2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings

Diego Marcilio
University of Brasília
Brasilia, Brazil
dvmarcilio@gmail.com

Carlo A. Furia
USI – Università della Svizzera Italiana
Lugano, Switzerland
bugcounting.net

Rodrigo Bonifácio
University of Brasília
Brasilia, Brazil
rbonifacio@unb.br

Gustavo Pinto
Federal University of Pará
Belém, Brazil
gpinto@ufpa.br

***Abstract*—Static code analysis tools such as FindBugs and SonarQube are widely used on open-source and industrial projects to detect a variety of issues that may negatively affect the quality of software. Despite these tools' popularity and high level of automation, several empirical studies report that developers normally fix only a small fraction (typically, less than 10% [1]) of the reported issues—so-called "warnings". If these analysis tools could also automatically provide *suggestions* on how to fix the issues that trigger some of the warnings, their feedback would become more actionable and more directly useful to developers.**

In this work, we investigate whether it is feasible to automatically generate fix suggestions for common warnings issued by static code analysis tools, and to what extent developers are willing to accept such suggestions into the codebases they're maintaining. To this end, we implemented a Java program transformation technique that fixes 11 distinct rules checked by two well-known static code analysis tools (SonarQube and SpotBugs). Fix suggestions are generated automatically based on templates, which are instantiated in a way that removes the source of the warnings; templates for some rules are even capable of producing multi-line patches. We submitted 38 pull requests, including 920 fixes generated automatically by our technique for various open-source Java projects, including the Eclipse IDE and both SonarQube and SpotBugs tools. At the time of writing, project maintainers accepted 84% of our fix suggestions (95% of them without any modifications). These results indicate that our approach to generating fix suggestions is feasible, and can help increase the applicability of static code analysis tools.

I. Introduction

Static code analysis tools (SATs) are becoming increasingly popular as a way of detecting possible sources of defects earlier in the development process [2]. By working *statically* on the source or byte code of a project, these tools are applicable to large code bases [3], [4], where they quickly search for patterns that may indicate problems—bugs, questionable design choices, or failures to follow stylistic conventions [5], [6]—and report them to users. There is evidence [7] that using these tools can help developers monitor and improve software code quality; indeed, static code analysis tools are widely used for both commercial and open-source software development [1], [2], [4]. Some projects' development rules even require that code has to clear the checks of a certain SAT before it can be released [1], [7], [8].

At the same time, some features of SATs limit their wider applicability in practice. One key problem is that SATs are necessarily *imprecise* in checking for rule violations; in other words, they report *warnings* that may or may not correspond to an actual mistake. As a result, the first time a static analysis tool is run on a project, it is likely to report thousands of warnings [2], [3], which saturates the developers' capability of sifting through them to select those that are more relevant and should be fixed [1]. Another related issue with using SATs in practice is that understanding the problem highlighted by a warning and coming up with a suitable fix is often nontrivial [1], [3].

Our research aims at improving the practical usability of SATs by automatically providing *fix suggestions*: modifications to the source code that make it compliant with the rules checked by the analysis tools. We developed an approach, called SpongeBugs and described in Section III, whose current implementation works on Java code. SpongeBugs detects violations of 11 different rules checked by SonarQube and SpotBugs (successor to FindBugs [2])—two well-known static code analysis tools, routinely used by very many software companies and consortia, including large ones such as the Apache Software Foundation and the Eclipse Foundation. The rules checked by SpongeBugs are among the most widely used in these two tools, and cover different kinds of code issues (ranging from performance, to correct behavior, style, and other aspects). For each violation it detects, SpongeBugs automatically generates a fix suggestion and presents it to the user.

By construction, SpongeBugs's suggestions remove the origin of a rule's violation, but the maintainer still has to decide—based on their overall knowledge of the project—whether to accept and merge each suggestion. To assess whether developers are indeed willing to accept SpongeBugs's suggestions, Section V describes the results of an empirical evaluation where we applied SpongeBugs to 12 Java projects, and submitted 920 fix suggestions as pull requests to the projects. At the time of writing, project maintainers accepted 775 (84%) fix suggestions—95% of them without any modifications. This high acceptance rate suggests that SpongeBugs often generates patches of high quality, which developers find adequate and useful. The empirical evaluation also indicates that SpongeBugs is applicable with good performance to large code bases; and reports (in Section V-D) several qualitative findings that can inform further progress in this line of work.

The work reported in this paper is part of a large body of

2470-6892/19/$31.00 ©2019 IEEE
DOI 10.1109/SCAM.2019.00013

research (see Section II) that deals with helping developers detecting and fixing bugs and code smells. SpongeBugs' approach is characterized by the following features: *i*) it targets static rules that correspond to frequent mistakes that are often fixable *syntactically*; *ii*) it builds fix suggestions that remove the source of warning *by construction*; *iii*) it scales to large code bases because it is based on lightweight program transformation techniques. Despite the focus on conceptually simple rule violations, SpongeBugs can generate nontrivial patches, including some that modify multiple hunks of code at once. In summary, SpongeBugs's focus privileges generating a large number of practically useful fixes over being as broadly applicable as possible. Based on our empirical evaluation, Section VI discusses the main limitations of SpongeBugs's approach, and Section VII outlines directions for further progress in this line of work.

II. Background and Related Work

Static analysis techniques reason about program behavior *statically*, that is without running the program [9]. This is in contrast to *dynamic* analysis techniques, which are instead driven by specific program inputs (provided, for example, by unit tests). Thus, static analysis techniques are often more scalable (because they do not require complete executions) but also less precise (because they over-approximate program behavior to encompass all possible inputs) than dynamic analysis techniques. In practice, there is a broad range of static analysis techniques from purely *syntactic* ones—based on code patterns—to complex *semantic* ones—which infer *behavioral* properties that can be used to perform program optimizations [10], [11] as well as performance problems and other kinds of vulnerabilities [12], [13].

Static Code Analysis Tools. Static code analysis *tools* (SATs) typically combine different kinds of analyses. This paper focuses on the output of tools such as SonarQube, FindBugs, and SpotBugs, because they are widely available and practically used [2]. The analysis performed by a SAT consists of several independent *rules*, which users can select in every run of the tool. Each rule describes a requirement that correct, high-quality code should meet. For example, a rule may require that *Constructors should not be used to instantiate* `String`—one should write `String s = "SpongeBugs"` instead of `String s = new String("SpongeBugs")`.

Whenever a SAT finds a piece of code that *violates* one of the rules, it outputs a *warning*, typically reporting the incriminated piece of code and a reference to the rule that it violates. It is then up to the developer to inspect the warning to decide whether the violation is real and significant enough to be addressed; if it is, the developer also has to come up with a *fix* that modifies the source code and removes the rule violation without otherwise affecting program behavior.

Empirical Studies on Static Code Analysis Tools. Software engineering researchers have become increasingly interested [2] in studying how SATs are used in practice by developers, and how to increase their level of automation.

Rausch et al. [14] performed an extensive study on the build failures of 14 projects, finding a frequent association between build failures and issues reported by SATs (when the latter are used as a component of continuous integration). Zampetti et al. [15] found that lack of a consistent coding style across a project is a frequent source of build failures.

Recent studies focused on the kinds of rule violations developers are more likely to fix. Liu et al. [16] compared a large number of *fixed* and *not fixed* FindBugs rule violations across revisions of 730 Java projects. They characterized the categories of violations that are often fixed, and reported several situations in which the violations are systematically ignored. They also concluded that developers disregard most of FindBugs violations as not being severe enough to be fixed during development process.

Digkas et al. [17] performed a similar analysis for SonarQube rules, revealing that a small number of all rules accounts for the majority of programmer-written fixes. Marcilio et al. [1] characterized the use of SonarQube in 246 Java projects, reporting a low resolution rate of issues (around 9% of the project's issues have been fixed), and also finding that a small subset of the rules reveal real design and coding flaws that developers consider serious. These findings suggest that a fix generation tool that focuses on a selection of rules is likely to be highly effective and relevant in practice to real project developers.

Automatic Fix Suggestions and Program Repair. Several researchers have developed techniques that propose fix suggestions for rules of the popular FindBugs in different ways: interactively, with the user exploring different alternative fixes for the same warning [5]; and automatically, by systematically generating fixes by mining patterns of programmer-written fixes [4], [16]. These studies focus on *behavioral* bugs such as those in Defects4J [18]—a curated collection of 395 real bugs of open-source Java projects. In contrast, SpongeBugs mainly targets rules that characterize so-called *code smells*—patterns that may be indicative of poor programming practices, and mainly encompass design and stylistic aspects. We focus on these because they are simpler to characterize and fix "syntactically" but are also the categories of warnings that developers using SATs are more likely to address and fix [1], [16], [17]. This focus helps SpongeBugs achieve high precision and scalability, as well as be practically useful.

The work closest to ours is probably Liu et al.'s study [4], which presents the AVATAR automatic program repair system. AVATAR recommends code changes based on the output of SAT tools. In its experimental evaluation, AVATAR generated correct fixes for 34 bugs among those of the Defects4J benchmark. This suggests that responding to warnings of SATs can be an effective way to fix some behavioral "semantic" bugs. While we may run SpongeBugs on Defects4J bugs as well, AVATAR's and SpongeBugs' scopes are mostly *complementary* in the kind of design trade-offs they target. Our approach focuses on "syntactic" design flaws that often admit simple yet effective fixes, with the main goal of being immediately and generally useful for the kinds of static analysis flaws that developers fix more often.

Aftandilian et al. [19] presented an extension of the OpenJDK compiler that detects potential bugs and recommends fixes at compile time, based on patterns that are similar to those used by SATs like FindBugs. Other approaches learn transformations from examples, using sets of bug fixes [20], bug reports [21], or source-code editing patterns [22]. We directly implemented SpongeBugs' transformations based on our expertise and the standard recommendations for fixing warnings from static analysis tools. Even though SpongeBugs cannot learn new fixing rules at the moment, this remains an interesting direction for further improving its capabilities.

Behavioral bugs are also the primary focus of techniques for "automated program repair", a research area that has grown considerably in the last decade. The most popular approaches to automated program repair are mainly driven by dynamic analysis (i.e., tests) [23], [24] and targets generic bugs. In contrast, SpongeBugs' approach is based on static code-transformation techniques, which makes it of more limited scope but also more easily and widely applicable.

III. SpongeBugs: Approach and Implementation

SpongeBugs provides fix suggestions for violations of selected rules that are checked by SonarQube and SpotBugs. Section III-A discusses how we selected the rules to check and suggest fixes for. SpongeBugs works by means of source-to-source transformations, implemented as we outline in Section III-B.

A. Rule Selection

One key design decision for SpongeBugs is which static code analysis rules it should target. Crucially, SATs are prone to generating a high number of false positives [25]. To avoid creating fixes to spurious warnings, we base our design on the assumption that rules whose violations are frequently fixed by developers are more likely to correspond to real issues of practical relevance [1], [16].

We collected and analyzed the publicly available datasets from three previous studies that explored developer behavior in response to output from SonarQube [1], [17] and FindBugs [16]. Based on this data, we initially selected the top 50% most frequently fixed rules, corresponding to 156 rules, extended with another 10 rules whose usage was not studied in the literature but appear to be widely applicable.

Then, we went sequentially through each rule, starting from the most frequently fixed ones, and manually selected those that are more amenable to automatic fix generation. The main criterion to select a rule is that it should be possible to define a syntactic fix template that is guaranteed to remove the source of warning without obviously changing the behavior. This led to discarding all rules that are not *modular*, that is, that require changes that affect clients in any files. An example is the rule *Method may return* `null`*, but its return type is* `@Nonnull`.[1] Although conceptually simple, the fix for a violation of this rule entails a change in a method's signature that weakens the guarantees on its return type. This is impractical, since we would need to identify and check every call of this method, and potentially introduces a breaking change [26]. We also discarded rules when automatically generating a syntactic fix would be cumbersome or would require additional design decisions. An example is the rule *Code should not contain a hard coded reference to an absolute pathname*, whose recommended solution involves introducing an environment variable. To provide an automated fix for this violation, our tool would need an input from developers, since pathnames are context specific; it would also need access to the application's execution environment, which is clearly beyond the scope of the source code under analysis.

We selected the top rules (in order of how often developers fix the corresponding warnings) that satisfy these feasibility criteria, leading to the 11 rules listed in Table I. Note that SonarQube and SpotBugs rules largely overlap, but the same rule may be expressed in slightly different terms in either tool. Since SonarQube includes all 11 rules we selected, whereas SpotBug only includes 7 of them, we use SonarQube rule names[2] for uniformity throughout the paper.

Consistently with SonarQube's classification of rules, we assign an identifier to each rule according to whether it represents a bug (B1 and B2) or a code smell (C1–C9). While the classification is fuzzy and of limited practical usefulness, note that the most of our rules are code smells in accordance with the design decisions behind SpongeBugs.

TABLE I: The 11 static code analysis rules that SpongeBugs can provide fix suggestions for. The rule descriptions are based on SonarQube's, which classifies rules in (B)ugs and (C)ode smells.

ID	RULE DESCRIPTION
B1	Strings and boxed types should be compared using `equals()`
B2	`BigDecimal(double)` should not be used
C1	String literals should not be duplicated
C2	String functions use should be optimized for single characters
C3	Strings should not be concatenated using `+` in a loop
C4	Parsing should be used to convert strings to primitive types
C5	Strings literals should be placed on the left-hand side when checking for equality
C6	Constructors should not be used to instantiate `String`, `BigInteger`, `BigDecimal`, and primitive wrapper classes
C7	`entrySet()` should be iterated when both key and value are are needed
C8	`Collection.isEmpty()` should be used to test for emptiness
C9	`Collections.EMPTY_LIST`, `EMPTY_MAP`, and `EMPTY_SET` should not be used

Rules C1 and C5 were selected *indirectly* on top of the feasibility criteria discussed above (which were used directly to select the other 9 rules). We selected rule C1 because it features very frequently among the open issues of many

[1] https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#np-method-may-return-null-but-is-declared-nonnull-np-nonnull-return-violation

[2] https://rules.sonarsource.com/java

projects; its fixes are somewhat challenging since they involve multiple lines and the insertion of a constant. We selected rule C5 because it can be fixed in conjunction with fixes to rule B1 (see Listing 1), making the code shorter while also avoiding `NullPointerException` from being thrown.

```
- if (render != null && render != "")
+ if (!"".equals(render))
```

Listing 1: Fixes for rules B1 (*Strings and boxed types should be compared using* `equals()`) and C5 (*Strings literals should be placed on the left-hand side when checking for equality*) applied in conjunction.

B. How SpongeBugs Works

SpongeBugs looks for rule violations and builds fix suggestions in three steps:

1) Find textual patterns that might represent a rule violation.
2) For every match identified in step 1, perform a full search in the AST looking for rule violations.
3) For every match confirmed in step 2, instantiate the rule's fix templates—producing the actual fix for the rule violation.

We implemented SpongeBugs using Rascal [27], a domain-specific language for source code analysis and manipulation. Rascal facilitates several common meta-programming tasks, including a first-class visitor language constructor, advanced pattern matching based on concrete syntax, and defining templates for code generation. We used Rascal's Java 8 grammar [28], which entails that our evaluation (Section V) is limited to Java projects that can be built using this version of the language.

We illustrate how SpongeBugs's three steps work for rule C8 (`Collection.isEmpty()` *should be used to test for emptiness*). Step 1 performs a fast, but potentially imprecise, search that is based on some textual necessary conditions for a rule to be triggered. For rule C8, step 1 looks for files that import some package in `java.util` *and* include textual patterns that indicate a comparison of `size()` with zero or one—as shown in Listing 2.

```
bool shouldContinueWithASTAnalysis(loc fileLoc) {
  javaFileContent = readFile(fileLoc);
  return findFirst(javaFileContent, "import java.util.") != -1 &&
    hasSizeComparison(javaFileContent);
}

bool hasSizeComparison(str javaFileContent) {
  return findFirst(javaFileContent, ".size() \> 0") != -1 ||
    findFirst(javaFileContent, ".size() \>= 1") != -1 ||
    findFirst(javaFileContent, ".size() != 0") != -1 ||
    findFirst(javaFileContent, ".size() == 0") != -1;

}
```

Listing 2: Implementation of step 1 for rule C8: find textual patterns that might represent a violation of rule C8.

Step 1 may report false positives: for rule C8, the comparison involving `size()` may not actually involve an instance of a collection but use a class that does not offer a method `isEmpty()`. Step 2 is more precise, but also more computationally expensive, as it performs a full AST matching; therefore, it is only applied after step 1 identifies code that has a high likelihood of being rule violations. In our example of rule C8, step 2 checks that the target of the possibly offending call to `size()` is indeed of type `Collection`—as shown in Listing 3.

```
case (EqualityExpression)
  ‘<ExpressionName beforeFunc>.size() == 0‘: {
  if (isBeforeFuncReferencingACollection(beforeFunc,
    mdl, unit)) {
```

Listing 3: Partial implementation of step 2 for rule C8: full AST search for rule violations.

Whenever step 2 returns a positive match, step 3 executes and finally generate a patch to fix the rule violation. Step 3's generation is entirely based on *code-transformation templates* that modify the AST matched in step 2 as appropriate according to the rule's semantics. For rule C8, step 3's template is straightforward: replace the comparison of `size()== 0` with a call to `isEmpty()`—its implementation is in Listing 4. (Step 3 for rule C8 also handles other patterns not shown in this example for brevity.)

```
refactoredExp = parse(#Expression, "<beforeFunc>.isEmpty()");
```

Listing 4: Implementation of step 3 for rule C8: instantiate the fix templates corresponding to the violated rule.

IV. Empirical Evaluation of SpongeBugs: Experimental Design

The general goal of this research is to investigate the use of techniques for fixing suggestions to address the warnings generated by static code analysis tools. Section IV-A presents the research questions answered by SpongeBugs's empirical evaluation, which is based on 15 open-source projects selected using the criteria we present in Section IV-B. We submitted the fix suggestions built by SpongeBugs as pull requests according to the protocol discussed in Section IV-C. All data created in this study and our tool implementation are available:

https://github.com/dvmarcilio/spongebugs

A. Research Questions

The empirical evaluation of SpongeBugs, whose results are described in Section V, addresses the following research questions, which are based on the original motivation behind this work: automatically providing fix suggestions that helps improve the practical usability of SATs.

RQ1. How widely applicable is SpongeBugs?
The first research question looks into how many rule violations SpongeBugs can detect and suggest a fix for.

RQ2. Does SpongeBugs generate fixes that are acceptable?
The second research question evaluates SpongeBugs's effectiveness by looking into how many of its fix suggestions were accepted by project maintainers.

RQ3. How efficient is SpongeBugs?
The third research question evaluates SpongeBugs's scalability in terms of running time on large code bases.

B. Selecting Projects for the Evaluation

In order to evaluate SpongeBugs in a realistic context, we selected 15 well-established open-source Java projects that can be analyzed with SonarQube or SpotBugs. Three projects were natural choices: the SonarQube and SpotBugs projects are obviously relevant for applying their own tools; and the Eclipse IDE project is a long-standing Java project one of whose lead maintainers recently requested[3] help with fixing SonarQube issues. We selected the other twelve projects, following accepted best practices [29], among those that satisfy all of the following:

1) the project is registered with SonarCloud (a cloud service that can be used to run SonarQube on GitHub projects);
2) the project has at least 10 open issues related to violations of at least one of the 11 rules handled by SpongeBugs (see Table I);
3) the project has at least one fixed issue;
4) the project has at least 10 contributors;
5) the project has commit activity in the last three months.

TABLE II: The 15 projects we selected for evaluating SpongeBugs. For each project, the table report its DOMAIN, and data from its GitHub repository: the number of STARS, FORKS, CONTRIBUTORS, and the size in lines of code. Since Eclipse's GitHub repository is a secondary mirror of the main repository, the corresponding data may not reflect the project's latest state.

PROJECT	DOMAIN	STARS	FORKS	CONTRIBUTORS	LOC[a]
Eclipse IDE	IDE	72	94	218	743 K
SonarQube	Tool	3,700	1,045	91	500 K
SpotBugs	Tool	1,324	204	80	280 K
atomix	Framework	1,650	282	30	550 K
Ant Media Server	Server	682	878	16	43 K
cassandra-reaper	Tool	278	125	48	88.5 K
database-rider	Test	182	45	14	21 K
db-preservation-toolkit	Tool	26	8	10	377 K
ddf	Framework	95	170	131	2.5 M
DependencyCheck	Security	1,697	464	117	182 K
keanu	Math	136	31	22	145 K
matrix-android-sdk	Framework	170	91	96	61 K
mssql-jdbc	Driver	617	231	40	79 K
Payara	Server	680	206	66	1.95 M
primefaces	Framework	1,043	512	110	310 K

[a] Non-comment non-blank lines of code calculated from Java source files using cloc (https://github.com/AlDanial/cloc)

C. Submitting Pull Requests With Fixes Made by SpongeBugs

After running SpongeBugs on the 15 projects we selected, we tried to submit the fix suggestions it generated as pull requests (PRs) in the project repositories. Following suggestions to increase patch acceptability [30], before submitting any pull requests we approached the maintainers of each project through online channels (GitHub, Slack, maintainers' lists, or email) asking whether pull requests were welcome. (The only exception was SonarQube itself, since we did not think it was necessary to check that they are OK with addressing issues raised by their own tool.) When the circumstances allowed so, we were more specific about the content of our potential PRs. For example, in the case of mssql-jdbc, we also asked: "We noticed on the Coding Guidelines that new code should pass SonarQube rules. What about already committed code?", and mentioned that we found the project's dashboard on SonarCloud. However, we never mentioned that our fixes were generated automatically—but if the maintainers asked us whether a fix was automatic generated, we openly confirmed it. Interestingly, some developers also asked for a possible IDE integration of SpongeBugs as a plugin, which may indicate interest. We only submitted pull requests to the projects that replied with an answer that was not openly negative.

While the actual code patches in submitted pull requests were generated automatically by SpongeBugs, we manually added information to present them in a way that was accessible by human developers—following good practices that facilitate code reviews [31]. We paid special attention to four aspects: 1) change description, 2) change scope, 3) composite changes, and 4) nature of the change. To provide a good change description and clarify the scope of a change, we always mentioned which rule a patch is fixing—also providing a link to a textual description of the rule. In a few cases we wrote a more detailed description to better explain why the fix made sense, and how it followed recommendations issued by the project maintainers. For example, mssql-jdbc recommends to "try to create small alike changes that are easy to review"; we tried to follow this guideline in all projects. To keep our changes within a small scope, we separated fixes to violations of different rules into different pull requests; in case of fixes touching several different modules or files, we further partitioned them into separate pull requests per module or per file. This was straightforward thanks to the nature of the fix suggestions built by SpongeBugs: fixes are mostly independent, and one fix never spans multiple classes.

TABLE III: Responses to our inquiries about whether it is OK to submit a pull request to each project, and how many pull requests were eventually submitted and approved.

		PULL REQUESTS	
PROJECT	OK TO SUBMIT?	SUBMITTED	APPROVED
Eclipse IDE	Positive	9	9
SonarQube	–	1	1
SpotBugs	Neutral	1	1
atomix	Positive	2	2
Ant Media Server	Positive	3	3
database-rider	Positive	4	4
ddf	Positive	3	2
DependencyCheck	Neutral	1	1
keanu	Positive	3	0
mssql-jdbc	Positive	1	1
Payara	Positive	6	6
primefaces	Positive	4	4
cassandra-reaper	No reply	–	–
db-preservation-toolkit	No reply	–	–
matrix-android-sdk	No reply	–	–
	Total:	38	34

[3] https://twitter.com/vogella/status/1096088933144952832

We consider a pull request *approved* when reviewers indicate so in the GitHub interface. Although the vast majority of the approved PRs were merged, two of them were approved but not merged yet at the time of writing. Since merging depends on other aspects of the development process[4] that are independent of the correctness of a fix, we do not distinguish between pull requests that were approved and those that were approved but not merged yet.

The reviewing process may approve a patch with or without modifications. For each patch generated by SpongeBugs and approved we record whether it was approved with or without modifications.

V. Empirical Evaluation of SpongeBugs: Results and Discussion

The results of our empirical evaluation of SpongeBugs answer the three research questions presented in Section IV-A. For uniformity, all experiments target the 12 projects whose maintainers were accepting of pull requests fixing static analysis warnings (top portion of Table III).

A. RQ1: Applicability

To answer **RQ1** ("How widely applicable is SpongeBugs?"), we ran SonarQube on each project, counting the warnings triggering a violations of any of the 11 rules SpongeBugs handles. Then, we ran SpongeBugs and applied all its fix suggestions. Finally, we ran SonarQube again on the fixed project, counting how many warnings had disappeared. Table IV shows the results of these experiments. Overall, SpongeBugs's fix suggestions remove the source of 81% of the all warnings violating the rules we considered in this research.

These results justify our decision of focusing on a limited number of rules. In particular, the three rules (C3, C4, C9) with the lowest percentages of fixing are responsible for less than 6% of the triggered violations. In contrast, a small number of rules triggers the vast majority of violations, and SpongeBugs is extremely effective on these rules.

To elaborate, consider rule C9: (`Collections.EMPTY_LIST`, `EMPTY_MAP`, *and* `EMPTY_SET` *should not be used*). SpongeBugs only looks for return statements that violate this rule, since it is simpler to check whether the return type of a method is a collection, rather than to track local variable declarations—which might depend on other local variables and constants. Listing 5 shows a fix for rule C9.

A widely applicable kind of suggestion are those for violations of rule C1 (*String literals should not be duplicated*), shown in Listing 6, which SpongeBugs can successfully fix in 90% of the cases in our experiments. Generating automatically these suggestions is quite challenging. First, fixes to violations of rule C1 change multiple lines of code, and add a new constant. This requires to automatically come up with a descriptive name for the constant, based on the content of the string literal. The name must comply with Java's rules for identifiers (e.g., it cannot start with a digit). The name must also not clash with other constant and variable names that are in scope. SpongeBugs's fix suggestions can also detect whether there is already another string constant with the same value—reusing that instead of introducing a duplicate.

```
public Collection<Binding> getSequencesFor(ParameterizedCommand
      ↪ command) {
    ArrayList<Binding> triggers = bindingsByCommand.get(command);
- return (Collection<Binding>) (triggers == null ?
      ↪ Collections.EMPTY_LIST : triggers.clone());
+ return (Collection<Binding>) (triggers == null ?
      ↪ Collections.emptyList() : triggers.clone());
}
```

Listing 5: Fix suggestion for a violation of rule C9 (`Collections.EMPTY_LIST`, `EMPTY_MAP`, *and* `EMPTY_SET` *should not be used*) in project Eclipse IDE.

```
public class AccordionPanelRenderer extends CoreRenderer {

+ private static final String FUNCTION_PANEL = "function(panel)";

@@ -130,13 +133,13 @@ public class AccordionPanelRenderer extends
      ↪ CoreRenderer {
         if (acco.isDynamic()) {
             wb.attr("dynamic", true).attr("cache", acco.isCache());
         }

         wb.attr("multiple", multiple, false)
- .callback("onTabChange", "function(panel)",
      ↪ acco.getOnTabChange())
- .callback("onTabShow", "function(panel)", acco.getOnTabShow())
- .callback("onTabClose", "function(panel)", acco.getOnTabClose());
+ .callback("onTabChange", FUNCTION_PANEL, acco.getOnTabChange())
+ .callback("onTabShow", FUNCTION_PANEL, acco.getOnTabShow())
+ .callback("onTabClose", FUNCTION_PANEL, acco.getOnTabClose());
```

Listing 6: Fix suggestion for a violation of rule C1 (*String literals should not be duplicated*) in project primefaces.

We also highlight that our approach is able to perform distinct transformations in the same file and statement. Listing 7 shows the combination of a fix for rule C1 (*String literals should not be duplicated*) applied in conjunction with a fix for rule C5 (*Strings literals should be placed on the left side when checking for equality*).

```
public class DataTableRenderer extends DataRenderer {

+ private static final String BOTTOM = "bottom";

- if (hasPaginator &&
      ↪ !paginatorPosition.equalsIgnoreCase("bottom")) {
+ if (hasPaginator && !BOTTOM.equalsIgnoreCase(paginatorPosition))
      ↪ {
```

Listing 7: Fix suggestion for a violation of rules C1 and C5 in the same file and statement found in project primefaces.

Another encouraging result is the negligible number of fix suggestions that failed to compile: only two among all those generated by SpongeBugs. We attribute this low number to our approach of refining SpongeBugs's implementation with the support of a curated and growing suite of examples to test against. We also note that one of the two fix suggestions that didn't compile is likely a false positive (reported by SonarQube). On line 6 of Listing 8, the string literal `"format"` is replaced by the constant `OUTPUT_FORMAT` which is only accessible within class `CliParser` using its qualified name `ARGUMENT.OUTPUT_FORMAT`. However, SonarQube's warning

[4] One case is a pull request to Ant Media Server that was approved but violates the project's constraint that new code must be covered by tests.

TABLE IV: For each project and each rule checked by SonarQube, the table reports two numbers x/y: x is the number of warnings violating that rule found by SonarQube on the original project; y is the number of warnings that have disappeared after running SpongeBugs on the project and applying all its fix suggestions for the rule. The two rightmost columns summarize the data per project (TOTAL), and report the percentage of warnings that SpongeBugs successfully fixed (FIXED %). The two bottom rows summarize the data per rule in the same way.

PROJECT	B1	B2	C1	C2	C3	C4	C5	C6	C7	C8	C9	TOTAL	FIXED %
Eclipse IDE	44/5	13/13	214/199	4/4	11/8	18/3	189/176	15/11	–	159/97	102/32	769/548	71%
SonarQube	–	–	104/81	–	–	–	7/7	–	–	–	–	111/88	79%
SpotBugs	12/8	1/0	289/247	2/1	1/0	11/1	141/125	–	–	30/20	–	487/402	82%
atomix	1/1	–	57/55	–	–	–	9/9	–	–	1/0	2/0	70/65	93%
Ant Media Server	–	–	30/30	1/0	1/0	2/1	3/3	3/0	–	4/2	4/2	48/38	79%
database-rider	–	–	5/5	5/5	–	–	2/2	–	1/1	1/1	–	14/14	100%
ddf	1/0	–	104/97	–	1/0	–	88/86	–	1/1	45/33	8/1	248/218	88%
DependencyCheck	–	–	61/51	10/4	–	–	3/3	–	–	4/2	–	78/60	77%
keanu	1/1	–	–	–	–	–	4/4	–	12/11	5/3	–	22/19	86%
mssql-jdbc	4/1	–	314/274	14/1	–	7/0	58/58	2/0	–	14/11	–	413/345	83%
Payara	39/36	–	1,413/1,305	213/163	66/14	114/10	1,830/1,620	200/88	50/44	438/265	58/20	4,421/3,565	81%
primefaces	–	–	336/286	2/0	9/6	3/3	336/329	–	1/1	1/0	4/1	692/626	90%
TOTAL	102/52	14/13	2,927/2,630	251/178	89/28	155/18	2,670/2,422	220/99	65/58	702/434	178/56	7,373/5,988	–
FIXED %	51%	93%	90%	71%	31%	12%	91%	45%	89%	62%	31%	81%	–

does not have this information, as it just says: “Use already-defined constant OUTPUT_FORMAT instead of duplicating its value here”.

```
1   public final class CliParser {
2
3   - final Option outputFormat =
        ↪ Option.builder(ARGUMENT.OUTPUT_FORMAT_SHORT)
4   - .argName("format").hasArg().longOpt(ARGUMENT.OUTPUT_FORMAT)
5   + final Option outputFormat =
        ↪ Option.builder(ARGUMENT.OUTPUT_FORMAT_SHORT)
6   + .argName(OUTPUT_FORMAT).hasArg().longOpt(ARGUMENT.OUTPUT_FORMAT)
7
8       public static class ARGUMENT {
9           public static final String OUTPUT_FORMAT = "format";
10      }
11  }
```

Listing 8: Example of an incorrect fix due to a false positive violation of rule C1. Line 6 references constant OUTPUT_FORMAT which is not available as an unqualified name.

B. RQ2: Effectiveness and Acceptability

As discussed in Section Section IV-C, we only submitted pull requests after informally contacting project maintainers asking to express their interest in receiving fix suggestions for warnings reported by SATs. As shown in Table III, project maintainers were often quite welcoming of contributions with fixes for SATs violations, with 9 projects giving clearly positive answers to our informal inquiries. For example an Ant Media Server maintainer replied “Absolutely, you’re welcome to contribute. Please make your pull requests”. A couple of projects were not as enthusiastic but still available, such as a maintainer of DependencyCheck who answered “I’ll be honest that I obviously haven’t spent a lot of time looking at SonarCloud since it was setup…That being said – PRs are always welcome”. Even those that indicated less interest in pull requests ended up accepting most fix suggestions. This indicates that projects and maintainers that do use SATs are also inclined to find valuable the fix suggestions in response to their warnings. We received no reply from 3 projects, and hence we did not submit any pull request to them (and we excluded them from the rest of the evaluation).

In order to answer **RQ2** (“Does SpongeBugs generate fixes that are acceptable?”), we submitted 38 pull requests containing 920 fixes for the 12 projects that responded our question on whether fixes were of interest for the project. We did not submit pull requests with all fix suggestions (more than 5,000) since we did not want to overwhelm the maintainers. Instead, we sampled broadly (randomly in each project) while trying to select a diverse collection of fixes.

Overall, 34 pull requests were accepted, some after discussion and with some modifications. Table III breaks down this data by project. The non-accepted pull requests were: 3 in project keanu that were ignored; and 1 in project ddf where maintainers argued that the fixes were mostly stylistic. In terms of fixes, 775 (84%) of all 920 submitted fixes were accepted; 740 (95%) of them were accepted without modifications.

How to turn these measures into a precision measure depends on what we consider a *correct* fix: one that removes the source of warnings (precision nearly 100%, as only two fix suggestions were not working), one that was accepted in a pull request (precision: 84%), or one what was accepted without modifications (precision: $740/920 = 80\%$). Similarly, measures of recall depend on what we consider the total amount of relevant fixes.

An aspect that we did not anticipate is how policies about code coverage of *newly added* code may impact whether fix suggestions are accepted. At first we assumed our transformations would not trigger test coverage differences. While this holds true for single-line changes, it may not be the case for fixes that introduce a new statement, such as those for rule C1 (*String literals should not be duplicated*), rule C3 (*Strings should not be concatenated using + in a loop*), and some cases of rule C7 (entrySet() *should be iterated when* *both key and value are needed*). For example, the patch shown in Listing 9 was not accepted because the 2 added lines were not covered by any test. One pull request to Ant Media Server which included 97 fixes in 20 files was not accepted due to insufficient test coverage of some added statements.

```
public class TokenServiceTest {

+ private static final String STREAMID = "streamId";

- token.setStreamId("streamId");
+ token.setStreamId(STREAMID);
```

Listing 9: The lines added by this fix were flagged as not covered by any existing tests.

Sometimes a fix's context affects whether it is readily accepted. In particular, developers tend to insist that changes be applied so that the overall stylistic consistency of the whole codebase is preserved. Let's see two examples of this.

Listing 10 fixes three violations of rule C2; a reviewer asked if line 3 should be modified as well to use a character ’*’ instead of the single-character string "*":

> "Do you think that for consistency (and maybe another slight performance enhancement) this line should be changed as well?"

```
1  - if (pattern.indexOf("*") != 0 && pattern.indexOf("?") != 0 &&
         ↪ pattern.indexOf(".") != 0) {
2  + if (pattern.indexOf(’*’) != 0 && pattern.indexOf(’?’) != 0 &&
         ↪ pattern.indexOf(’.’) != 0) {
3      pattern = "*" + pattern;
4    }
```

Listing 10: Fix suggestion for a violation of rule C2 that introduces a stylistic inconsistency.

The pull request was accepted after a manual modification. Note that we do not count this as a modification to one of our fixes, as the modification was in a line of code other than the one we fixed.

Commenting on the suggested fix in Listing 11, a reviewer asked:

> "Although I got the idea and see the advantages on refactoring I think it makes the code less readable and in some cases look like the code lacks a standard, e.g one may ask why only this map entry is a constant?"

```
+ private static final String CASE_SENSITIVE_TABLE_NAMES =
      ↪ "caseSensitiveTableNames";

putIfAbsent(properties, "batchedStatements", false);
putIfAbsent(properties, "qualifiedTableNames", false);
- putIfAbsent(properties, "caseSensitiveTableNames", false)
+ putIfAbsent(properties, CASE_SENSITIVE_TABLE_NAMES, false);
putIfAbsent(properties, "batchSize", 100);
putIfAbsent(properties, "fetchSize", 100);
putIfAbsent(properties, "allowEmptyFields", false);
```

Listing 11: Fix suggestion for a violation of rule C3 that introduces a stylistic inconsistency.

This fix was declined in project database-rider, even though similar ones were accepted in other projects (such as Eclipse) after the other string literals were extracted as constants in a similar way.

Sometimes reviewers disagree on their opinion about pull requests. For instance, we received four diverging reviews from four distinct reviewers about one pull request containing two fixes for violations of rule C3 in project primefaces. One developer argued for rejecting the change, others for accepting the change with modifications (with each reviewer suggesting a different modification), and others still arguing against other reviewers' opinions. These are interesting cases that may deserve further research, especially because several projects require at least two reviewers to agree to approve a change.

Sometimes fixing a violation is not enough [5]. Developers may not be completely satisfied with the fix we generate, and may request changes. In some initial experiments, we received several similar modification requests for fix suggestions to violations of rule C7 (`entrySet()` *should be iterated when both key and value are needed*); in the end, we changed the way the fix is generated to accommodate the requests. For example, the fix in Listing 12 received the following feedback from maintainers of Eclipse:

> "For readability, please assign `entry.getKey()` to the `menuElement` variable"

```
- for (MMenuElement menuElement : new
      ↪ HashSet<>(modelToContribution.keySet())) {
-   if (menuElement instanceof MDynamicMenuContribution) {
+ for (Entry<MMenuElement, IContributionItem> entry :
      ↪ modelToContribution.entrySet()) {
+   if (entry.getKey() instanceof MDynamicMenuContribution) {
```

Listing 12: Fix suggestion for a violation of rule C7 generated in a preliminary version of SpongeBugs.

We received practically the same feedback from developers of Payara, which prompted us to modify how SpongeBugs generates fix suggestions for violations of rule C7. Listing 13 shows the fixed suggestion with the new template. All fixes generated using this refined fix template, which we used in the experiments reported in this paper, were accepted by the developers without modifications.

```
- for (MMenuElement menuElement : new
      ↪ HashSet<>(modelToContribution.keySet())) {
+ for (Entry<MMenuElement, IContributionItem> entry :
      ↪ modelToContribution.entrySet()) {
+ MMenuElement menuElement = entry.getKey();
    if (menuElement instanceof MDynamicMenuContribution) {
```

Listing 13: Fix suggestion for a violation of rule C7 generated in the final version of SpongeBugs.

Overall, SpongeBugs's fix suggestions were often found of high enough quality perceived to be accepted—many times without modifications. At the same time, developers may evaluate the acceptability of a fix suggestions within a broader context, which includes information and conventions that are not directly available to SpongeBugs or any other static code analyzer. Whether to enforce some rules may also depend on a developer's individual preferences; for example one developer remarked that fixes for rule C5 (*Strings literals should be placed on the left side when checking for equality*) are "*style preferences*". The fact that many of such fix suggestions were still accepted is additional evidence that SpongeBugs's approach was generally successful.

C. RQ3: Performance

To answer **RQ3** ("How efficient is SpongeBugs?"), we report some runtime performance measures of SpongeBugs on the projects. All experiments ran on a Windows 10 laptop with an Intel-i7 processor and 16 GB of RAM. We used Rascal's native benchmark library[5] to measure how long our transformations take to run on the projects considered in Table IV. Table V show the performance outcomes. For each of the measurements in this section, we follow recommendations on measuring performance [32]: we restart the laptop after each measurement, to avoid any startup performance bias (i.e., classes already loaded); and also provide summary descriptive statistics on 5 repeated runs of SpongeBugs.

TABLE V: Descriptive statistics summarizing 5 repeated runs of SpongeBugs. Time is measured in minutes.

		RUNNING TIME	
PROJECT	FILES ANALYZED	MEAN	ST. DEV.
Eclipse IDE	5,282	63.9 m	3.31 m
SonarQube	3,876	25 m	3.23 m
SpotBugs	2,564	26.4 m	2.05 m
Ant Media Server	228	3.8 m	0.15 m
atomix	1,228	8.1 m	0.23 m
database-rider	109	0.8 m	0.04 m
ddf	2,316	29.7 m	3.98 m
DependencyCheck	245	4.9 m	0.13 m
keanu	445	2.6 m	0.1 m
mssql-jdbc	158	14.2 m	0.27 m
Payara	8,156	141.5 m	5 m
primefaces	1,080	11.8 m	0.15 m

Project mssql-jdbc is an outlier due to its relatively low count of files analyzed with a long measured time. This is because its files tend to be large—multiple files with more than 1K lines. Larger files might imply more complex code, and therefore more complex ASTs, which consequently leads to more rule applications. To explore this hypothesis we ran our transformations on a subset of these larger files. As seen in Table VI, five larger files are responsible for more than 5 minutes of running time. Additionally, file `dtv` takes, on average, almost 40 seconds (56%) longer than `SQLServerBulkCopy`; even though they have roughly the same size, file `dtv` has numerous class declarations and methods with more than 300 lines, containing multiple `switch`, `if`, and `try`/`catch` statements.

Generating some fix suggestions takes longer than others. We investigated this aspect more closely in SpotBugs, as it includes more than a thousand files containing multiple test cases for the rules it implements. Excluding test files in `src/test/java` does not work for SpotBugs, which puts tests in another location, thus greatly increasing the amount of code that SpongeBugs analyzes. SpongeBugs takes considerably longer to run on rules B1, B2/C6, and C1. The main reason is that step 1 in these rules raises several false positives, which are then filtered out by the more computationally expensive step 2 (see Section III-B). For example, step 1's filtering for rule B1 (*Strings and boxed types should be compared using* `equals()`), shown in Listing 14, is not very restrictive. One can imagine that several files have a reference to a `String` (covered by `hasWrapper()`) and also use `==` or `!=` for comparison operators. Contrast this to step 1's filtering for rule C9 (`Collections.EMPTY_LIST`*... should not be used*), shown in Listing 15, which is much more restrictive; as a result SpongeBugs runs in under 5 seconds for rule C9.

TABLE VI: Descriptive statistics summarizing 5 repeated runs of SpongeBugs on the 5 largest files in projects mssql-jdbc. Time is measured in seconds.

		RUNNING TIME	
FILE	LOC	MEAN	ST. DEV.
SQLServerConnection	4,428	116 s	2.4 s
SQLServerResultSet	3,858	99 s	3.8 s
dtv	2,823	106 s	3.1 s
SQLServerBulkCopy	2,529	68 s	5 s
SQLServerPreparedStatement	2,285	64 s	4.8 s

```
return hasWrapper(javaFileContent) &&
    ↪ hasEqualityOperator(javaFileContent);
```

Listing 14: Violation textual pattern in the implementation of rule B1

```
return findFirst(javaFileContent, "Collections.EMPTY") != -1;
```

Listing 15: Violation textual pattern in the implementation of rule C9

Overall, we found that SpongeBugs's approach to fix warnings of SATs is scalable on projects of realistic size. SpongeBugs could be reimplemented to run much faster if it directly used the output of static code analysis tools, which indicate precise locations of violations. While we preferred to make SpongeBugs's implementation self contained to decouple from the details of each specific SAT, we plan to explore other optimizations in future work.

D. Additional Findings

In this section we summarize findings we collected based on the feedback given by reviews of our pull requests.

Some fixes are accepted without modifications. Some fixes are uniformly accepted without modifications. For example those for rule C2 (*String function use should be optimized for single characters*), which bring performance benefits and only involve minor modifications (as shown in Listing 16: change string to character).

```
- int otherPos = myStr.lastIndexOf("r");
+ int otherPos = myStr.lastIndexOf('r');
```

Listing 16: Example of a fix for a violation of rule C2.

SAT adherence is stricter in new code. Some projects require SAT compliance only on new pull requests. This means that previously committed code represent accepted technical debt.

[5] http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Libraries/util/Benchmark/benchmark/benchmark.html

For instance, mssql-jdbc's contribution rules state that "New developed code should pass SonarQube rules". A SpotBugs maintainer also said "I personally don't check it so seriously. I use SonarCloud to prevent from adding more problems in new PR". Some use SonarCloud not only for identifying violations, but for test coverage checks.

Fixing violations as a contribution to open source. Almost all the responses to our questions about submitting fixes were welcoming—along the lines of *help is always welcome*. Since one does not need a deep understanding of a project domain to fix several SATs' rules, and the corresponding fixes are generally easy to review, submitting patches to fix violations is an approachable way of contributing to open source development.

Fixing violations induce other clean-code activities. Sometimes developers requested modifications that were not the target of our fixes. While our transformations strictly resolved the issue raised by static analysis, developers were aware of the code as a whole and requested modifications to preserve and improve code quality.

Fixing issues promotes discussion. While some fixes were accepted "as is", others required substantial discussion. We already mentioned a pull request for primefaces that was intensely debated by four maintainers. A maintainer even drilled down on some Java Virtual Machine details that were relevant to the same discussion. Developers are much more inclined to give feedback when it is about code they write and maintain.

VI. Limitations and Threats to Validity

Some of SpongeBugs's transformations may violate a project's stylistic guidelines [16]. As an example, project primefaces uses a rule[6] about the order of variable declarations within a class that requires that private constants (`private static final`) be defined after public constants. SpongeBugs's fixes for rule C1 (*String literals should not be duplicated*) may violate this stylistic rule, since constants are added as the first declaration in the class. Another example of stylistic rule that SpongeBugs may violate is one about empty lines between statements[7]. Overall, these limitations appear minor, and it should not be difficult to tweak SpongeBugs's implementation so that it fixes comply with additional stylistic rules.

Static code analysis tools are a natural target for fix suggestion generation, as one can automatically check whether a transformation removes the source of violation by rerunning the static analyzer [24]. In the case of SonarCloud, which runs in the cloud, the appeal of automatically generating fixes is even greater, as any technique can be easily scaled to benefit a huge numbers of users. We checked the applicability of SpongeBugs on hundreds of different examples, but there remain cases where our approach fails to generate a suitable fix suggestions. There are two reasons when this happens:

1) *Implementation limitations*. One current limitation of SpongeBugs is that its code analysis is restricted to a single file at a time, so it cannot generate fixes that depend on information in other files. Another limitation is that SpongeBugs does not not analyze methods' return types.
2) *Restricted fix templates*. While manually designed templates can be effective, the effort to implement them can be prohibitive [4]. With this in mind, we deliberately avoided implementing templates that were too hard to implement relative to how often they would have been useful.

SpongeBugs's current implementation does not rely on the output of SATs. This introduces some occasional inconsistencies. as well as cases where SpongeBugs cannot process a violation reported by a SAT. An example, discussed above, is rule C9: SpongeBugs only considers violation of the rule that involve a return statement. These limitations of SpongeBugs are not fundamental, but reflect trade-offs between efficiency of its implementation and generality of the technique it implements. We only ran SpongeBugs on projects that normally used SonarQube or SpotBugs. Even though SpongeBugs is likely to be useful also on general projects, we leave a more extensive experimental evaluation to future work.

VII. Conclusions

In this work we introduced a new approach and a tool (SpongeBugs) that finds and repairs violations of rules checked by static code analysis tools such as SonarQube, FindBugs, and SpotBugs. We designed SpongeBugs to deal with rule violations that are frequently fixed in both private and open-source projects. We assessed SpongeBugs by running it on 12 popular open source projects, and submitted a large portion (total of 920) of the fixes it generated as pull requests in the projects. Overall, project maintainers accepted 775 (84%) of those fixes—most of them (95%) without any modifications. We also assessed SpongeBugs's performance, showing that it scales to large projects (under 10 minutes on projects as large as half a million LOC). These results suggest that SpongeBugs can be an effective approach to help programmers fix warnings issued by static code analysis tools—thus contributing to increasing the usability of these tools and, in turn, the overall quality of software systems.

For *future work*, we envision using SpongeBugs to prevent violations to static code analysis rules from happening in the first place. One way to achieve this is by making its functionality available as an IDE plugin, which would help developers in real time. Another approach is integrating SpongeBugs as a tool in a continuous integration toolchain. We plan to pursue these directions in future work.

Acknowledgments. We thank the maintainers for reviewing our patches and the reviewers for their helpful comments. This was is partially supported by CNPq (#406308/2016-0).

[6] http://checkstyle.sourceforge.net/apidocs/com/puppycrawl/tools/checkstyle/checks/coding/DeclarationOrderCheck.html

[7] http://checkstyle.sourceforge.net/config_whitespace.html#EmptyLineSeparator

References

[1] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed?: A closer look at realistic usage of SonarQube," in *Proceedings of the 27th International Conference on Program Comprehension*, ser. ICPC '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 209–219. [Online]. Available: https://doi.org/10.1109/ICPC.2019.00040

[2] 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*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 317–328. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238213

[3] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486877

[4] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandè, "AVATAR: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2019, pp. 1–12.

[5] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill, "From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 211–221.

[6] K. F. Tómasdóttir, M. Aniche, and A. van Deursen, "Why and how JavaScript developers use linters," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 578–589.

[7] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 470–481.

[8] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Sep. 2008.

[9] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin, Heidelberg: Springer-Verlag, 1999.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[11] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2009.

[12] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 480–491. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250789

[13] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 265–266. [Online]. Available: http://doi.acm.org/10.1145/2892208.2892235

[14] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. [Online]. Available: https://doi.org/10.1109/msr.2017.54

[15] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. D. Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. [Online]. Available: https://doi.org/10.1109/msr.2017.2

[16] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon, "Mining fix patterns for FindBugs violations," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[17] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou, "How do developers fix issues and pay back technical debt in the Apache ecosystem?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 153–163.

[18] 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 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[19] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible Java compiler," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, Sep. 2012, pp. 14–23.

[20] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 315–324. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806847

[21] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2fix: Automatically generating bug fixes from bug reports," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, March 2013, pp. 282–291.

[22] R. Rolim, G. Soares, R. Gheyi, and L. D'Antoni, "Learning quick fixes from code repositories," *CoRR*, vol. abs/1803.03806, 2018. [Online]. Available: http://arxiv.org/abs/1803.03806

[23] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Software Eng.*, vol. 45, no. 1, pp. 34–67, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2017.2755013

[24] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3105906

[25] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 672–681.

[26] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Why and how Java developers break APIs," in *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 255–265.

[27] P. Klint, T. Van Der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009, pp. 168–177.

[28] R. Dantas, A. Carvalho, D. Marcílio, L. Fantin, U. Silva, W. Lucas, and R. Bonifácio, "Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate Java programs," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 497–501.

[29] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, sep 2015. [Online]. Available: https://doi.org/10.1007/s10664-015-9393-5

[30] Y. Tao, D. Han, and S. Kim, "Writing acceptable patches: An empirical study of open source project patches," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 271–280.

[31] A. Ram, A. A. Sawant, M. Castelluccio, and A. Bacchelli, "What makes a code change easier to review: an empirical investigation on code change reviewability," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 201–212. [Online]. Available: https://doi.org/10.1145/3236024.3236080

[32] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 57–76. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297033