Method Replacer” provided in PAR [4] is presented as:

\[
\text{obj.method1(param) \rightarrow obj.method2(param)}
\]

where the faulty method call \text{method1} is replaced by another method call \text{method2} with compatible parameters and return type. A method call is the context information for this template to match buggy code fragment. Thus, this template can be applied to any faulty statement that includes at least one method call expression. The template further guides the patch candidate generation where changes are proposed to replace the potentially faulty method call with another method call.

Mining fix patterns has some intrinsic issues. The first issue relates to the variety of patterns that must be identified to support the fixing of different bug types. There are two strategies in fix pattern mining: (1) manual design and (2) automatic mining. The former can effectively create precise fix patterns. Unfortunately, it requires human effort, which can be prohibitive [4]. The latter infers common modification rules [14] or searches for the most redundant sub-patch instance [20], [21]. While this strategy can substantially increase the number of fix patterns, it is subject to noisy input data due to tangled changes [26], which make the inferred patterns less relevant. The second issue relates to the granularity (i.e., the degree of abstraction). Coarse-grained and monolithic patterns [41] can cover many types of bugs but they may not be actionable in APR. A fine-grained or micro pattern [14] can be readily actionable, but cannot cover many defects.

B. Static Analysis Violations

Static analysis tools help developers check for common programming errors in software systems. The targeted errors include syntactic defects, security vulnerabilities, performance issues, and bad programming practices. These tools are qualified as “static” because they do not require dynamic execution traces to find bugs. Instead, they are directly applied to source code or bytecode. In contrast with dynamic analysis tools which must run test cases, static tools can cover more paths, although it makes over-approximations that make them prone to false positives.

Many software projects rigorously integrate static analysis tools into their development cycles. The Linux kernel development project is such an example project where developers systematically run static analyzers against their code before pushing it to maintainers repositories [42]. More generally, FindBugs [31], PMD [43] and Google Error-Prone [35] are often used in Java projects, while C/C++ projects tend to adopt Splint [44], cppcheck [45], and Clang Static Analyzer [46].

Static analysis tools raise warnings, which are also referred to as alerts, alarms, or violations. Given that these warnings are due to the detection of code fragments that do not comply with some analysis rules, in the remainder of this paper we refer to the issues reported by static analysis tools as violations.

Figure 2 shows an example patch for a violation detected by FindBugs. This violation (the red code) is reported because the equals method should work for all object types (i.e.,
Mining fix patterns for static analysis violations has recently been explored in the literature [29], [30]. The general objective so far, however, is to learn quick fixes for speeding maintenance tasks and towards understanding which violations are prioritized by developers for fixing. To the best of our knowledge, our work is the first reported attempt to investigate fix patterns of static analysis violations in the context of automated program repair (where patches are generated and validated systematically with developer test cases).

There have been two recent studies of mining fix patterns addressing static analysis violations. Our previous study [29] focuses on identifying fix patterns for FindBugs violations [47], while Rolim et al. [30] consider PMD violations [48]. Both approaches, which were developed concurrently, leverage a similar methodology in the inference process. We summarise below the process of fix pattern mining of static analysis violations into three basic steps (as shown in Figure 1): data collection, data preprocessing, and fix pattern mining. Implementation details are strictly based on the approach of our previous study [29].

A. Data Collection

The objective of this step is to collect patches that are relevant to static analysis violations. This step is done in the wild based on the commit history of open-source projects by implementing a strict strategy to limit the dataset of changes to those that are relevant in the context of static analysis violations. To that end, it is necessary to systematically run static bug detection tools to each and every revision of the programs. This process can be resource-intensive: for example, FindBugs takes as input compiled versions of Java classes, requiring to build thousands of project revisions.

This step collects code changes (i.e., patches) only if they are identified as violation-fixing changes. For a given violation instance, we can assume that a change commit is a (candidate) fix for the instance when it disappears after the commit: i.e., the violation instance is identified in a revision of a program, but is no longer identified in the consecutive revision. Then, it is necessary to figure out whether the change actually fixed the violation instance or it just disappears by coincidence. If the affected code lines are located within the code change diff of the commit, it is regarded as an actual fix for the given violation instance. Otherwise, the violation instance might be removed just by deleting a method, class, or even a file. Eventually, all code change diffs associated to the identified fixed violation instances are collected to form the input data for fix pattern mining.

B. Data Preprocessing

Once violation patch data are collected, they are processed to extract concrete change actions. Patches submitted to program repositories are presented in the form of line-based GNU diffs where changes are reported in a text-based format of edit script. Given that, in modern programming languages, such as Java, source code lines do not represent a semantic entity of a study.
code entity (e.g., a statement may span across several lines), it is challenging to directly mine fix patterns from GNU diffs.

Pattern-mining studies leverage edit scripts of program Abstract Syntax Trees (ASTs). Concretely, the buggy version (i.e., program revision file where the violation can be found) and the fixed version (i.e., consecutive program revision file where the violation does not appear) are given as inputs to the GumTree [49], an AST-based code differencing tool, to produce the relevant AST edit script. This edit script describes in a fine-grained manner the repair actions that are implemented in the patch. Figure 4 provides an example GNU Diff for a bug fix patch, while Figure 5 illustrates the associated AST edit scripts.

--- a/src/com/google/javascript/jsccomp/Compiler.java
+++ b/src/com/google/javascript/jsccomp/Compiler.java
@@ -1283,4 +1283,3 @@
 // Check if the sources need to be re-ordered.
 if (options.dependencyOptions.needsManagement() &&
     !options.skipAllPasses &&
---UPD InfixExpression@"infix-expression code"
-----DEL PrefixExpression@"'"!options.skipAllPasses'"
-----DEL Operator@"&&"

Fig. 4. Patch of the bug Closure-13 in Defects4J.

---UPD IfStatement@"if statement code"
-----DEL PrefixExpression@"'"!options.skipAllPasses'"
-----DEL Operator@"&&"

Fig. 5. AST edit scripts produced by GumTree for the patch in Fig. 4.

C. Fix Pattern Mining

Given a set of edit scripts, the objective of the pattern mining step is to group “similar” scripts in order to infer the common subset of edit actions, i.e., a consistent pattern across the group. To that end, Rolim et al. [30] rely on the greedy algorithm to compute the distance among edit scripts. Edit scripts with low distances among them are grouped together. Our previous study [29], on the other hand, leverage a deep representation learning framework (namely, CNNs [50]) to learn features of edit scripts, which are then used to find clusters of similar edit scripts. Clustering is performed based on the X-means algorithm. Finally, the largest common subset of edit actions among all edit scripts in a cluster is considered as the pattern.

Mined fix patterns with this approach have already been proven useful by the authors. For example, our previous study [29] and Rolim’s work [30] conducted live studies by making pull requests to projects in the wild: the pull requests contained change details of a patch that is generated based on the inferred fix patterns to fix static analysis violations in developer code. Developers accepted to merge 67 out of 116 patches generated for FindBugs violations in our previous study [29]. Similarly, 6 out of 16 pull requests by Rolim et al. [30] have been merged by developers in the wild. Such promising results demonstrated the possibility to automatically fix bugs that are addressed by static bug detection tools.

**Fix patterns of static analysis violations have been explored in the literature to automate patch generation for bugs that are statically detected. To the best of our knowledge, AVATAR is the first attempt to leverage fine-grained patterns of static analysis violations as fix ingredients for automated program repair that addresses semantic bugs revealed by test cases.**

IV. OUR APPROACH

As shown in Figure 3, AVATAR consists of four major steps for automated program repair. In this section, we detail the objective and design of each step, and provide concrete information on implementation.

A. Fault Localization

We rely on the GZoltar\(^4\) [51] framework to automate the execution of test cases for each program. In the framework, we leverage the Ochiai [52] ranking metric to actually compute the suspiciousness scores of statements that are likely to be the faulty code locations. This ranking metric has been demonstrated in several empirical studies [53]–[56] to be effective for localizing faults in object-oriented programs. The GZoltar framework for fault localization is also widely used in the literature of APR [15], [16], [20], [21], [57]–[60].

B. Fix Pattern Matching

In the running of the repair pipeline, once fault localization produces a list of suspicious code locations, AVATAR iteratively attempts to match each of these locations with a given pattern from the database of mined fix patterns. Fix patterns in our database are collected from the artifacts released by Liu et al. [29] and Rolim et al. [30]. Table I shows statistics about the pattern collection in these previous works. As most of the fix patterns released by Liu et al. will not change the program behavior, we only select 13 of them for 10 violation types after manually checking that they can change the program behavior (details shown in the aforementioned website).

Recall that each pattern is actually an edit script of repair actions on specific AST node types. AST nodes associated

\(^{3}\)http://program-repair.org/defects4j-dissection/#/bug/Closure/31

\(^{4}\)http://www.gzoltar.com

459
to the faulty code locations are then regarded as the context of matching the fixing patterns; i.e., these nodes are checked against the nodes involved in the edit scripts of fix patterns. For example, the fix pattern shown in Figure 6 contains three levels of contexts: (1) IfStatement, which means that the pattern is matched only if the suspicious faulty statement is an IfStatement; (2) InfixExpression indicates that the pattern is relevant when the predicate expression of the suspicious IfStatement is an InfixExpression; (3) the matched InfixExpression predicate in the suspicious statement must contain at least two sub-predicate expressions.

Fix Pattern: Remove a useless sub-predicate expression.
UPD IfStatement
---UPD InfixExpression@expA Operator expB
-----DEL Expression@expB
-----DEL Operator

Fig. 6. A fix pattern for UC_USELESS_CONDITION violation [29].

A pattern is found to be relevant to a faulty code location only if all AST node contexts at this location matches with the AST node of the pattern. For example, the bug shown in Figure 4 is located within an IfStatement with an InfixExpression which is formed by three sub-predicate expressions. This buggy fragment thus matches the fix pattern shown in Figure 6.

C. Patch Generation

Given a suspicious code location and an associated matching fix pattern, AVATAR applies the repair actions in the edit scripts of the pattern to generate patch candidates. For example, the code change action of the fix pattern in Figure 6 is interpreted as removing a sub-condition expression (or sub-predicate expression) in a faulty IfStatement. Thus, three patch candidates, as shown in Figure 7, can be generated by AVATAR for the buggy code in Figure 4 since the statement has three candidate sub-predicates expressions.

D. Patch Validation

Patch candidates generated by AVATAR must then be systematically assessed. Eventually, using test cases, our approach verifies whether a patch candidate is a plausible patch or not. We target two types of plausible patches:

- **Fully-fixing** patches, which are patches that make the program pass all available test cases. Once such a patch is validated, the execution iterations of AVATAR are halted.
- **Partially-fixing** patches, which are patches that make the program pass not only all previously-passing test cases, but also part of the previously-failing test cases.

The first generated fully-fixing patch is prioritized over any other generated patch, and is considered as the plausible

---

TABLE I

STATISTICS ON FIX PATTERNS OF STATIC ANALYSIS VIOLATIONS.

<table>
<thead>
<tr>
<th>Project</th>
<th># Projects</th>
<th># violation</th>
<th># violation</th>
<th># fix patch types</th>
<th># patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Liu et al. [29]</td>
<td>730</td>
<td>89,927</td>
<td>111</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>Kolim et al. [30]</td>
<td>9</td>
<td>288,899</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

---

TABLE II

DEFECTS4J DATASET INFORMATION.

<table>
<thead>
<tr>
<th>Project</th>
<th>Bugs</th>
<th>kLoC</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart (Chart)</td>
<td>26</td>
<td>96</td>
<td>2,205</td>
</tr>
<tr>
<td>Closure compiler (Closure)</td>
<td>133</td>
<td>90</td>
<td>7,927</td>
</tr>
<tr>
<td>Apache commons-lang (Lang)</td>
<td>65</td>
<td>22</td>
<td>2,245</td>
</tr>
<tr>
<td>Apache commons-math (Math)</td>
<td>106</td>
<td>85</td>
<td>3,602</td>
</tr>
<tr>
<td>Mockito</td>
<td>38</td>
<td>11</td>
<td>1,457</td>
</tr>
<tr>
<td>Joda-Time (Time)</td>
<td>27</td>
<td>28</td>
<td>4,130</td>
</tr>
<tr>
<td>Total</td>
<td>395</td>
<td>332</td>
<td>21,566</td>
</tr>
</tbody>
</table>

"Bugs", "kLoC", and "Tests" denote respectively the number of bugs, the program size in kLoC (i.e., thousands of lines of code), and the number of test cases for each subject. The overall number of kLoC and test cases for project Mockito are not indicated in the Defects4J paper [61] from where the reported information is excerpted.

A. Research Questions

Our investigation into the repair performance of AVATAR seeks to answer the following research questions (RQs):

- RQ1: How effective are fix patterns of static analysis violations for repairing programs with semantic bugs?

Recall that we broadly consider as semantic bugs all bugs
that are uncovered by executing developer test cases. Our first research question assesses how many bugs in the Defects4J benchmark can be fixed with fix patterns of static analysis violations. To that end, we first (1) investigate how effectively AVATAR can fix such semantic bugs that appear to be localizable by static analysis tools. Then, (2) building on the assumption that the location of the semantic bug is known, we investigate whether AVATAR can generate a correct patch to fix it.

- **RQ2: Which bugs and which patterns are relevant targets for AVATAR in an automated program repair scenario?** This research question dissects the data yielded during the investigation of RQ1, with the objective of assessing the diversity of bugs that can be fixed as well as the types of violation fix patterns that have been successfully leveraged.

- **RQ3: How does AVATAR compare to the state-of-the-art with respect to repair performance?** With this research question, we aim at showing whether the proposed approach is relevant in the landscape of APR systems. Does AVATAR offer comparable performance? To what extent can AVATAR complement existing APR systems?

**B. Experimental Setup**

For evaluation purpose, we apply different fault localization schemes to the experiment of each RQ, while the default setting of AVATAR is to use the GZoltar framework with the Ochiai ranking metric for ordering suspicious statements. The usage of GZoltar and Ochiai reduces the comparison biases since both are widely used by APR systems in the literature.

- First, we apply AVATAR to Defects4J bugs that are localized by three state-of-the-art static analysis tools (namely, SpotBugs [33], Facebook Infer [34], and Google ErrorProne [35]) (for RQ1; see Section V-C). To that end, we consider recent data reported by Habib and Pradel [32]. This configuration focuses on the effectiveness of AVATAR on such semantic bugs that can also be detected statically.

- Second, we apply AVATAR on all faulty code positions in the benchmark (for RQ1; see Section V-D). We thus assume that a perfect localization is possible, and assess the performance of the approach on all bugs.

- Finally, for RQ3, we compare AVATAR with the state-of-the-art APR tools that are evaluated on the Defects4J benchmark (see Section V-F). To that end, we attempt to replicate two scenarios of fault localization used in APR assessments: the first scenario assumes that the faulty method name is known [10] and thus focuses on ranking the inner-statements based on Ochiai suspiciousness scores; the second scenario makes no assumption on fault location and thus uses the default setting of AVATAR.

**C. Applying AVATAR to Statically-Detected Bugs in Defects4J**

Table III provides details on the Defects4J bugs that can be detected by static analysis tools and are successfully repaired by AVATAR. We report that out of the 14, 4, and 7 bugs that can be detected respectively by SpotBugs, Facebook Infer and Google ErrorProne on version 1.2.0 of Defects4J, AVATAR can successfully generate correct patches for 3, 2 and 1 bugs.

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>SpotBugs</th>
<th>Infer</th>
<th>ErrorProne</th>
<th>Static Analysis Violation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart-1</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>NP_NONNULL_NULL</td>
</tr>
<tr>
<td>Chart-4</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>NP_NONNULL_ON_SOME_PATH</td>
</tr>
<tr>
<td>Chart-24</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>DLS_DEAD_LOCAL_STORE</td>
</tr>
<tr>
<td>Math-77</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>JVM_UNCALLED_PRIVATE_METHOD</td>
</tr>
<tr>
<td>Total</td>
<td>14 (18)</td>
<td>24 (5)</td>
<td>7 (8)</td>
<td></td>
</tr>
</tbody>
</table>

\[
x/y(z) \text{ reads as: } x \text{ is the number of bugs fixed by AVATAR among the } y \text{ bugs in version 1.2.0 of Defects4J that can be localized by each static analysis tool. } z \text{ is given as an indicator for the number of statically localizable bugs in an augmented version of Defects4J. The information on localizable bugs is excerpted from the study of Habib and Pradel [32]. Since most of the state-of-the-art APR systems targeting Java program are evaluated on the version 1.2.0, our experiments focus on localizable bugs in this version.}

Overall four distinct localizable bugs have been correctly fixed with patches generated from the fix patterns that were mined from patches fixing FindBugs violations [29]. All the four bugs are related to distinct violation types. In their work, Liu et al. [29] claimed that their mined patterns could be applied to violations reported by other static analysis tools. Our experiment indeed shows that these patterns fixed two bugs detected by SpotBugs (i.e., the successor of FindBugs), which are also detected by Facebook Infer.

**RQ1►** AVATAR demonstrates the usefulness of violation fix patterns in a scenario of automating some maintenance tasks involving systematic checking of developer code with static checkers.

Our experiments, however, reveal that AVATAR’s fix pattern for static analysis violations are not effective on many supposedly statically-detectable bugs in Defects4J. We investigate the cases of such bugs, and find that:

1) some of the bugs have a code context that does not match any of the mined fix patterns;
2) in some cases, the detection is actually a coincidental false positive reported in [32], since the violation does not really match the semantically faulty code entity that must be modified. Figure 8 provides a descriptive example of such false positives.
3) finally, in other cases, the fixes are truly domain-specific, and no pattern is thus applicable.

//\ Violation Type: FE_FLOATING_POINT_EQUALITY
//\ The comparing result of two floating point values for
//\ equality may not be accurate.
//\ Defects4J Dissection:
//\ Bug Pattern: Conditional block removal.
- if (x == x1) {
  - \( x0=0.5*(x0+x1-FastMath.max(\text{atol}*FastMath.abs(x1),\text{atol})) \);
  - f0=\text{computeObjectiveValue}(x0);
  - }
  break;

Fig. 8. A bug is false-positively identified as a statically-detected bug in [32] (Math-50): the violation is not related to the test case failures.

7\ A null pointer is dereferenced and will lead to a NullPointerException when the code is executed.
8\ Possible null pointer dereference.
9\ Dead store to a local variable.
10\ A private method is never called.
11\ https://github.com/rjust/defects4j/pull/112
D. Applying Avatar to All Defects4J Bugs

After focusing on those Defects4J bugs that can be statically detected, we run Avatar on all the dataset bugs. Given that the objective is to assess whether a correct patch can be generated if the bug is known, we assume in this experiment that the faulty code locations are known. Concretely, we do not rely on any fault localization tool. Instead, we consider the ground truth of developer patches and list the locations that have been modified as the faulty locations.

The repair operations thus consist in generating patches for the relevant bug locations. Table IV details the number of Defects4J bugs that are fixed by Avatar. Fully and partially fixed bugs are fixed with fully-fixing and partially-fixing patches (c.f. Section IV-D) generated by Avatar, respectively. Overall, Avatar can fix 49 bugs with plausible patches (i.e., that pass all available tests). 35 of them are further manually confirmed to be correct (i.e., they are syntactically or at least semantically equivalent to the ground truth patches submitted by developers). We also note that, for 14 other bugs, Avatar generates plausible patches that make the program pass some previously-failing test cases, without failing any of the previously-passing test cases. Five among these patches are manually found to be correctly fixing part of the bugs. To the best of our knowledge, Avatar is the first APR tool which partially, but correctly, fixes bugs in Defects4J that have multiple fault code locations.

Table: Number of Defects4J bugs fixed by Avatar with an assumption of perfect localization information.

<table>
<thead>
<tr>
<th>Fixed Bugs</th>
<th>Chart (C)</th>
<th>Closure (Cl)</th>
<th>Lang (L)</th>
<th>Math (M)</th>
<th>Mockito (Moc)</th>
<th>Time (T)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Fully Fixed Bugs</td>
<td>7/8</td>
<td>10/13</td>
<td>5/10</td>
<td>8/13</td>
<td>2/2</td>
<td>2/3</td>
<td>34/49</td>
</tr>
<tr>
<td># Partially Fixed Bugs</td>
<td>2/3</td>
<td>1/4</td>
<td>1/5</td>
<td>1/4</td>
<td>0/0</td>
<td>0/0</td>
<td>5/14</td>
</tr>
</tbody>
</table>

† In each column, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a plausible patch is generated by the APR tool. The same as the following similar tables.

E. Dissecting the Fix Ingredients

We now investigate how fix patterns of static analysis violations can be leveraged to address semantic bugs from the Defects4J benchmark. To that end, we dissect the ingredients that were successfully leveraged in the generated correct patches. Table V provides the summary of this dissection.

First, we note that all correctly fixed bugs were addressed with patches generated from patterns mined in the study of Liu et al. [29] (i.e., based on FindBugs violations). Fix patterns from the study by Rolim et al. [30] (which are based on PMD violations) are indeed associated to exceedingly simple violations, which are unlikely to be revealed as semantic bugs (i.e., detected via developer test cases). An example of such simple pattern is their EP7 fix pattern: “replace List<String> a = new ArrayList() with List<String> a = new ArrayList<>()”.

F. Comparing against the State-of-the-Art

To reliably compare against the state-of-the-art in Automated Program Repair (APR), we must ensure that the Fault Localization (FL) step is properly tuned as FL could bias the bug fixing performance of APR tools [63]. We identify three major configurations in the literature:

1) Normal FL-based APR: in this case, APR systems directly use off-the-shelf fault localization techniques to localize the faulty code positions. In this case, which is realistic, the suspicious list of fault locations may be inaccurate leading to a poor repair performance. APR tools that are run in ASTOR [57] work under this configuration.

2) Restricted FL-based APR: in this case, APR systems make the assumption that some information of the code location is available. For example, in HDRepair [10],

The current code contains a useless control flow statement, where the control flow continues onto the same place regardless of whether or not the branch is taken.

The cast expression is unchecked or unconfirmed, and not all instances of the type casted from can be cast to the type it is being cast to. It needs to check that the program logic ensures that this cast will not fail.
TABLE V

FIX INGREDIENTS LEVERAGED IN THE STATIC ANALYSIS VIOLATION FIX PATTERNS USED BY AVATAR TO CORRECTLY FIX SEMANTIC BUGS.

<table>
<thead>
<tr>
<th>Violation Types associated with the Fix Patterns</th>
<th>Fix Ingredients from the Violation Fix Patterns</th>
<th>Fixed Bug IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP_ALWAYS_NULL</td>
<td>Mutate the operator of null-check expression: &quot;var ! = null&quot;; or &quot;var != l= null&quot;.</td>
<td>C-1, C-4, 26, C-14, 19, C-25, Cl-2, M-4, M-89, Moc-29, 38.</td>
</tr>
<tr>
<td>NP_NULL_ON_SOME_PATH</td>
<td>1. Wrap buggy code with if-null-check block: &quot;if (var == null) {buggy code}&quot;; 2. Insert if-null-check block before buggy code: &quot;if (var == null) {return false;} buggy code;&quot; or &quot;if (var == null) {throwIllegalArgumentException.} buggy code;&quot;.</td>
<td>C-11, 24, L-6, 57, 59, M-33, 59, T-7.</td>
</tr>
<tr>
<td>DLS_DEAD_LOCAL_STORE</td>
<td>Replace a variable with other one: e.g., &quot;var1 = var3;&quot;.</td>
<td>Cl-18, 31, Cl-38, 62, Cl-63, 73, L-15, M-46, M-82, 85, T-19.</td>
</tr>
<tr>
<td>UC_USELESS_CONDITION</td>
<td>1. Mutate the operator of an expression in an if statement: e.g., &quot;if {expA ! = expB} {...}&quot;; 2. Remove a sub-predicate expression in an if statement: &quot;if {expA ! = expB} {...}&quot; or &quot;if {expA ! = expB} {...}&quot;; 3. Remove the conditional expression: &quot;expA ! = expB&quot; or &quot;expA ! = expB expC&quot;.</td>
<td>Cl-18, Cl-106, Cl-115, 126, L-7, 10, M-50.</td>
</tr>
<tr>
<td>UCF_USELESS_CONTROL_FLOW(^{13})</td>
<td>1. Remove an if statement but keep the code inside its block: &quot;if {code }&quot;; 2. Remove an if statement with its block code: &quot;if {code }&quot;.</td>
<td>Cl-18, Cl-106, Cl-115, 126, L-7, 10, M-50.</td>
</tr>
<tr>
<td>UFM_UNCALLED_PRIVATE_METHOD</td>
<td>Remove a method declaration: &quot;remove method declaration (Parameters: {code}).&quot;</td>
<td>Cl-46, M-77.</td>
</tr>
<tr>
<td>BC_UNCONFIRMED_CAST(^{14})</td>
<td>Wrap buggy code with if-instanceof-check block: &quot;if {var instanceof Type} {buggy code} else {throwIllegalArgumentException.}&quot;.</td>
<td>M-89.</td>
</tr>
</tbody>
</table>

\(^1\) Only correctly fixed (including partially correctly-fixed bugs highlighted with italic) bugs are listed in this table.

TABLE VI

DISSECTION OF BUGS CORRECTLY FIXED BY AVATAR.

<table>
<thead>
<tr>
<th>Bug IDs</th>
<th>Bug Patterns</th>
<th>Repair Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-1</td>
<td>Conditional expression modification</td>
<td>Logic expression modification.</td>
</tr>
<tr>
<td>C-4, 26.</td>
<td>Missing non-null check addition</td>
<td>Conditional (if) branch addition.</td>
</tr>
<tr>
<td>C-14, 19, 25, Cl-2, M-4, Moc-29, 38.</td>
<td>Missing null check addition</td>
<td>Conditional (if) branch addition.</td>
</tr>
<tr>
<td>C-11, 24, L-6, 57, 59, M-33, 59, T-7.</td>
<td>Wrong variable reference</td>
<td>Variable replacement by another variable.</td>
</tr>
<tr>
<td>Cl-38.</td>
<td>Logic expression expansion</td>
<td>Conditional expression expansion.</td>
</tr>
<tr>
<td>Cl-62, 63, 73, M-82, 85, T-19.</td>
<td>Logic expression modification</td>
<td>Conditional expression modification.</td>
</tr>
<tr>
<td>C-18, Cl-106, M-46.</td>
<td>Unwraps-from if-else statement</td>
<td>Conditional (if or else) branch removal.</td>
</tr>
<tr>
<td>Cl-115, 126, L-7, 10, M-50.</td>
<td>Conditional block removal</td>
<td>Conditional (if or else) branch removal.</td>
</tr>
<tr>
<td>Cl-46, M-77.</td>
<td>Unclassified</td>
<td>Method definition removal.</td>
</tr>
<tr>
<td>M-89.</td>
<td>Wraps-with if-else statement</td>
<td>Conditional (if-else) branches addition.</td>
</tr>
</tbody>
</table>

We thus compare the bug fixing performance of AVATAR with the state-of-the-art APR tools after classifying them into these three groups.

1) Comparison against a Restricted_FL-based APR System: We first compare AVATAR against the HDRepair [10] state-of-the-art APR system, which implements a restricted fault localization configuration. We select faulty locations using the same assumption as HDRepair, i.e., focusing on attempting to repair suspicious code statements that are reported by our fault localization tool but filtering only those that are within the known faulty methods. This assumption leaves out many noisy statements, reducing the probability of generating overfitting patches for bugs and further increasing the chance to generate a correct patch before a plausible one or any execution timeout.

Table VII presents the comparing results. Comparing with HDRepair, AVATAR correctly fixes many more bugs (31+4 vs 6) and yields a higher probability to generate correct patches among all plausible patches (cf. P(%) in Table VII). 34
TABLE VII  
COMPARISON OF Avatar with HDRRepair [10].

<table>
<thead>
<tr>
<th>Project</th>
<th>HDRRepair</th>
<th>Avatar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fully fixed</td>
<td>Partially fixed</td>
</tr>
<tr>
<td>Chart</td>
<td>0/2</td>
<td>7/9</td>
</tr>
<tr>
<td>Closure</td>
<td>0/7</td>
<td>9/15</td>
</tr>
<tr>
<td>Lang</td>
<td>2/6</td>
<td>5/12</td>
</tr>
<tr>
<td>Math</td>
<td>4/7</td>
<td>6/14</td>
</tr>
<tr>
<td>Mockito</td>
<td>0/0</td>
<td>2/2</td>
</tr>
<tr>
<td>Time</td>
<td>0/1</td>
<td>2/3</td>
</tr>
<tr>
<td>Total</td>
<td>6/23</td>
<td>31/55*</td>
</tr>
<tr>
<td>P (%)</td>
<td>26.1</td>
<td>56.4</td>
</tr>
</tbody>
</table>

The results for HDRRepair are provided by its author.  *The number of bugs fixed by Avatar shown in this table is a little different from the data in Table IV. For fixing each bug, the input of Avatar is a ranked list of suspicious statements in the faulty methods, which is different from the input of Avatar in the experiment of Section V-D.

(except Lang-6) out of the 35 bugs fixed by Avatar are not addressed by HDRRepair. Avatar also correctly fixes 7 bugs (as highlighted with bold in Figure 9) that are only plausibly (but incorrectly) fixed by HDRRepair. Finally, Avatar partially fixes 11 bugs that have multiple faulty code fragments, and 4 of the associated patches are correct. Figure 9 illustrates the space of correctly fixed bugs by HDRRepair and Avatar.

![HDRepair vs Avatar](image)

Fig. 9. Bugs correctly fixed by HDRRepair and Avatar, respectively.

**RQ3** Avatar substantially outperforms the HDRRepair approach on the Defects4J benchmark.

2) **Comparison against Normal_FL-based APR Systems:** We compare the bug fixing performance of Avatar with the Normal_FL-based state-of-the-art APR tools that are evaluated on the Defects4J benchmark. These APR tools take as input a ranked list of suspicious statements that are reported by an off-the-shelf fault localization technique. In this experiment, we consider a group of APR systems, namely JGenProg [66], jKali [66], jMutRepair [57], Nopol [15], FixMiner [21] and LSRepair [60], which leverage a similar configuration as Avatar for fault localization: GZoltar/Ochiai.

Table VIII reports the comparison results in terms of the number of plausibly-fixed bugs and the number of correctly-fixed bugs. Data about the fixed bugs are directly excerpted from the results reported in the relevant research papers. We note that Avatar outperforms all of the Normal_FL-based APR systems, both in terms of the number of plausibly fixed bugs and the number of correctly fixed bugs. It also yields a higher probability to generate correct patches among its plausible patches than those tools (except FixMiner). Finally, 18 (15 + 3, as shown in the second row of Table IX) among the 30 (27 + 3) bugs correctly fixed by Avatar have not been correctly fixed by those Normal_FL-based APR tools.

3) **Comparison against Supplemented_FL-based APR Systems:** We also compare Avatar against APR systems which use supplementary information to improve fault localization accuracy. We include in this category other APR systems whose authors do not explicitly describe the actual fault localization configuration, but which still manage to fix bugs that we could not localize with GZoltar/Ochiai. We include in this group the following state-of-the-art works targeted at Java programs: ACS [16], ELIXIR [40], JAID [12], ssFix [58], CapGen [59], SketchFix [65] and SimFix [20].

The compared performance results are also illustrated in Table VIII. Based on the number of correctly fixed bugs, Avatar is only inferior to SimFix but outperforms other Supplemented_FL-based APR systems. Avatar further correctly fixes 14 (11 + 3, as shown in the third row of Table IX) out of 31 bugs that have never been addressed by any Supplemented_FL-based state-of-the-art APR system.

To sum up, Avatar correctly fixes 11 (as shown in the fourth row of Table IX) out of 31 bugs that have never been addressed by any state-of-the-art APR system. We also note that Avatar outperforms state-of-the-art APR tools on fixing bugs in project Chart, Closure and Mockito.

**RQ3** Avatar underperforms against some of the most recent APR systems. Nevertheless, Avatar is still complementary to them as it is capable of addressing some Defects4J bugs that the state-of-the-art cannot fix.

VI. **THREATS TO VALIDITY**

A threat to external validity is related to use of Defects4J bugs as a representative set of semantic bugs. This threat is mitigated as it is currently a widely used dataset in the APR literature related to Java. A threat to internal validity is due to the use of Java programs as subjects. Eventually, we only considered fix patterns for FindBugs and PMD violations. Other static tools, especially for C programs, such as Splint, cppcheck, and Clang Static Analyzer are not investigated. A threat to construct validity may involve the assumption of perfect localization to assess Avatar. This threat is minimized by the different other experiments that are comparable with evaluations in the literature.

VII. **RELATED WORK**

The software development practice is increasingly accepting generated patches [42]. Recently, various directions in the literature have been explored to contribute to the advancement of automated program repair. One commonly studied direction is the pattern based (also called example-based) APR. Kim et al. [4] initiated with PAR a milestone of APR based on fix
templates that were manually extracted from 60,000 human-written patches. Later studies [10] have shown that the six templates used by PAR could fix only a few bugs in Defects4J. Long and Rinard also proposed a patch generation system, Prophet [11], that learns code correctness models from a set of successful human patches. They further proposed a new system, Genesis [14], which can automatically infer patch generation transforms from developer submitted patches for program repair.

Motivated by PAR [4], more effective automated program repair systems have been explored. HDRepair [10] was proposed to repair bugs by mining closed frequent bug fix patterns from graph-based representations of real bug fixes. Nevertheless, its fix patterns, except the fix templates from PAR, still limits the code change actions at abstract syntax tree (AST) node level, but are not specific for some types of bugs. ELIXIR [40] aggressively uses method call related templates from PAR with local variables, fields, or constants, to construct more expressive repair-expressions that go into synthesizing patches.

Tan et al. [39] integrated anti-patterns into two existing search-based automated program repair tools (namely, GenProg [3] and SPR [8]) to help alleviate the problem of incorrect or incomplete fixes resulting from program repair. In their study, the anti-patterns are defined by themselves and limited to the control flow graph. Additionally, their anti-patterns are not meant to solve the problem of deriving better patches automatically, provide more precise repair hints to developers.

More recently, CapGen [59], SimFix [20], FixMiner [21] are further proposed to fix bugs automatically based on the frequently occurred code change operations (e.g., Insert If-Statement (c.f., Table 3 in [20]) that are extracted from the patches in developer change histories.

So far however, pattern-based APR approaches focus on leveraging patches that developer applied to semantic bugs. To the best of our knowledge, our approach is first to investigate the case of leveraging patches that fix static analysis violations: they are many more, better identifiable, and more consistent.

VIII. CONCLUSION

The correctness of patches generated is now identified as a barrier in the adoption of automated program repair systems. Towards guaranteeing correctness, researchers have been investigating example-based approaches where fix patterns from human patches are leveraged in patch generation. Nevertheless, such ingredients are often hard to collect reliably. In this work, we propose to rely on developer patches that address static analysis bugs. Such patches are concise and precise, and their efficacy (in removing the bugs) are systematically assessed (by the static detectors). We build AVATAR, an APR system that utilizes fix ingredients from static analysis violations patches. We empirically show that AVATAR is indeed effective in repairing programs that have semantic bugs. AVATAR outperforms several state-of-the-art approaches and complements others by fixing some of the Defects4J bugs which were not yet fixed by any APR system in the literature.

As future work, we plan to assess avatar on bigger bug datasets [67], and use this concept of static analysis-based patterns for improving method refactoring research [68], [69].

ACKNOWLEDGEMENTS

This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, through RECOMMEND 15/IS/1044967 and FIXPATTERN C15/IS/9964569.


