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
{kui.liu, anil.koyuncu, 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 finding 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 are patches that 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

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

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:

`obj.`**`method1`**`(param)` → `obj.`**`method2`**`(param)`

where the faulty method call `method1` is replaced by another method call `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.,

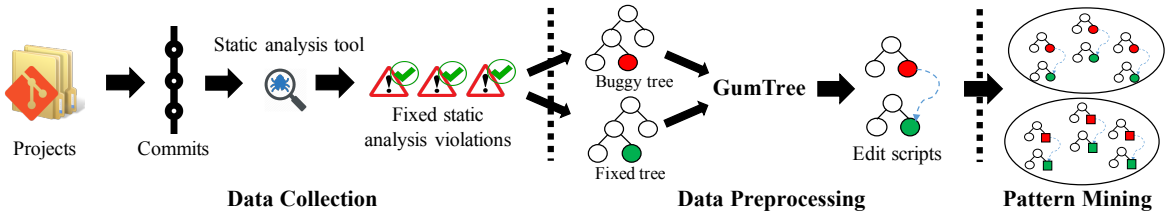

Fig. 1. Summarized steps of static analysis violation fix pattern mining.

```
 public boolean equals(Object obj) {
 // Violation Type: BC_EQUALS_METHOD_SHOULD_WORK_FOR_ALL_OBJECTS
-    return getModule().equals(}
-           ((ModuleWrapper) obj).getModule());
+    return obj instanceof ModuleWrapper &&
+           getModule().equals(
+           ((ModuleWrapper) obj).getModule());
 }
```

Fig. 2. Example patch for fixing a violation detected by FindBugs. Example excerpted from [29].

`Object`): in this case, the method code violates the rule since it assumes a specific type (i.e., `ModuleWrapper`).

Note that, not all violations are accepted by developers as actual defects. Since static analysis tools use limited information, detected violations could be correct code (i.e., false positive) or the warning may be irrelevant (e.g., cannot occur at runtime, or not a serious issue). In the literature, many studies assume that a violation can be classified as actionable if it is discarded after a developer changed the location where the violation is detected. The violation in Figure 2 [29] is fixed by adding an `instanceof` check (c.f., the green code in the patch diff); this violation can thus be regarded as *actionable* since this violation is gone after fixing its source code.

Motivation*:* Mining patterns from developer patches that fix static analysis violations may help overcome the issues of fix pattern mining described in Section II-A. First, since static analysis tools specify the type of each violation (e.g., bug descriptions[1] of FindBugs), each bug instance is already classified as long as it is fixed by code changes. Thus, we can reduce the manual effort to collect and classify bugs and their corresponding patches for fix pattern mining. Second, we can mitigate the issue of tangled changes [26] because violation-fixing changes can be localized by static analysis tools [28]. Finally, the granularity of fix patterns can be appropriately adjusted for each violation type since static analysis tools often provide information on the scope of each violation instance.

III. Mining Fix Patterns for Static Violations

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 [2] 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

[1]http://findbugs.sourceforge.net/bugDescriptions.html

[2]A "code change diff" consists of two code snapshots. One snapshot represents the code fragment that will be affected by a code change, while the other one represents the code fragment after it has been affected by the code change.

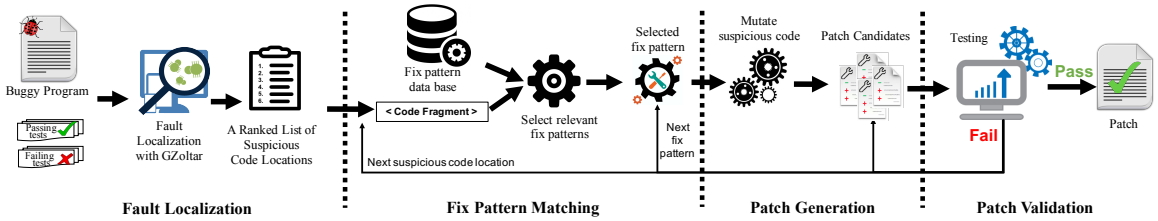

Fig. 3. Overview bug fixing process with AVATAR.

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/jscomp/Compiler.java
+++ b/src/com/google/javascript/jscomp/Compiler.java
@@ -1283,4 +1283,3 @@
   // Check if the sources need to be re-ordered.
   if (options.dependencyOptions.needsManagement() &&
-              !options.skipAllPasses &&
               options.closurePass) {

// Defects4J Dissection:
// Repair Action: Conditional expression reduction.
```

Fig. 4. Patch of the bug Closure-13[3] in Defects4J.

```
UPD IfStatement@@``if statement code''
---UPD InfixExpression@@``infix-expression 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

TABLE I
STATISTICS ON FIX PATTERNS OF STATIC ANALYSIS VIOLATIONS.

	# Projects	# violation fix patches	# violation types	# fix patterns
Liu et al. [29]	730	88,927	111	174
Rolim et al. [30]	9	288,899	9	9

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`[5] 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 patch for the given bug. After iterating over all suspicious statements with all matching fix patterns, if AVATAR fails to generate a *fully-fixing* patch for a bug, but generates some *partially-fixing* patches, these patches are considered as plausible patches. Nevertheless, we implement a selection scheme where *partially-fixing* patches that change the program control-flow are prioritized over those that only change data-flow.

Partially-fixing patches that change the control flow are further naïvely ordered by the edit distances (at AST node level) between the patched fragments and the buggy fragment: smaller edit distances are preferred. Ties are broken by considering precedence in the generation: the first generated *partially-fixing* patch is the final plausible patch.

[5]The condition has no effect and always produces the same result as the value of the involved variable was narrowed before. Probably something else was meant or condition can be removed.

```
  // Patch Candidate I.
- if (options.dependencyOptions.needsManagement() &&
-             !options.skipAllPasses &&
+ if (!options.skipAllPasses &&
              options.closurePass) {

  // Patch Candidate II.
  if (options.dependencyOptions.needsManagement() &&
-             !options.skipAllPasses &&
              options.closurePass) {

  // Patch Candidate III.
  if (options.dependencyOptions.needsManagement() &&
-             !options.skipAllPasses &&
-              options.closurePass) {
+             !options.skipAllPasses) {
```

Fig. 7. Patch Candidates generated by AVATAR with a fix pattern that is mined from patches for `UC_USELESS_CONDITION` violations (cf. Fig. 6), and which matches the buggy statement in Closure-13 bug (cf. Fig. 4).

V. ASSESSMENT

We evaluate AVATAR on Defects4J [61], which is widely used by state-of-the-art APR systems targeting Java programs. Table II summarizes the statistics on the number of bugs and test cases available in the version 1.2.0[6] of Defects4J.

TABLE II
DEFECTS4J DATASET INFORMATION.

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

"**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

[6]https://github.com/rjust/defects4j/releases/tag/v1.2.0

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 III
STATICALLY-DETECTED BUGS FIXED BY AVATAR.

Bug ID	SpotBugs	Infer	ErrorProne	Static Analysis Violation Type
Chart-1	●	●		NP_ALWAYS_NULL[7]
Chart-4	●	●		NP_NULL_ON_SOME_PATH[8]
Chart-24	●			DLS_DEAD_LOCAL_STORE[9]
Math-77			●	UPM_UNCALLED_PRIVATE_METHOD[10]
Total	3/14 (18)	2/4 (5)	1/7 (8)	

$x/y(z)$ reads as: x is the number of bugs fixed by AVATAR among the y bugs in version 1.2.0 of Defects4J that can be localized by each static analysis tool. z is given as an indicator for the number of statically localizable bugs in an augmented version[11] 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 patterns 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(rtol*FastMath.abs(x1),atol));
-   f0=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

TABLE IV
NUMBER OF DEFECTS4J BUGS FIXED BY AVATAR WITH AN ASSUMPTION OF PERFECT LOCALIZATION INFORMATION.

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

† 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.

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.

RQ1▶AVATAR *can effectively fix several semantic bugs from the Defects4J dataset. We even observe that the fine-grained fix ingredients can be helpful to target bugs with multiple faulty code locations.*

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<>()`". In any case, 6 among the 9 fix patterns released by Rolim et al. are related to performance, code practice or code style. Our manual investigation of Defects4J bugs reveals that none of the bugs are associated to these types of issues.

Second, we note that the fix patterns of only seven (out of 10) violation types have been successfully used to generate correct patches for Defects4J bugs (c.f. Table V). Among the 40 (fully or partially) correctly fixed bugs, 36 (90%) are fixed with fix patterns of 4 violation types: `NP_NULL_ON_SOME_PATH`, `DLS_DEAD_LOCAL_STORE`, `UC_USELESS_CONDITION`, and `UCF_USELESS_CONTROL_FLOW`. The latter two violation types are related to the issues of conditional code entities (e.g., If statements and conditional expressions), which are relevant to 18 (45%) of the fixed bugs. Comparing against the ACS [16] state-of-the-art APR tool which focuses on repairing condition-related faulty code entities, we find that AVATAR correctly fixes 15 relevant bugs that are not fixed by ACS.

Finally, we investigate the diversity of the bugs that are correctly addressed by AVATAR. To that end, we resort to the dissection study of Defects4J bugs by Sobreira et al. [62]. Table VI summarizes the bug patterns and the associated repair actions for the bugs that are correctly fixed by AVATAR. We note that AVATAR can currently address 11 bug patterns out of the 60 bug patterns enumerated in the dissection study.

RQ2▶AVATAR *exploits a variety of fix ingredients from static violations fix patterns to address a diverse set of bug patterns in the Defects4J dataset.*

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],

[12]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.

[13]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.

Violation Types associated with the Fix Patterns	Fix Ingredients from the Violation Fix Patterns	Fixed Bug IDs
NP_ALWAYS_NULL	Mutate the operator of null-check expression: "`var != == null`", or "`var == != null`".	C-1.
NP_NULL_ON_SOME_PATH	1. Wrap buggy code with if-non-null-check block: "`if (var != null) {buggy code}`"; 2. Insert if-null-check block before buggy code: "`if (var == null) {return false;} buggy code;`" or "`if (var == null) {return null;} buggy code;`" or "`if (var == null) {throw IllegalArgumentException.} buggy code;`".	C-4,26, C-14,19, *C-25*,Cl-2, M-4, Moc-29,38.
DLS_DEAD_LOCAL_STORE	Replace a variable with other one: e.g., "`var1 = ~~var2~~var3;`".	C-11,24, L-6,57,59, M-33,59,T-7.
UC_USELESS_CONDITION	1. Mutate the operator of an expression in an `if` statement: e.g., "`if (expA ~~>>=~~ expB) {...}`"; 2. Remove a sub-predicate expression in an `if` statement: "`if (~~expA ||~~ expB) {...}`" or "`if (expA ~~|| expB~~) {...}`"; 3. Remove the conditional expression: "`~~expA ? expB :~~ expC`" or "`~~expA ?~~ expB ~~: expC~~`".	Cl-18,31, Cl-38,62, Cl-63,73, *L-15*,M-46, M-82,85, T-19.
UCF_USELESS_CONTROL_FLOW[12]	1. Remove an `if` statement but keep the code inside its block: "`~~if (exp) {~~ code ~~}~~`"; 2. Remove an `if` statement with its block code: "`~~if (exp) {~~ code ~~}~~`".	*C-18*, *Cl-106*, Cl-115,126, L-7,10, M-50.
UPM_UNCALLED_PRIVATE_METHOD	Remove a method declaration: "`~~Modifier ReutrnType methodName(Parameters) {~~ code ~~}~~`".	Cl-46,*M-77*.
BC_UNCONFIRMED_CAST[13]	Wrap buggy code with if-instanceof-check block: "`if (var instanceof Type) {buggy code}` `else {throw IllegalArgumentException;}`".	M-89.

† 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.

Bug IDs	Bug Patterns	Repair Actions
C-1.	Conditional expression modification	Logic expression modification.
C-4, 26.	Missing non-null check addition	Conditional (if) branch addition.
C-14, 19, 25, Cl-2, M-4, Moc-29, 38.	Missing null check addition	Conditional (if) branch addition.
C-11, 24, L-6, 57, 59, M-33, 59, T-7.	Wrong variable reference	Variable replacement by another variable.
Cl-38.	Logic expression expansion	Conditional expression expansion.
Cl-18, 31, L-15.	Logic expression reduction	Conditional expression reduction.
Cl-62, 63, 73, M-82, 85, T-19.	Logic expression modification	Conditional expression modification.
C-18, Cl-106, M-46.	Unwraps-from if-else statement	Conditional (if or else) branch removal.
Cl-115, 126, L-7, 10, M-50.	Conditional block removal	Conditional (if or else) branch removal.
Cl-46, M-77.	Unclassified	Method definition removal.
M-89.	Wraps-with if-else statement	Conditional (if-else) branches addition.

fault localization is restricted to the faulty methods, which are assumed to be known. Such a restriction actually substantially increases the accuracy of the target list of fault locations for which a patch must be generated. In Section V-D, we have made a similar strong assumption that fault locations are known as our objective was to assess the patch generation performance of AVATAR.

3) ***Supplemented_FL*-based APR**: in this case, APR systems leverage existing fault localization tools but improve the localization approach with some heuristics to ensure that the patch generation targets an accurate list of code locations. For example, the SimFix [20] recent state-of-the-art system employs a test purification [64] technique to improve the accuracy of the fault localization.

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 HDREPAIR [10].

Project	HDRepair	AVATAR	
		Fully fixed	Partially fixed
Chart	0/2	7/9	1/3
Closure	0/7	9/15	1/2
Lang	2/6	5/12	1/3
Math	4/7	6/14	1/3
Mockito	0/0	2/2	0/0
Time	0/1	2/3	0/0
Total	6/23	31/55*	4/11*
P (%)	26.1	56.4	36.4

The results for HDRepair 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 HDRepair. AVATAR also correctly fixes 7 bugs (as highlighted with **bold** in Figure 9) that are only plausibly (but incorrectly) fixed by HDRepair. 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 HDRepair and AVATAR.

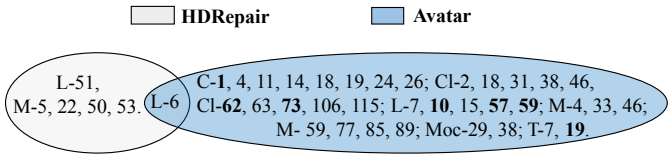

Fig. 9. Bugs correctly fixed by HDRepair and AVATAR, respectively.

RQ3►AVATAR *substantially outperforms the HDRepair 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.

RQ3►*In terms of quantity and quality of generated plausible patches,* AVATAR *addresses more bugs than its immediate competitors. Nevertheless, we note that* AVATAR *is actually complementary to the other state-of-the-art APR systems, fixing bugs that others do not fix.*

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

TABLE VIII
NUMBER OF BUGS REPORTED AS HAVING BEEN FIXED BY DIFFERENT APR SYSTEMS.

Fault Localization	APR Tool		Chart	Closure	Lang	Math	Mockito	Time	Total	P*(%)
Normal_FL-based APR	AVATAR	Fully Fixed	**5/12**	**8/12**	5/11	6/13	**2/2**	1/3	27/53*	50.9
		Partially Fixed	1/2	1/1	0/2	1/3	0/0	0/0	3/8*	37.5
	jGenProg [57]		0/7	0/0	0/0	5/18	0/0	0/2	5/27	18.5
	jKali [57]		0/6	0/0	0/0	1/14	0/0	0/2	1/22	4.5
	jMutRepair [57]		1/4	0/0	0/1	2/11	0/0	0/1	3/17	17.6
	Nopol [15]		1/6	0/0	3/7	1/21	0/0	0/1	5/35	14.3
	FixMiner [21]		5/8	5/5	2/3	12/14	0/0	1/1	25/31	80.65
	LSRepair [60]		3/8	0/0	8/**14**	7/14	1/1	0/0	19/37	51.4
Supplemented_FL-based APR	**ACS [16]**		2/2	0/0	3/4	12/16	0/0	1/1	18/23	78.3
	ELIXIR [40]		4/7	0/0	8/12	12/19	0/0	**2**/3	26/41	63.4
	JAID [12]		2/4	5/11	1/8	1/8	0/0	0/0	9/31	29.0
	ssFix [58]		3/7	2/11	5/12	10/**26**	0/0	0/**4**	20/**60**	33.3
	CapGen [59]		4/4	0/0	5/5	12/16	0/0	0/0	21/25	**84.0**
	SketchFix [65]		**6**/8	3/5	3/4	7/8	0/0	0/1	19/26	73.1
	SimFix [20]		4/8	6/8	**9**/13	**14/26**	0/0	1/1	**34**/56	60.7

"**P**" is the probability of generated plausible patches to be correct.
*The number of bugs fixed by AVATAR shown in this table is a little different from the data in Table IV and Table VII. In this experiment, for fixing each bug, the input of AVATAR is a ranked full list of suspicious statements in the faulty program, which is different from the input of AVATAR in the experiments of Table IV and Table VII.

TABLE IX
BUGS FIXED BY AVATAR BUT NOT CORRECTLY FIXED BY OTHER APR TOOLS.

APR tool group	Bug IDs	
	Fully-fixed	Partially-fixed
Normal_FL-based APR tools	C-14,19,Cl-2,18,31,46,L-6,7,10,M-4,46,59,Moc-29,38,T-7.	C-18, Cl-106, M-77.
Supplemented_FL-based APR tools	C-4,Cl-2,31,38,46,L-6,7,10,M-46,Moc-29,38.	C-18, Cl-106, M-77.
All APR tools	Cl-2,31,46,L-7,10,M-46,Moc-29,38.	C-18, Cl-106, M-77.

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/10449467 and FIXPATTERN C15/IS/9964569.

REFERENCES

[1] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 772–781.

[2] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. ACM, 2009, pp. 364–374.

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

[4] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the International Conference on Software Engineering*. ACM, 2013, pp. 802–811.

[5] Z. Coker and M. Hafiz, "Program transformations to fix c integers," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2013, pp. 792–801.

[6] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2015, pp. 295–306.

[7] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. ACM, 2015, pp. 448–458.

[8] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 166–178.

[9] X.-B. D. Le, Q. L. Le, D. Lo, and C. Le Goues, "Enhancing automated program repair with deductive verification," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 428–432.

[10] X. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering,*. IEEE, 2016, pp. 213–224.

[11] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 298–312, 2016.

[12] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proceedings of the 32nd International Conference on Automated Software Engineering*. IEEE, 2017, pp. 637–647.

[13] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 593–604.

[14] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 727–739.

[15] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[16] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*. ACM, 2017, pp. 416–426.

[17] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 129–139.

[18] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 532–543.

[19] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 831–841.

[20] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 298–309.

[21] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *arXiv preprint arXiv:1810.01791*, 2018.

[22] M. Monperrus, "A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 234–242.

[23] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Proceedings of the 36th International Conference on Software Engineering-Companion*. ACM, 2014, pp. 492–495.

[24] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon, "A closer look at real-world patches," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2018, pp. 275–286.

[25] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 832–837.

[26] K. Herzig and A. Zeller, "The Impact of Tangled Code Changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. ACM, 2013, pp. 121–130.

[27] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 41–50.

[28] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. de Moor, M. Schäfer, and J. Tibble, "Tracking static analysis violations over time to capture developer characteristics," in *Proceedings of the 37th International Conference on Software Engineering*. ACM, 2015, pp. 437–447.

[29] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, 2019 to appear.

[30] R. Rolim, G. Soares, R. Gheyi, and L. D'Antoni, "Learning quick fixes from code repositories," *arXiv preprint arXiv:1803.03806*, 2018.

[31] "FindBugs," http://findbugs.sourceforge.net, last accessed: Oct.2018.

[32] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 317–328.

[33] "SpotBugs," https://spotbugs.github.io, last accessed: Oct.2018.

[34] "Facebook Infer," https://fbinfer.com, last accessed: Oct.2018.

[35] "Google Error-Prone," https://errorprone.info, last accessed: Oct.2018.

[36] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An experience report on using code smells detection tools," in *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 450–457.

[37] N. Moha, Y.-G. Gueheneuc, A.-F. Duchien *et al.*, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[38] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *Proceedings of the 20th Working Conference on Reverse Engineering*. IEEE, 2013, pp. 242–251.

[39] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 727–738.

[40] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2017, pp. 648–659.

[41] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[42] A. Koyuncu, T. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Impact of Tool Support in Patch Construction," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 237–248.

[43] "PMD," https://pmd.github.io, last accessed: Oct.2018.

[44] "Splint," https://www.splint.org, last accessed: Oct. 2018.

[45] "cppcheck," http://cppcheck.sourceforge.net, last accessed: Oct.2018.

[46] "Clang Static Analyzer," https://clang-analyzer.llvm.org, last accessed: Oct.2018.

[47] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[48] T. Copeland, "Pmd Applied," 2005.

[49] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated Software Engineering*. ACM, 2014, pp. 313–324.

[50] M. Matsugu, K. Mori, Y. Mitari, and Y. Kaneda, "Subject independent facial expression recognition with robust face detection using a convolutional neural network," *Neural Networks*, vol. 16, no. 5-6, pp. 555–559, 2003.

[51] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 378–381.

[52] R. Abreu, A. J. Van Gemund, and P. Zoeteweij, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[53] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 314–324.

[54] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 4, p. 31, 2013.

[55] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 191–200.

[56] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*. ACM, 2017, pp. 609–620.

[57] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 441–444.

[58] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2017, pp. 660–670.

[59] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 1–11.

[60] K. Liu, K. Anil, K. Kim, D. Kim, and T. F. Bissyandé, "LSRepair: Live search of fix ingredients for automated program repair," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. IEEE, 2018, pp. 658–662.

[61] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[62] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 130–140.

[63] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2019.

[64] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 52–63.

[65] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 12–23.

[66] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[67] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs. jar: a large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 10–13.

[68] M. Pradel and K. Sen, "Deepbugs: a learning approach to name-based bug detection," *PACMPL*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018.

[69] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*. IEEE, 2019.