SpongeBugs: Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings

Diego Marcilio[a], Carlo A. Furia[a], Rodrigo Bonifácio[b], Gustavo Pinto[c]

[a] *USI Università della Svizzera italiana, Lugano, Switzerland*
[b] *University of Brasília, Brasília, Brazil*
[c] *Federal University of Pará, Belém, Brazil*

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 are maintaining. To this end, we implemented SpongeBugs, 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. Based on the suggestions provided by SpongeBugs, we submitted 38 pull requests, including 946 fixes generated automatically by our technique for various open-source Java projects, including Eclipse UI—a core component of the Eclipse IDE—and both SonarQube and SpotBugs. Project maintainers accepted 87% of our fix suggestions (97% of them without any modifications). We further evaluated the applicability of our technique on software written by students and on a curated collection of bugs. All results indicate that our approach to generating fix suggestions is feasible, flexible, and can help increase the applicability of static code analysis tools.

Email addresses: diego.marcilio@usi.ch (Diego Marcilio), rbonifacio@unb.br (Rodrigo Bonifácio), gpinto@ufpa.br (Gustavo Pinto)
URL: bugcounting.net (Carlo A. Furia)

1. 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 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 3, 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 suggests and presents a fix to the user.

By construction, the fixes SpongeBugs suggests remove the origin of a rule's violation, but the maintainers still have 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 5 presents the result of an empirical evaluation where we applied it to 12 open-source Java projects, and submitted 946 fix suggestions as pull requests to the projects. Project maintainers accepted 825 (87%) fix suggestions—97% 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 (1.2 minutes to process 1,000 lines of code on average). SpongeBugs is also *accurate*, as it generates false positives (spurious rule violations) in less than 0.6% of all reported violations. We actually found several cases where SpongeBugs correctly detected cases of rule violations that were missed by SonarQube (Section 5.1.1).

To further demonstrate SpongeBugs's versatility, Section 5 also discusses how SpongeBugs complements program repair tools (e.g., AVATAR [4]) and how it performs on software whose main contributors are non-professionals (i.e., students). With few exceptions—which we discuss throughout Section 5 to inform further progress in this line of work—SpongeBugs worked as intended: it provides sound, easy to apply suggestions to fix static rule violations.

The work reported in this paper is part of a large body of research (see Section 2) 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 6 discusses the main limitations of SpongeBugs's approach, and Section 7 outlines directions for further progress in this line of work.

Extended Version. This journal article extends a previous conference publication [9] by significantly expanding the empirical evaluation of SpongeBugs with: (1) an extended and updated evaluation on SpongeBugs' applicability in Section 5.1, using a revised implementation with numerous bug fixes; (2) a detailed analysis of accuracy (false positives and false negatives, in Section 5.1.2 and Section 5.1.3); (3) a smaller-scale evaluation involving student projects (Section 5.1.4); (4) an experimental assessment (Section 5.3.2) of how SpongeBugs's three-stage process trade-offs a modicum of precision for markedly better performance; and (5) an experimental comparison with the Defects4J curated collection of real-world Java bugs (Section 5.4).

2. Background and Related Work

Static analysis techniques reason about program behavior *statically*, that is without running the program [10]. 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 optimization [11, 12] as well as performance problems and other kinds of vulnerabilities [13, 14].

2.1. 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`—that is, one should write `String s = "SpongeBugs"` instead of instantiating new `String` objects in the heap with `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. [15] 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. [16] 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. [17] 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. [18] performed a similar analysis for SonarQube rules, revealing that a small number of all rules account 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 projects' 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.

2.2. Automatic Fix Suggestions

Aftandilian et al. [19] presented *error-prone*: an extension of the OpenJDK compiler that detects potential bugs and recommends fixes at compile time, based on patterns that are similar to some of those used by SATs like FindBugs. More precisely, *error-prone* targets bugs that admit simple fixes, which usually "should change the behavior of the program" [19]; for example, adding a null check to a method to avoid null-deferencing errors. In contrast, SpongeBugs targets violations of rules that are primarily about style and design; therefore, its fixes are mostly behavior-preserving.

Other approaches learn transformations from examples, using sets of bug fixes [20], bug reports [21], or source-code editing patterns [22]. Commercial project CodeLingo[1] is also based on learning, which it performs while monitoring and assisting the code reviewing process. We directly implemented SpongeBugs' transformations based on our expertise and the standard recommendations for fixing warnings from static analysis tools. SpongeBugs cannot learn new fixing rules at the moment, but this is an interesting direction for improving its capabilities.

Coccinelle [23] is an approach to automatically apply code-transformation templates to large code bases. Unlike SpongeBugs, which works automatically on a predefined set of templates, Coccinelle supports the user *definition* of modular, complex patterns that capture semantic information. Its implementation represents patterns as CTL formulas and expresses their application as a model-checking problem on the program's control-flow graph. Coccinelle's main application has been to support *evolution* of large codebases: when an API or a language feature changes, manual refactoring is error-prone because there's a high risk of introducing inconsistencies or other mistakes. Using Coccinelle, it has been possible to support evolution of the Linux kernel (written in C) in a much more robust and streamlined way.

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, 17]. These studies focus on *behavioral* bugs such as those in Defects4J [24]—a curated collection of hundreds of 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, 17, 18]. This focus helps SpongeBugs achieve high precision and scalability, as well as be practically useful.

[1] `https://codelingo.it`

2.3. Automated Program Repair

Behavioral bugs are also the primary focus of techniques for "automated program repair", a research area that has grown considerably in the last decade [25]. The most popular approaches to automated program repair are mainly driven by dynamic analysis (i.e., tests) [26, 25] and target generic bugs. In contrast, SpongeBugs's approach is based on static code-transformation techniques, which make it simpler and of more limited scope but also more easily and widely applicable.

Using dynamic analysis, bug detection is straightforward—a failing test is a bug—whereas fault localization and fix synthesis are the challenging parts. Therefore, early work in automated program repair [27, 28, 29] introduced the idea of expressing the search for a fix as a search problem, which can be tackled using genetic programming or other kinds of general-purpose search algorithms. These works pioneered key concepts and problems in automated program repair, but were still quite limited in the number of correct fixes (identical to those written by a programmer) they could produce [30]. More recently, approaches such as SapFix [31] revisited several of the same techniques and concepts as the early work, in a way that they have become applicable on a larger scale with higher precision: SapFix has been deployed in a continuous integration environment at Facebook, where it was able to generate fixes to several applications.

Several recently developed automated program repair techniques are based on *machine learning*, which they use to automatically build source-code modification patterns that are similar to those used by programmers in fixing bugs. The key challenge in patch mining [32] is how to summarize changes in a way that is sufficiently fine-grained to capture correct fixes, but also sufficiently general that it is applicable to different instances of the same kind of bug. Getafix [33] uses clustering algorithms to this effect. Prophet [34] builds abstractions of conditionals that lead to failed tests, and modifies them, according to a probabilistic model that ranks possible fixes, using techniques based on expression synthesis. Genesis [35] applies the idea of patch mining (inferring fixing patterns that can be used to generate new fixes) to program repair, focusing on null-dereferencing bugs. In contrast, SpongeBugs does not use any learning: on the one hand, this limits its flexibility and the range of issues it can address; on the other hand, this makes it much faster (machine learning is typically time consuming) and precise in its application domain.

Liu et al.'s study [4] presented the AVATAR automatic program repair system. AVATAR recommends code changes based on the output of SAT tools, and hence it is somewhat comparable to SpongeBugs. 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. Despite these high-level similarities, AVATAR's and SpongeBugs' approaches are quite different, since their scopes are mostly *complementary* in the kind of design trade-offs they target. AVATAR uses the output of SATs *indirectly* to help its fault localization process, and to guide mining programmer-written fixes for patterns. Getafix [33] (which

we mentioned above among the techniques based on learning) uses a similar approach, focusing on fixes that solve issues reported by SATs. The idea is that if a human-written fix removes a SAT warning, it may be indicative of a useful bug-fixing transformation, and hence AVATAR or Getafix abstract it into a fixing pattern that can be used on new bugs. In contrast, 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.

Bavishi et al. [36] proposed PHOENIX, a system that learns fix patches for violations detected by SATs. Similarly to AVATAR, PHOENIX primarily uses SATs *offline*: in the experiments [36], PHOENIX ran FindBugs on revisions of 517 selected GitHub projects to identify programmer-written fixes of violations and to learn fix patterns from them. Based on them, PHOENIX built a large number of fix suggestions for faults in various projects; manually-selected fix suggestions among them were included in 19 pull requests that were accepted by the projects' developers. Some of these suggestions fix violations of rules that SpongeBugs also targets (see Section 3.1; the rules in question are B1, C3, and C7).

3. SpongeBugs: Approach and Implementation

SpongeBugs provides fix suggestions for violations of selected rules that are checked by SonarQube and SpotBugs. Section 3.1 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 3.2. This approach has advantages but also limitations in its applicability, which we discuss in Section 3.3.

3.1. 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 [37]. 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, 17].

We collected and analyzed the publicly available datasets from three previous studies that explored developer behavior in response to output from SonarQube [1, 18] and FindBugs [17]. These three studies combined analyzed 1,033 projects with 18 million violations of over 400 rules. We initially selected the top 50% most frequently fixed rules in each of the three datasets, 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`.[2] 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—potentially introducing a breaking change [38]. 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 1. 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 identifiers[3].

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 somewhat limited practical usefulness, most of our rules are code smells consistently with how we selected them.

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 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.

Behavioral Changes. May the refactorings needed to comply with the rules in Table 1 change program *behavior*? The majority of rules require purely stylistic changes that cannot affect meaning. Complying with rules B1, C5, C6, and C7, however, might introduce behavioral changes if the original code relies on *reference equality* or a specific allocation of objects.

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

[3] `https://rules.sonarsource.com/java`

Table 1: 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	SonarQube ID	Rule description
B1	S4973	Strings and boxed types should be compared using `equals()`
B2	S2111	`BigDecimal(double)` should not be used
C1	S1192	String literals should not be duplicated
C2	S3027	String functions use should be optimized for single characters
C3	S1643	Strings should not be concatenated using `+` in a loop
C4	S2130	Parsing should be used to convert strings to primitive types
C5	S1132	Strings literals should be placed on the left-hand side when checking for equality
C6	S2129	Constructors should not be used to instantiate `String`, `BigInteger`, `BigDecimal`, and primitive wrapper classes
C7	S2864	`entrySet()` should be iterated when both key and value are needed
C8	S1155	`Collection.isEmpty()` should be used to test for emptiness
C9	S1596	`Collections.EMPTY_LIST`, `EMPTY_MAP`, and `EMPTY_SET` should not be used

For example, rule B1 suggests to use object equality instead of reference equality to compare strings. This recommendation implicitly assumes that a program does not rely on a behavior where different string objects representing the same sequence of characters should not be considered equal. Indirectly, it also implicitly assumes that all string variables are not null—since you cannot compare two null references using `equals()`. Similar issues may affect the behavior when enforcing rules C5 (no null strings) and C6 (different objects with the same content). Rule C7 is a bit different: here the problem is that `entrySet()` may return elements in a different *order* than `keySet()`—even though an implementation should not generally rely on the order of elements in a *set*.

We remark that the situations where complying with a rule may introduce a behavioral change are still very specific and uncommon to occur in programs that follow widely accepted practices. However, we found a few instances where this happens in the evaluation of SpongeBugs (Section 5.2).

3.2. 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 [39], a domain-specific language for source code analysis and manipulation. Rascal facilitates several common meta-programming tasks, including implementing a first-class visitor language constructor, advanced pattern matching based on concrete syntax, and defining templates for code generation. We used Rascal's latest Java grammar [40], which targets Java 8; thus, our evaluation (Section 5) is limited to Java projects that can be built using this version of the language.

Let's illustrate how SpongeBugs's three steps work for rule C2 (*String functions use should be optimized for single characters*). Step 1 performs a fast, but potentially imprecise, textual search that is based on some necessary conditions for a rule to be triggered. For rule C2, step 1 looks for files that have a method call to either `lastIndexOf()` or `indexOf()`—as shown in Listing 2.

```
boolean shouldContinueWithASTAnalysis(loc fileLoc) {
   javaFileContent = readFile(fileLoc);
   return findFirst(javaFileContent, ".lastIndexOf(\"") != -1 ||
      findFirst(javaFileContent, ".indexOf(\"") != -1;
```

Listing 2: Excerpt of the implementation of step 1 for rule C2: find textual patterns that might represent a violation of rule C2.

Step 1 may report false positives: for rule C2, the call to `lastIndexOf()` or `indexOf()` may not actually involve an instance of a `String`, or the argument of the function might not be a single character. 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 containing rule violations. In our example of rule C2, step 2 checks that the target of the possibly offending call to `indexOf()` is indeed of type `String` and that the argument is a single character—as shown in Listing 3.

```
case (MethodInvocation)
  `<Primary varName>.<TypeArguments? ts>indexOf(<ArgumentList? args>)`: {
    if (isVarAString(mdl, varName) && isSingleArgOfInterest(args)) {
```

Listing 3: Excerpt of the implementation of step 2 for rule C2: full AST search for rule violations.

If step 2 returns a positive match, step 3 executes and generates a patch to fix the rule violation. Step 3's generation is entirely based on *code-transformation templates* defined on the concrete program syntax and corresponding to the AST matched in step 2. In other words, abstract syntax is used for detection in step 2, whereas concrete syntax is used for modification in step 3—which makes it possible to apply the transformations directly on the original source code. For rule C2, step 3's template is straightforward: replace a single character string (double quotes) with a character (single quotes)—its implementation is in Listing 4. (Steps 2 and 3 for rule C2 also handle other patterns not shown in this example for brevity.)

```
argAsChar = parseSingleCharStringToCharAsArgumentList(argsStr);
insert (MethodInvocation) `<Primary varName>.<TypeArguments? ts>indexOf(<ArgumentList
    ↪ argAsChar>)`;
```

Listing 4: Excerpt of the implementation of step 3 for rule C2: instantiate the fix templates corresponding to the violated rule.

Steps 1 and 2 achieve different trade-offs between speed and precision. Step 1 is very fast but also sensitive to differences in layout that should be immaterial (for example, presence of extra newline characters). In contrast, step 2 is considerably slower but robust against textual changes that don't affect the program semantics. The experiments in Section 5.3.2 demonstrate how the combination of these two steps achieves a fast performance without significantly compromising overall precision.

3.3. Limitations of SpongeBugs

SpongeBugs's current implementation has some restrictions of applicability. A few of them are deliberate design choices that help make detection more precise or more efficient; others are limitations of the current implementation. Section 5.1.3 shows concrete examples of code that incurred these limitations.

Local analysis. SpongeBugs's analysis is strictly local to each method: if a method `m` calls another method `n`, `m`'s analysis has no information about `n`'s effects and results other than its local calling context. This limitation may cause false negatives in the detection of all rules.

Contextual restrictions. SpongeBugs analyzes some rules with additional restrictions on their context of applicability. For example, it does not check rules that may be violated inside *lambda expressions*, since supporting their semantics would have significantly complicated SpongeBugs's analysis just to handle a few additional cases.

Typing information. SpongeBugs's analysis has access to typing information that is *incomplete*. In particular, it keeps track of the type of variables and expressions, but does not reconstruct how user-defined types are related by inheritance—which may prevent it from detecting the violation of rule C8 in some cases.

Toolchain. The remaining limitations of SpongeBugs follow from limitations of the tools we used to build it. In particular, Rascal's Java 8 grammar is not complete. While implementing SpongeBugs, we extended the Rascal grammar[4] to cover some of the missing cases, but a few language features remain beyond its capabilities.

4. Empirical Evaluation of SpongeBugs: Experimental Design

The overall goal of this research is suggesting fixes to warnings generated by static code analysis tools. Section 4.1 presents the research questions we answer in this empirical study, which targets:

- Fifteen open-source projects selected using the criteria we present in Section 4.2;

[4]Our extension is now available in Rascal's repository.

- Five student projects developed as part of software engineering courses;
- Defects4J: a curated collection of faulty Java programs, widely used to evaluate automated program repair (see Section 2) tools.

All data created in this study and our tool implementation are available:

`https://github.com/dvmarcilio/spongebugs`

4.1. Research Questions

The empirical evaluation of SpongeBugs, whose results are described in Section 5, addresses the following research questions, which are based on the original motivation behind this work: automatically providing fix suggestions that helps to 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. We also analyze cases in which SpongeBugs's detection of rule violations are not accurate.

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 and passed the projects' test suites.

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

RQ4. How does SpongeBugs work on code with *semantic* bugs?
The fourth research question runs SpongeBugs on Defects4J, a curated collection of semantic bugs in real-world Java program; while SpongeBugs is not designed to fix these kinds of behavioral bugs, it is interesting to see how its heuristics interact with code that has different kinds of errors.

4.2. Selecting Projects for the Evaluation

The evaluation of SpongeBugs uses different Java projects, which we describe in the following subsections.

4.2.1. Open-Source Projects

The bulk of the evaluation—addressing RQ1, RQ2, and RQ3—targets large open-source Java projects, which provide a real-world usage scenario. 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 Platform UI[5] project—a core component of the Eclipse IDE—is a long-standing Java project one of whose lead maintainers requested[6] help with

[5] For brevity, we call it Eclipse UI in the rest of the paper.
[6] `https://twitter.com/vogella/status/1096088933144952832`

fixing SonarQube issues. We selected the other twelve projects, following accepted best practices [41], 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 1);
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 2: The 15 projects we selected for evaluating SpongeBugs. For each project, the table reports its DOMAIN, and data from its GitHub repository: the number of STARS, FORKS, CONTRIBUTORS, and the size in non-blank non-comment 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 UI	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-blank non-comment lines of code calculated from Java source files using `cloc` (`https://github.com/AlDanial/cloc`)

4.2.2. Student Projects

The effort devoted to improving code quality may be different in projects developed by students as opposed to large open-source projects such as those that we identified in Section 4.2.1. SpongeBugs can still be effective on both kinds of projects, but the impact and scope of its suggestions may differ.

To investigate this aspect, and to demonstrate SpongeBugs's applicability on heterogeneous software, we considered 5 student projects from courses taught at USI (the first and second author's affiliation)[7] in the latest two years. The student projects were developed before our analysis with SpongeBugs and independent of it. Table 3 summarizes the projects' characteristics. We selected these projects because they were (mainly) written in Java, of substantial size, and developed by students with experience (that is, non-beginners).

Table 3: The student projects we analyzed using SpongeBugs. For each project, the table reports whether it was developed by undergraduates or master's students (LEVEL); the number of students working on the project (# STUDENTS); its size in non-blank non-comment lines of code (LOC); and whether students already analyzed it using SonarQube (SQ?).

PROJECT	LEVEL	# STUDENTS	LOC	SQ?
Project 1	U	12	6.8 K	Yes
Project 2	U	13	5.2 K	Yes
Project 3	M	7	2.5 K	Yes
Project 4	M	8	2.4 K	Yes
Project 5	M	4	3.6 K	No

Projects 1 and 2 were developed by undergraduate students[8] of "Software Atelier 4", a software engineering project course where groups of about 12 students work to develop an application following realistic best practices of code development and team coordination. The projects included a front-end written in JavaScript, which we ignored for the purpose of evaluating SpongeBugs. The projects required students to use SonarQube to spot potential problems in their code, and to monitor test coverage.

Projects 3, 4, and 5 were developed by master's students[9] of "Software Analytics", a course about using and building tools to monitor software development artifacts and their evolution; each project was developed by a group of 4–8 students. As for projects 1 and 2, we only considered the project modules that are written in Java. Project 5 did not require students to use SonarQube, whereas projects 3 and 4 did.

4.2.3. Curated Collection of Bugs

As we discussed in detail in Section 3, SpongeBugs's targets SAT rules that are *syntactic* and *modular*. This is a deliberate restriction of SpongeBugs's applicability, but also one that makes it very effective in its domain (as we demonstrate empirically in Section 5).

In contrast, the related work on "automated program repair" (see also Section 2) typically targets the more diverse category of *semantic* (*behavioral*) bugs,

[7] However, the courses were not taught by the authors.
[8] The undergraduate program lasts 3 years.
[9] The master's program lasts 2 years.

which include any program behavior that deviates from the intended one. Defects4J has become a popular benchmark to evaluate the performance of such tools for Java. It is a curated collection of (mostly semantic) bugs found in open-source Java projects. Each bug in Defects4J comes with some tests that trigger it, as well with a programmer-written patch that corrects the behavior as intended. Table 4 lists the basic characteristics of the code included in Defects4J (version 1.5.0).

Table 4: A summary of the code included in Defects4J, grouped by the Java project it comes from. For each group, we list the overall size in non-blank non-comment lines of code, the number of available tests, and the number of bugs triggered by the tests.

PROJECT		LOC	TESTS	BUGS
Chart	JFreechart	96 K	2,278	26
Closure	Closure Compiler	90 K	8,300	176
Lang	Apache Commons-Lang	22 K	2,341	65
Math	Apache Commons-Math	84 K	3,619	106
Mockito	Mockito Framework	11 K	1,546	38
Time	Joda-Time	30 K	4,186	27
TOTAL		330.5 K	22,270	438

In order to get a better idea of the difference between syntactic and semantic bugs, we ran SpongeBugs on all 438 Defects4J bugs. Precisely, we ran SpongeBugs on the *buggy* version of each program in Defects4J. While we do not expect SpongeBugs to repair the bugs (SpongeBugs targets violations of different rules), this experiment can shed light on the interaction between the syntactic modifications introduced by SpongeBugs and the semantic behavior of programs (as checked by the tests in Defects4J). In other words, we would like to ascertain that SpongeBugs's suggestions do not adversely interfere with the intended program behavior, and they can be applied even when the code is buggy (as it often is).

4.3. Submitting Pull Requests With Fixes Made by SpongeBugs

After running SpongeBugs on the 15 open-source projects listed in Table 2, we submitted the fix suggestions it generated as pull requests (PRs) in the project repositories. Following suggestions to increase patch acceptability [42], 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 automatically generated, we openly confirmed it.

We only submitted pull requests to projects that replied with an answer that was not openly negative. As shown in Table 5, most answers were openly positive—some developers even asked for a possible IDE integration of SpongeBugs as a plugin, which may indicate interest. The single clearly negative response pointed out as reason that the project (matrix-android-sdk) was being discontinued.

Table 5: Responses to our inquiries about whether it is OK TO SUBMIT a pull request to each project. For projects that accepted them, we report how many pull requests were SUBMITTED and eventually APPROVED, and the average time (in hours) from submission to DECISION (accept or reject a pull request), and from submission to MERGE (an approved pull request, excluding one pull request of Ant Media Server that was approved but not merged).

		PULL REQUESTS		AVERAGE TIME [h]	
PROJECT	OK TO SUBMIT?	SUBMITTED	APPROVED	DECISION	MERGE
Eclipse UI	Positive	9	9	82.6	88.0
SonarQube	–	1	1	25.7	27.0
SpotBugs	Neutral	1	1	2.1	5.1
atomix	Positive	2	2	299.3	487.5
Ant Media Server	Positive	3	3	18.3	18.3
database-rider	Positive	4	4	14.7	14.7
ddf	Positive	3	2	55.9	4168.0
DependencyCheck	Neutral	1	1	10.2	10.2
keanu	Positive	3	0	–	–
mssql-jdbc	Positive	1	1	650.4	650.7
Payara	Positive	6	6	9.0	170.1
primefaces	Positive	4	4	0.3	6.9
cassandra-reaper	No reply	–	–	–	–
db-preservation-toolkit	No reply	–	–	–	–
matrix-android-sdk	Negative	–	–	–	–
	Total:	38	34		

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 [43]. 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 was fixing—also providing a link to SonarQube's official 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 of 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, one fix never spans multiple classes, and each rule can be analyzed independent of the others.

Overall, our manual effort to compile the pull requests was low. The most time-consuming task was complying with some projects' style requirements, which we discuss in more detail in Section 6. Other tasks that we had to attend to involved artifacts other than the source code. For instance, projects Eclipse UI, Payara, and mssql-jdbc require every contributor to sign a Contributor Agreement License to handle intellectual property concerns. Project Eclipse UI uses the Gerrit Code Review platform;[10] thus, we had to sign up for an account in Gerrit and configure the Eclipse UI itself to use it. Another complication came from Eclipse UI's versioning process,[11] which requires updates to a file `MANIFEST.MF` in a way that they match modifications in the source code. Project Ant-Media-Server's GitHub repository only accepts pull requests from members of the group of maintainers; therefore, the first author's GitHub account had to be added to the group.

We consider a pull request *approved* when reviewers indicate so in the GitHub interface (or, for project Eclipse UI, when the code changes receive positive review points in Gerrit). Only one pull request was approved but not merged at the time of writing. Since merging depends on other aspects of the development process[12] that are independent of the correctness of a fix, we treat the pull request that was approved but not merged like the others (which were also merged).

As shown in Table 5, the time to review a pull request and reach a decision spans a range that goes from less than 20 minutes (project primefaces) to over 27 days (project mssql-jdbc), not counting the one case (project keanu) where our pull requests were never reviewed. As indicated by related work [44], the time to review pull requests depends on a number of factors, including the extent of the changes, the complexity of the reviewing process, and the amount of resources that each project devotes to reviewing. In most cases, however, the reviewing time was reasonably short, which corroborates the evidence that SpongeBugs's fixes are local and readable.

The reviewing process may approve a patch with or without modifications.[13]

[10] `https://www.gerritcodereview.com/`

[11] `https://www.vogella.com/tutorials/EclipsePlatformDevelopment/article.html#gerrit-verification-failures-due-to-missing-version-update`

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

[13] All of the artifacts and discussions part of the pull request process, which we report in this

For each approved patch generated by SpongeBugs we record whether it was approved with or without modifications.

For most projects, merging occurred within a couple of hours after a pull request was accepted—for four projects within a few minutes. The outliers were projects atomix, ddf, and Payara where merging took as long as (or longer than) deciding whether to approve a pull request. In projects where merging occurred very quickly, it is typically an automated step of the reviewing process. In contrast, heterogeneous reasons can explain a long merging time: in some projects, code changes are merged only in batches at predefined times; in others, contingencies delay the formal approval of a merge.

5. Empirical Evaluation of SpongeBugs: Results and Discussion

The results of our empirical evaluation of SpongeBugs answer the four research questions presented in Section 4.1. For uniformity, the experiments[14] related to RQ1–3 target the 12 projects whose maintainers accepted the pull requests fixing static analysis warnings (top portion of Table 5).

5.1. RQ1: Applicability

To answer **RQ1** ("How widely applicable is SpongeBugs?"), we ran SonarQube on each selected open-source project, counting the warnings triggering 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 been fixed. Table 6 shows the results of these experiments. Overall, SpongeBugs removes 85% of all warnings violating the rules we considered in this research. Another encouraging result is that all fix suggestions generated by SpongeBugs compiled successfully.

These results justify our decision of focusing on a limited number of rules. In particular, the two rules (C3 and C4) with the lowest percentages of fixing are responsible for approximately 3% of the triggered violations. In contrast, a small number of rules trigger the vast majority of violations—on which SpongeBugs is extremely effective.

A widely applicable kind of suggestion are those for violations of rule C1 (*String literals should not be duplicated*), shown in Listing 5, which SpongeBugs can successfully fix in 91% 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 introducing 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

paper, are public according to GitHub's terms of service or Eclipse's contributor agreement.

[14] With the exception of Section 5.1.4, which describes a complementary set of experiments targeting student projects.

Table 6: 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 been fixed 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 UI	41/5	13/10	214/199	4/4	3/2	19/3	189/176	17/11	–	138/94	102/97	740/601	81%
SonarQube	–	–	104/94	–	–	–	7/7	–	–	–	–	111/101	91%
SpotBugs	12/8	1/0	289/247	2/1	1/0	11/1	141/141	–	–	30/26	–	486/424	87%
atomix	1/1	–	57/57	–	–	–	9/9	–	–	1/0	2/1	70/68	97%
Ant Media Server	–	–	28/28	3/2	1/0	2/1	23/23	3/0	–	4/2	4/4	68/60	88%
database-rider	–	–	5/5	5/5	–	–	2/2	–	1/1	1/1	–	14/14	100%
ddf	1/0	–	104/98	–	1/0	–	88/86	–	1/1	45/37	8/8	247/230	93%
DependencyCheck	–	–	61/51	10/9	–	–	3/3	–	–	4/2	–	78/65	83%
keanu	1/1	–	–	–	–	–	4/4	–	12/12	5/5	–	22/22	100%
mssql-jdbc	4/1	–	314/282	14/1	–	7/1	58/58	2/0	–	14/14	–	413/357	86%
Payara	39/36	–	1,413/1,305	214/169	61/14	114/10	1,830/1,627	200/88	50/44	438/301	58/50	4,417/3,644	82%
primefaces	–	–	336/286	11/9	6/6	3/3	336/329	–	1/1	1/0	4/4	698/638	91%
TOTAL	99/52	14/10	2,925/2,652	263/200	71/22	156/19	2,690/2,465	222/99	65/59	681/448	178/164	7,364/6,224	–
FIXED %	53%	71%	91%	76%	31%	12%	92%	45%	91%	71%	92%	85%	–

can also detect whether there is already another string constant with the same value—reusing that instead of introducing a duplicate.

```
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 5: 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 6 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 6: Fix suggestion for a violation of rules C1 and C5 in the same file and statement found in project primefaces.

5.1.1. Detection Accuracy

To better evaluate SpongeBugs's applicability, we investigate when its analysis produces *false positives* and *false negatives*. A false positive is a rule violation that is erroneously reported; in these cases, SpongeBugs produces a suggestion that changes something that need not be changed. A false negative is a rule violation that is *not* reported; in these cases, SpongeBugs should produce some suggestion that is instead missing.

From the point of view of *usability*, false positives are those with the greater potentially negative impact. Indeed, a high false-positive rate is one of the key reasons that limit the adoption of SATs [37]: a user flooded with many false positives quickly concludes that the tool is not reliable because it points to a lot of violations that are incorrect or irrelevant [17]. In contrast, false negatives can be seen as a minor problem, *as long as* a tool is still widely applicable and reports suggestions that help improve the design or other quality attributes of the program.

Since SpongeBugs provides suggestions to enforce rules that are checked by SonarQube, we use SonarQube's detected rule violations as ground truth[15] to count SpongeBugs's false positives and false negatives:

- A suggestion provided by SpongeBugs for which SonarQube reports no rule violation is a *false positive*;
- A rule violation reported by SonarQube for which SpongeBugs provides no suggestions is a *false negative*.

Since it is known that SonarQube's detection algorithm can incur false positives [45] as well as false negatives (a few of which we encountered in our experiments of Section 5.1.2), using it as ground truth may affect SpongeBugs's estimates of the same measures. Precisely, SonarQube's false positives lead to overestimating SpongeBugs's false negatives; and SonarQube's false negatives lead to overestimating SpongeBugs's false positives. Despite these limitations, there is still value in using SonarQube as ground truth: since SpongeBugs is designed to respond to SonarQube's warnings, their output should usually be consistent. On the other hand, we also point out some cases in which SonarQube's misdetections affect what should be considered the "right" false positive and false negative rate of SpongeBugs.

As we discuss in detail in the following subsections, SpongeBugs generates *very few* false positives (0.6% of all violations it detects), and many of these are actually misdetections by SonarQube. In contrast, SpongeBugs generates

[15] Remember that SpongeBugs does not use SonarQube's output to identify rule violations but performs its own detection; otherwise this discussion would be moot.

a more significant number of false negatives (15% of all violations SonarQube detects); but these are not so problematic for applicability since they simply reflect some design decisions that trade-off some detection capabilities in exchange for soundness.

5.1.2. False Positives

Table 7 reports SpongeBugs's false-positive rate in each project and for each rule. Overall, only 0.6% of all fix suggestions provided by SpongeBugs do not correspond to a violation reported by SonarQube. Such a low rate of false positives corroborates the data about SpongeBugs's accuracy and hence practical relevance.

To better understand the few cases in which SpongeBugs generates false positives, we classify all of them into categories according to their origin. We found out that the majority of false positives (23 out of 37 cases) are actually likely *not* false positives but rather false negatives of SonarQube's detection.

Table 7: For each project and each rule checked by SonarQube, the table reports the number of false positives (a fix suggestion provided by SpongeBugs for which SonarQube reported no rule violation). The bottom rows summarize the TOTAL number of false positives, and what percentage of the overall violations reported by SonarQube these false positives correspond to; similarly the rightmost column reports the TOTAL per project.

PROJECT	B1	B2	C1	C2	C3	C4	C5	C6	C7	C8	C9	TOTAL
Eclipse UI	0	0	3	0	0	4	0	0	1	3	1	12
SonarQube	1	0	0	0	0	1	0	0	2	0	0	4
ddf	0	0	6	0	0	0	0	0	0	0	0	6
Payara	0	0	2	0	0	12	0	0	0	1	0	15
TOTAL	1	0	11	0	0	17	0	0	3	4	1	37
FP %	1.9%	0%	0.4%	0%	0%	47.2%	0%	0%	4.8%	0.8%	0.6%	0.6%

True Positives According to SonarLint. SonarLint[16] is a plugin that integrates SonarQube inside the Eclipse IDE. Somewhat surprisingly, SonarLint disagrees with SonarQube's analysis on four rule violations that were fixed by SpongeBugs but not reported by SonarLint. It is possible this disagreement is due to different filtering rules (or warning prioritization rules) used by SonarLint and SonarQube. In any case, these four violations can be considered true positives (even though we classified them as false positives when using SonarQube's output as ground truth).

True Positives According to Developers. Out of all fix suggestions built by SpongeBugs that were accepted as pull requests by developers (see Section 4.3), 17 do not correspond to any rule violation reported by SonarQube. All but one of these fix suggestions correspond to violations of rule C4 (*Parsing should be used to convert strings to primitive types*); 4 of them were accepted by the

[16] Available at: `https://www.sonarsource.com/products/sonarlint/`

maintainers of Eclipse UI and 12 by the maintainers of Payara. This convincingly indicates that most, if not all, of these cases are true positives (and false negatives of SonarQube); at the very least, SpongeBugs's suggestions are considered not harmful by maintainers of the code.

Listing 7 shows an example of SpongeBugs fix suggestion that does not correspond to a rule violation according to SonarQube. In order to understand why SonarQube failed to report this as a violation, we looked for other similar violations of rule C4 that were instead reported by SonarQube as well as fixed by SpongeBugs, such as line 9 in Listing 8. It turns out that if we remove the outer parentheses in subexpression `(Double.valueOf(value))` SonarQube will report a violation of rule C4 in Listing 7 as well. The sensible conclusion is that this is a miss in SonarQube's detection algorithm.

```
public static double asDouble(String value) throws DataFormatException {
    try {
- return (Double.valueOf(value)).doubleValue();
+ return Double.parseDouble(value);
    } catch (NumberFormatException e) {
        throw new DataFormatException(e.getMessage());
    }
}
```

Listing 7: Fix suggestion generated by SpongeBugs for a violation of rule C4 not detected by SonarQube.

```
1  public int getInt(String key) throws NumberFormatException {
2      String setting = items.get(key);
3          if (setting == null) {
4              // Integer.valueOf(null) will throw a NumberFormatException and
5              // meet our spec, but this message is clearer.
6              throw new NumberFormatException(
7                      "There is no setting associated with the key \"" + key +
                           ↪ "\"");//$NON-NLS-1$ //$NON-NLS-2$
8          }
9          return Integer.valueOf(setting).intValue();
10 }
```

Listing 8: Violation of rule C4 detected by both SonarQube and SpongeBugs.

We found several other examples of fragile behavior of SonarQube detecting violations of rule C4, which indicate failures of its analysis algorithm. Like every static analyzer, SonarQube sometimes limits the cases in which a rule is checked, to improve scalability and precision of detection in the other cases. SpongeBugs is no different, but it sometimes achieves different trade-offs than SonarQube—hence the discrepancies we observed in these experiments. By and large, however, SpongeBugs and SonarQube's detection results are consistent and correct.

Actual False Positives. Only 17 out of all fix suggestions produced by SpongeBugs are actual false positives: they correspond to spurious rule violations. In five of these cases we could find a programmer's annotation that explicitly turns off checking of certain rules; unfortunately, SpongeBugs does not process these annotations, and hence it will obliviously flag what it considers a violation.

Listing 9 shows examples of such annotations that should suppress detection. On line 1 a generic annotation suppresses checking all rules in the whole class;

on lines 4–7 an annotation turns of two specific rules within a method; on line 15 a special comment turns off checking a specific rule on the same line where the comment appears.[17]

```
1  @SuppressWarnings("all") // prevents detection of all rules within the entire class
2  public class JavaClass {
3
4      @SuppressWarnings({ // suppress detection of two rules within aMethod()
5          "squid:S1192", // S1192 is SonarQube's identifier for rule C1
6          "squid:S106"
7      })
8      public static void aMethod() {
9          // ...
10     }
11
12     public static void anotherMethod(String s1, String s2) {
13         // the following comment suppresses detection of the rule on the line where NOSONAR
               ↪ appears
14         if (s1 == s2) { // NOSONAR false-positive: Compare Objects With Equals
15     }
16 }
```

Listing 9: Examples of annotations used to suppress detection in SonarQube. SpongeBugs ignores them.

We also found two fix suggestions corresponding to a violation of rule C7 (`entrySet()` *should be iterated when both key and value are needed*) in project SonarQube that are spuriously reported by SpongeBugs. Listing 10 shows the corresponding code, where there is an important difference between an iteration over `keySet()` and one over `entrySet()`. Since a `TreeSet` is created passing `Map prop` as keys, the map is directly used as underlying implementation of the set of keys. An enumeration over `keySet()` will then follow the ordering defined over keys—alphabetical order in this case; in contrast, an enumeration over `entrySet()` may list the elements in a different order (the one in which they are stored in the map). Since method `writeGlobalSettings` produces user output, alphabetical order is expected and should not be changed.

```
1  private void writeGlobalSettings(BufferedWriter fileWriter) throws IOException {
2      fileWriter.append("Global server settings:\n");
3      Map<String, String> props = globalServerSettings.properties();
4      for (String prop : new TreeSet<>(props.keySet())) {
5        dumpPropIfNotSensitive(fileWriter, prop, props.get(prop));
6      }
7  }
```

Listing 10: A *spurious* violation of rule C7 on line 4 reported by SpongeBugs.

The remaining 11 cases of false positives correspond to spurious violations of rule C1 (*String literals should not be duplicated*) where string literals occur inside Java *annotations*. Listing 9 shows an example of spurious fix generated by SpongeBugs. The fix does not alter program behavior in any way; it is just not idiomatic since it mixes program code (a static field) and annotations (which should only be used by the compiler or runtime).

[17] Curiously, SonarQube includes a rule (S1291), which checks that `NOSONAR` annotations are *not* used. This rule is, however, disabled by default.

```
1   public class JCDIServiceIMPL {
2   + private static final String UNCHECKED = "unchecked";
3
4   - @SuppressWarnings("unchecked")
5   + @SuppressWarnings(UNCHECKED)
6       @Override
7       public <T> void injectEJBInstance(JCDIInjectionContext<T> injectionCtx) {
8           JCDIInjectionContextImpl<T> injectionCtxImpl = (JCDIInjectionContextImpl<T>)
                ↪ injectionCtx;
9           // ...
10      }
11  }
```

Listing 11: Fix suggestion generated by SpongeBugs for a spurious violation of rule C1 on line 4. SonarQube does not report this as a violation.

5.1.3. False Negatives

Table 8 reports SpongeBugs's false-negative rate in each project and for each rule. Overall, 15% of rule violations reported by SonarQube were not detected—and hence not fixed—by SpongeBugs. As we mentioned above, this significant false negative rate is not much detrimental to SpongeBugs's practical applicability, since the tool still provides thousands of useful, accurate suggestions for rule violations. As we discuss in Section 3, we designed SpongeBugs to make sure that, when it detects a rule violation, it has all the necessary information to produce a suitable fix suggestion. Therefore, part of the false negative are a consequence of design decisions to trade-off some detection capability for additional precision.

Table 8: For each project and each rule checked by SonarQube, the table reports the number of false negatives (a rule violation reported by SonarQube for which SpongeBugs provides no suggestions). The bottom rows summarize the TOTAL number of false negatives, and what percentage of the overall violations reported by SonarQube these false negatives correspond to; similarly the rightmost column reports the TOTAL per project.

PROJECT	B1	B2	C1	C2	C3	C4	C5	C6	C7	C8	C9	TOTAL
Eclipse UI	36	3	15	0	1	15	13	6	–	44	5	138
SonarQube	–	–	10	–	–	–	0	–	–	–	–	10
SpotBugs	4	1	42	1	0	10	0	–	–	4	–	62
atomix	0	–	0	–	–	–	0	–	–	1	1	2
Ant Media Server	–	–	0	1	1	1	0	3	–	2	0	8
database-rider	–	–	0	0	–	–	0	–	0	0	–	0
ddf	1	–	6	–	0	–	2	–	0	8	0	17
DependencyCheck	–	–	10	1	–	–	0	–	–	2	–	13
keanu	0	–	–	–	–	–	0	–	0	0	–	0
mssql-jdbc	3	–	32	13	–	7	0	2	–	0	–	57
Payara	3	–	108	45	47	104	203	112	6	137	8	773
primefaces	–	–	50	2	0	0	7	–	0	1	0	60
TOTAL #	47	4	273	63	49	137	223	123	6	199	14	1,140
OVERALL %	52%	29%	9%	24%	69%	88%	8%	55%	9%	29%	8%	15%

The percentage of false negatives changes considerably with different rules. To better understand SpongeBugs limitations in practice, the rest of this section

presents several examples of false negatives—with at least one example per kind of limitation (see Section 3.3).

Local Analysis. Listing 12 shows a violation of rule C4 (*Parsing should be used to convert strings to primitive types*) that is detected by SonarQube but is not fixed by SpongeBugs. The latter's analysis of line 5 is oblivious to the fact that method `getStatementTimeout` returns a `String`. Without this information, SpongeBugs cannot detect that rule C4 is being violated by passing a string to `valueOf` instead of `parseInt`.

```
1  public String getStatementTimeout() {
2      return spec.getDetail(DataSourceSpec.STATEMENTTIMEOUT);
3  }
4  // ...
5  int statementTimeout = Integer.valueOf(getStatementTimeout());
```

Listing 12: Violation of rule C4 on line 5, which is not detected by SpongeBugs.

It may seem that providing SpongeBugs with the information that is needed to detect violations such as the one in Listing 12 is straightforward: after all, method `getStatementTimeout` is defined in the same class as where the violation occurs. In our experiments, however, most of the false negatives due to local analysis involve a method that is called in one class but is defined in a different class. Listing 13 shows another violation of rule C4 that SpongeBugs does not detect. In this case, the offending method `getSecurityEnabled` is called in class `IIOPSSLSocketFactory` but is declared in interface `IiopListener`.

```
1  public interface IiopListener {
2      String getSecurityEnabled();
3  }
4
5  public class IIOPSSLSocketFactory {
6      public void aMethod() {
7          for (IiopListener listener : iiopListeners) {
8              boolean securityEnabled = Boolean.valueOf(listener.getSecurityEnabled());
9          }
10     }
11 }
```

Listing 13: Violation of rule C4 on line 8, which is not detected by SpongeBugs.

Extending SpongeBugs's analysis beyond purely local would require to process multiple files at once, and to collect more detailed typing information. While such an extension is beyond SpongeBugs's current design—which privileges simplicity and precision over broader applicability—we may consider it in future work. In order to do it efficiently, we may pre-process the whole codebase at once [46]; then, each individual analysis could access this system-wide information as needed in an efficient way. To be truly system-wide, this approach would also need to track dependencies outside a project's source code—such as in calls to pre-compiled or even native libraries.

Contextual Restrictions. SpongeBugs checks rule C3 (*Strings should not be concatenated using `+` in a loop*) only when the concatenated string is eventually returned by a method. Thus, SpongeBugs misses the violation of rule C3 on

line 7 in Listing 14 because string `containerStyleClass`, which is built by concatenation in a loop, is not returned by method `encodeElements`.

```
1  protected void encodeElements(Menu menu, List<MenuElement> elements) {
2      boolean toggleable = menu.isToggleable();
3      for (MenuElement element : elements) {
4          String containerStyleClass = menuItem.getContainerStyleClass();
5
6          if (toggeable) {
7              containerStyleClass = containerStyleClass + " " + Menu.SUBMENU_CHILD_CLASS;
8          }
9      }
10 }
```

Listing 14: Violation of rule C3 on line 7, which is not detected by SpongeBugs.

Extending SpongeBugs so that it can handle cases like this would trigger a significant number of false positives unless we also sharpened its analysis with much more contextual information. The current restriction on rule C3 is a good trade-off because it curbs the number of rule violations that are detected while still covering the most salient cases.

Other contextual restrictions mainly simplify the analysis, helping ensure that a rule violation's can be detected correctly. For example, SpongeBugs only checks rule C8 (`Collection.isEmpty()` *should be used to test for emptiness*) inside regular methods. Thus, it misses the violation in Listing 15 because it appears inside a lambda expression, which makes the analysis of local variables considerably more difficult.

```
1  public class LeaderElectorProxy {
2      private final Map<String, Set<LeadershipEventListener<byte[]>>> topicListeners =
           ↪ Maps.newConcurrentMap();
3
4      public synchronized CompletableFuture<Void> removeListener(String topic, Listener<btye[]
           ↪ listener>) {
5          if (!topicListeners.isEmpty()) {
6              topicListeners.computeIfPresent(topic, (t, s) -> {
7                  s.remove(listener);
8                  return s.size() == 0 ? null : s;
9              });
10         }
11         // ...
12     }
13 }
```

Listing 15: Violation of rule C8 on line 8, which is not detected by SpongeBugs.

Typing Information. Another restriction in the application of rule C8 follows from SpongeBugs's limited information about how types are related by inheritance. Listing 16 shows a violation of this rule on line 5: `CopyOnWriteArrayList` implements the `List` interface, and hence the conditional on line 5 should be expressed as `!Collections.isEmpty()`. However, SpongeBugs only knows about the most common implementations of collection classes, and hence it misses this violation. This restriction also affects the other rules that involve `Collection` classes, that is rules C7 and C9.

```
1  import java.util.concurrent.CopyOnWriteArrayList;
2
3  protected CopyOnWriteArrayList<IPlayItem> items;
```

```
4   // ...
5   if (items.size > 0)
```

Listing 16: Violation of rule C8 on line 5, which is not detected by SpongeBugs.

Toolchain. Because Rascal's Java 8 grammar is incomplete, SpongeBugs could not parse 16 classes that SonarQube could analyze. These include unsupported syntax such as adding a semicolon after the closing bracket of a class declaration, or casting a lambda expression. None of these are widely used features, and in fact they appeared only in a tiny fraction of all analyzed code.

5.1.4. Student Projects

To further confirm the applicability and usefulness of SpongeBugs also on code written by non-professionals, we ran additional experiments in which we applied it to the five student projects presented in Section 4.2.2. While we have no a priori reason to expect that SpongeBugs would perform poorly on code written by students, extending the pool of analyzed programs beyond open-source projects helps strengthen the generalizability of the results of our empirical evaluation.

First of all, we ran SonarQube on these projects checking the usual 11 rules supported by SpongeBugs. Table 9 shows the results in terms of number of violations reported by SonarQube for every thousand lines of code. The table also shows the same measure for the open-source projects used in the rest of the evaluation. Overall, rule violations occur with higher frequencies in the student projects than in the open-source projects—as it can be expected from code written by non-professional. Nonetheless, the student code generally is of high quality since its violations are not *much* higher. Remember that all projects but Project 5 explicitly required students to check their code with SonarQube and to modify it to reduce the number of reported violations. The only project in which students were not required to use SonarQube is also the one with the largest number of violations per line of code, but the difference with projects of the same course is not big. Since all students knew SonarQube as a tool from previous courses, it is possible that they used it regardless of whether it was required by the project's specification, and that they generally paid attention to writing code that conforms with accepted coding guidelines.

Table 10 summarizes the results of applying SpongeBugs to the rule violations in student projects. For all violations but one (98% of all violations), SpongeBugs produced a correct fix suggestion that avoided the violation. At a high level, these results are comparable to those obtained on the open-source projects.

The results on student projects are consistent with those on open-source projects also in terms of which *rules* are most frequently violated and fixed. The most frequent violations are of rules C1, C5, and C8—in this order in both the student projects and in the open-source projects. On the other hand, student projects violated none of rules B1, B2, C4, and C9; in the open-source projects there were several violations of these rules, but they accounted for only about the 6% of all violations, and did not occur in all projects. Students ran

Table 9: Top half of table: for each student project, its size LOC in non-blank non-comment lines of code and the number of violations of the 11 rules checked by SpongeBugs that are detected by SonarQube per thousands of lines of code (VIOLATIONS/KLOC). Bottom half of table: the same data about the open-source projects used in the rest of the evaluation.

PROJECT	LOC	VIOLATIONS / KLOC
Project 1	6.8 K	0.00
Project 2	5.2 K	4.23
Project 3	2.5 K	0.80
Project 4	2.4 K	4.60
Project 5	3.6 K	5.80
Eclipse UI	743 K	1.03
SonarQube	500 K	0.22
SpotBugs	280 K	1.80
atomix	550 K	0.13
Ant Media Server	43 K	1.12
database-rider	21 K	0.67
ddf	2.5 M	0.01
DependencyCheck	182 K	0.43
keanu	145 K	0.15
mssql-jdbc	79 K	5.23
Payara	1.95 M	2.26
primefaces	310 K	2.23

SonarQube with a custom profile, which excluded some of the rules SpongeBugs checks; as a result even projects that were required to use SonarQube still incur some rule violations.

The lone violation that SpongeBugs could not fix is a false negative similar to one discussed in Section 5.1.3 and Listing 14 for the same rule: SpongeBugs's detection of rule C3 only analyzes variables that are returned by a method.

Overall, these experiments confirm that SpongeBugs is usefully applicable on a variety of projects, and can help programmers automatically address a significant fraction of SonarQube warnings.

5.2. RQ2: Effectiveness and Acceptability

In order to answer **RQ2** ("Does SpongeBugs generate fixes that are acceptable?"), we assess the *acceptability* of SpongeBugs's suggestions in two ways. First, we submitted pull requests including a large selection of fixes for the projects used in SpongeBugs's evaluation. Second, we ran the official test suites of each project on its implementation modified by including all of SpongeBugs's suggestions. The two following sections report the results of these experiments.

Table 10: For each student project and each rule, the table reports two numbers x/y: x is the number of warnings violating 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. For brevity, the table only reports the rules for which SonarQube found at least one violation in some project.

PROJECT	C1	C3	C5	C7	C8	TOTAL	FIXED %
Project 1	–	–	–	–	–	–	–
Project 2	18/18	–	–	–	4/4	22/22	100%
Project 3	–	–	2/2	–	–	2/2	100%
Project 4	2/2	–	8/8	1/1	–	11/11	100%
Project 5	8/8	1/0	3/3	2/2	7/7	21/20	95%
TOTAL	28/28	1/0	13/13	3/3	11/11	56/55	–
FIXED %	100%	0%	100%	100%	100%	98%	–

5.2.1. Acceptability Through Pull Requests

As discussed in Section 4.3, 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 5, 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 one negative answer from a project, and no timely reply from two other projects, and hence we did not submit any pull request to them (and we excluded them from the rest of the evaluation).

Accordingly, we submitted 38 pull requests containing 946 fixes for the 12 projects that answered positively or neutrally to our inquiries. 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 5 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, 825

(87%) of all 946 submitted fixes were accepted; 797 (97%) 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 (100% precision, as all fixes compile correctly), one that was accepted in a pull request (precision: 87%), or one what was accepted without modifications (precision: 797/946 = 84%). 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 17 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 accepted but not merged, 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 17: 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 18 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 18: 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 19, 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 19: 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 UI) 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 20 received the following feedback from maintainers of Eclipse UI:

> "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 20: 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 21 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 21: 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.

5.2.2. Acceptability Through Testing

SpongeBugs's suggestions should be *refactorings*: they should change the code's structure and style without otherwise changing its (input/output) behavior. One of the main reasons software developers equip their projects with regression test suites is precisely to easily detect unintended behavioral changes [26, 25]. Therefore, we used the projects' test suites to help determine if SpongeBugs's suggestions may have introduced deleterious changes in program behavior.

We were able to run the test suites of 8 out of the 12 open-source projects used in SpongeBugs's evaluation. Running the tests of the other 4 projects was not possible because we could not reproduce their testing setup. Namely: 1. Running tests in projects database-rider and mssql-jdbc requires a database instance or a stub, neither of which is available in the projects' repositories. 2. We could not configure projects Eclipse UI and Ant Media Server in a way suitable to run their tests. These two projects' build documentation[18] strictly recommends building *without* running tests, which probably indicates their testing configuration requires a certain infrastructure or information that is not directly available in the projects' repositories.

Table 11 shows the results of running the test suites of these 8 projects. By and large, they confirm that SpongeBugs suggestions are unlikely to alter program behavior. Only a single test case that was passing on the original code failed after introducing SpongeBugs's suggestions; all other tests had the same behavior—passing or failing—on the original code as on the code modified by SpongeBugs. The lone failing test displays behavior similar to that of Listing 10, and hence SpongeBugs's suggestion is based on a false positive of its detection.

[18] See this paper's replication package for details.

Table 11: Summary of running test suites in all projects. Column KLOC reports the size (in thousands of lines of code) of the test suites; column RAN TESTS? indicates for which projects we could run test suites; column #FAILING indicates the number of tests that failed on the projects modified with all of SpongeBugs's suggestions.

PROJECT	KLOC	RAN TESTS?	#FAILING
Eclipse UI	159	No[a]	–
SonarQube	278	Yes	1
SpotBugs	3	Yes	0
Ant Media Server	11	No[a]	–
atomix	17	Yes	0
database-rider	8	No[b]	–
ddf	164	Yes	0
DependencyCheck	9	Yes	0
keanu	25	Yes	0
mssql-jdbc	31	No[b]	–
Payara	59	Yes	0
primefaces	4	Yes	0

[a] Recommends building without executing tests.
[b] Requires a database.

5.3. RQ3: Performance

To answer **RQ3** ("How efficient is SpongeBugs?"), we report various 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; Rascal was installed as a plugin in Eclipse, therefore we ran SpongeBugs in Eclipse. We used Rascal's native benchmark library[19] to measure how long our transformations take to run on the projects considered in Table 6. Table 12 show the performance outcomes. For each of the measurements in this section, we follow recommendations on measuring performance [47]: we restart the laptop after each measurement, to avoid any startup performance bias (i.e., classes already loaded); and summarize, through descriptive statistics, five repeated runs of SpongeBugs on each project.

5.3.1. Overall Performance

Table 12 shows SpongeBugs's running time on the 12 open-source projects used in the evaluation. The running time is roughly proportional to the number of files analyzed in each project, and scales even to the largest, most complex projects.

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

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

Table 12: Descriptive statistics summarizing 5 repeated runs of SpongeBugs. Time is measured in minutes.

		RUNNING TIME [min.]	
PROJECT	FILES ANALYZED	MEAN	ST. DEV.
Eclipse UI	5,282	102.3	2.31
SonarQube	3,876	25.3	0.84
SpotBugs	2,564	30.6	1.20
Ant Media Server	228	3.9	0.14
atomix	1,228	10.1	0.74
database-rider	109	0.8	0.04
ddf	2,316	28.6	1.55
DependencyCheck	245	5.5	0.21
keanu	445	2.9	0.09
mssql-jdbc	158	23.2	0.62
Payara	8,156	166.1	8.88
primefaces	1,080	15.5	1.45

files with more than one thousand 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 13, five larger files are responsible for more than 12 minutes (52%) of running time. Additionally, file `dtv` takes, on average longer to run than `SQLServerConnection`, even though it is 36% smaller; the reason is that file `dtv` has numerous class declarations and methods with more than 300 lines, containing multiple `switch`, `if`, and `try`/`catch` statements.

Table 13: Descriptive statistics summarizing five repeated runs of SpongeBugs on the five largest files in projects mssql-jdbc. Time is measured in seconds; size in non-blank non-comment lines of code LOC.

		RUNNING TIME [s]	
FILE	LOC	MEAN	ST. DEV.
SQLServerConnection	4,428	202	14.1
SQLServerResultSet	3,858	158	22.9
dtv	2,823	208	17.8
SQLServerBulkCopy	2,529	86	5.4
SQLServerPreparedStatement	2,285	78	6.9

Generating some fix suggestions takes longer than others. We investigated this aspect more closely on the SpotBugs project, as it includes more than a thousand files, and contains multiple test cases for the rules it implements. While SpongeBugs normally excludes files in `src/test/java`, this does not work

for SpotBugs, which puts tests in another location; therefore, SpongeBugs ends up analyzing a much larger amount of code.

SpongeBugs takes considerably longer to run on rules B1, B2, C1, and C6. 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 3.2). For example, step 1's filtering for rule B1 (*Strings and boxed types should be compared using* `equals()`), shown in Listing 22, 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. Contrast this to step 1's filtering for rule C9 (`Collections.EMPTY_LIST`... *should not be used*), shown in Listing 23, which is much more restrictive; as a result SpongeBugs runs in under 20 seconds for rule C9.

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

Listing 22: Violation textual pattern in the implementation of rule B1

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

Listing 23: Violation textual pattern in the implementation of rule C9

Overall, we found that SpongeBugs's approach to fix warnings of SATs scales 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.

5.3.2. Precision vs. Performance Trade-Off

SpongeBugs's detection process (described in Section 3.2) starts with a first step that is fast but potentially imprecise: since it is based on textual matching, it is sensitive to differences in layout such as extra whitespaces. How much imprecision does step 1 introduce, and what performance gains does it bring in return?

We ran our evaluation again using SpongeBugs with step 1 disabled; this means that it directly performs the precise yet slower step 2, which is based on AST matching. Table 14 and Table 15 show the results in comparison with the rest of the experiments (where step 1 is enabled).

With step 1 disabled, SpongeBugs detected (and fixed) an additional 18 rule violations; this is a tiny fraction of all rule violations detected by SpongeBugs (with or without step 1). The new violations are all in project Payara, which is responsible for almost 60% of all 7,364 warnings detected by SonarQube. Listing 24 shows a violation of rule C5 (*Strings literals should be placed on the left side when checking for equality*) that is only detected without step 1. The blank space between the open parenthesis and the string argument fails the strict textual pattern of step 1 for this rule.

```
if ( lookupName.equals( "java:comp/BeanManager" ) ) {
```

Listing 24: A violation of rule C5 that step 1 misses.

Table 14: For each project and each rule checked by SonarQube, the table reports the number of *additional* fixes generated by SpongeBugs *without* executing step 1 (see Section 3.2); that is the fixes corresponding to rule violations that are missed by step 1. The bottom rows summarize the TOTAL number of such additional fixes, and what percentage of the overall violations reported by SonarQube these additional fixes correspond to; similarly the rightmost column reports the TOTAL per project.

PROJECT	B1	B2	C1	C2	C3	C4	C5	C6	C7	C8	C9	TOTAL
Eclipse UI	0	0	0	0	0	0	0	0	–	0	0	0
SonarQube	–	–	0	–	–	–	0	–	–	–	–	0
SpotBugs	0	0	0	0	0	0	0	–	–	0	–	0
atomix	0	–	0	–	–	–	0	–	–	0	0	0
Ant Media Server	–	–	0	0	0	0	0	0	–	0	0	0
database-rider	–	–	0	0	–	–	0	–	0	0	–	0
ddf	0	–	0	–	0	–	0	–	0	0	0	0
DependencyCheck	–	–	0	0	–	–	0	–	–	0	–	0
keanu	0	–	–	–	–	–	0	–	0	0	–	0
mssql-jdbc	0	–	0	0	–	0	0	0	–	0	–	0
Payara	0	–	**10**	**1**	0	0	**7**	0	0	0	0	**18**
primefaces	–	–	0	0	0	0	0	–	0	0	0	0
TOTAL #	0	0	10	1	0	0	7	0	0	0	0	18
OVERALL %	–	–	0.34%	0.38%	–	–	0.26%	–	–	–	–	0.24%

To achieve such a modest increase in detection precision, the running time more than doubled—nearly tripling for projects SonarQube and keanu. In con-

Table 15: Summary of 5 repeated runs of SpongeBugs WITH and WITHOUT violation textual pattern filtering (step 1), and the corresponding SLOW DOWN ratio. Time is measured in minutes.

	RUNNING TIME [min.]		
PROJECT	WITH STEP 1	WITHOUT STEP 1	SLOW DOWN
Eclipse UI	102.3	235.6	2.3
SonarQube	25.3	74.6	2.9
SpotBugs	30.6	86.8	2.8
Ant Media Server	3.9	8.4	2.1
atomix	10.1	28.9	2.9
database-rider	0.8	2.0	2.5
ddf	28.6	67.2	2.3
DependencyCheck	5.5	11.3	2.1
keanu	2.9	8.5	2.9
mssql-jdbc	23.2	33.2	1.4
Payara	166.1	342.1	2.1
primefaces	15.5	31.4	2.0
OVERALL	414.6	930.0	2.2

trast, project mssql-jdbc had the lowest increase in running time, which is probably due to abundance of large classes: as we discussed in Table 13, step 2 was running on most of these classes anyway.

In conclusion, it seems that SpongeBugs's two-step detection process achieves a good balance between precision and performance: disabling step 1 increases precision only marginally, but more than doubles the running time.

5.4. RQ4: Code with Behavioral Bugs

To answer RQ4, we ran SpongeBugs on all 438 bugs from Defects4J. Each experiment targets one bug and consists of two steps:

1. Run SpongeBugs on the *buggy* code, and apply all suggested fixes.
2. Run the *tests* associated with the code (which include at least one failing test) on the version with all SpongeBugs suggestions, and record which tests are passing or failing.

By running the test suite that comes with every bug, we can assess whether SpongeBugs's suggestions interfere with the intended program behavior; and, if they do, whether they improve or worsen correctness as captured by the tests.

Overall, SpongeBugs suggested 675 fix suggestions across all bugs in Defects4J; as usual, all suggestions compile without errors. In the overwhelming majority of cases, SpongeBugs's suggestions did *not* alter program behavior: for 22,253 out of 22,270 tests in Defects4J, tests that were previously passing were still passing, and tests that were previously failing were still failing. SpongeBugs did not fix any of the semantic bugs in Defects4J, which is what we expected since its rules do not target behavioral correctness.

The exceptions involved 17 tests in Defects4J that were originally passing (in the buggy version of each program) but turned into failing tests after we applied SpongeBugs's suggestions. In these cases, SpongeBugs altered program behavior in a way that is inconsistent with the intended one captured by the originally passing tests.

All these 17 cases involved spurious violations of rule B1 *Strings and boxed types should be compared using* `equals()`. We observed one similar case of spurious violation in SpongeBugs's detection of rule B1 in the experiments of Section 5.1.2, but the phenomenon is more prominent in Defects4J. Let's outline these cases of spurious detection to better understand where SpongeBugs fails. In all cases, SonarQube reports the same spurious warnings.

Out of the 17 failing tests, 14 are from project Closure[20]—a JavaScript optimizing compiler written in Java. SpongeBugs found 5 violations of rule B1 in the buggy project version included in Defects4J; all of these violations, reported by both SpongeBugs and SonarQube, are spurious. The root cause of these is the nature of project Closure, which relies on sophisticated optimizations involving string manipulation. Reference equality `==` is used instead of object

[20] `https://github.com/google/closure-compiler`

equality `equals()` as much as possible—when it is semantically correct—because it is faster. SpongeBugs's fix suggestions replace expressions like `s == t` with `s.equals(t)`; however, the latter implies the former *only if* `s` is not `null`. Thus, some of SpongeBugs's fix suggestions introduce a crash in test that exercise such code with null strings. The following example–which is the single responsible for the 14 failing tests—in project Closure is hard to miss, since the programmer explicitly documented their intention to use reference equality:

```
//yes, s1 != s2, not !s1.equals(s2)
 if (lastSourceFile != sourceFile)
```

The other 3 tests that became failing after applying SpongeBugs's suggestions are from project Lang—Apache's popular Commons Lang base library for Java. SpongeBugs found 3 violations of rule B1 in the buggy project version included in Defects4J; all of these violations, reported by both SpongeBugs and SonarQube, are spurious. The root cause of these is again the way in which `null` strings are handled. Project Lang's API for strings uses defensive programming, and hence it generally supports `null` values instead of valid `String` objects. Take, for example, method `indexOfDifference(String s1, String s2)` of class `StringUtils`[21], which returns the lowest index at which `s1` differs from `s2`. The method's JavaDoc documentation explicitly says that `s1`, `s2`, or both may be null; correspondingly, reference equality is generally used *before* object equality, so as to be able to reliably compare strings that are `null`. When SpongeBugs introduces changes like:

```
if (str1 == str)   ⟶   if (str1.equals(str2))
```

the program will throw a `NullPointerException` whenever str1 is `null`.

Interestingly, in version 3.0 of the library, the developers of Apache Commons modified method `indexOfDifference` so that it inputs two `CharacterSequence` objects instead of strings. Comparing `CharacterSequence` objects by reference is not considered an anti-pattern, and hence neither SonarQube nor SpongeBugs would flag this more recent version of the library. While we could not verify the actual intentions of the developers, it is possible that they did consider string comparisons using `==` something to be avoided whenever possible—thus partially vindicating SonarQube's and SpongeBugs's strict application of rule B1.

Another similar spurious violation of rule B1 occurs in Apache Commons class `BooleanUtils`: Method `toBoolean(String str)` accepts `null` as argument `str`, but SpongeBugs's fix suggestion would crash with a `NullPointerException` in this case:

```
if (str1 == "true")   ⟶   if (str1.equals("true"))
```

Interestingly, if we combine SpongeBugs's suggestion for rule B1 with its suggestion for rule C5 (*Strings literals should be placed on the left-hand side when*

[21] https://commons.apache.org/proper/commons-lang/javadocs/api-2.0/org/apache/commons/lang/StringUtils.html#indexOfDifference(java.lang.String,%20java.lang.String)

checking for equality) the code reverts to handling the case `str == null` correctly:

```
if (str1.equals("true"))   ⟶   if ("true".equals(str1))
```

This suggests that static analysis rules are sometimes not independent—and hence stylistic guidelines should be followed consistently.

The main conclusions we can draw from the experiments with Defects4J are as follows:

- As expected by its design, SpongeBugs cannot fix semantic (behavioral) bugs because it targets syntactic (stylistic) rules.
- By and large, SpongeBugs's fix suggestions are unlikely to alter program behavior in unintended ways.
- For programs following unusual conventions or particular implementation styles, the rules checked by SonarQube and SpongeBugs may sometimes misfire. Often, it is still possible to refactor the program so that it follows the intended behavior while also adhering to conventional stylistic rules.

5.5. 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 25: change string to character).

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

Listing 25: 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 represents accepted technical debt. 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 SAT rules, and the corresponding fixes are generally easy to review, submitting patches to fix violations is an approachable way to start contributing to open source projects.

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.

6. Limitations and Threats to Validity

Some of SpongeBugs's transformations may violate a project's stylistic guidelines [17]. Take, for example, project primefaces, which uses a rule[22] about the order of variable declarations within a class requiring 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.[23] Overall, these limitations appear minor, since stylistic rules can often be enforced automatically using a pretty printer (the style checker of most projects include one).

SATs 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 [25]. 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.

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

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

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 C3: 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 on any kinds of projects, we leave a more extensive experimental evaluation to future work.

7. 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 946) of the fixes it generated as pull requests in the projects. Overall, project maintainers accepted 825 (87%) of those fixes—most of them (97%) without any modifications. A manual analysis also confirmed that SpongeBugs is very accurate, as only a tiny fraction (0.6%) of all its fix suggestions can be classified as false positives. 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); and its applicability to student code and to the Defects4J curated collection of bugs. Overall, the 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. Since SpongeBugs does not provide commit messages (which are typically needed in a pull request, and which we wrote manually in SpongeBugs's evaluation), we also plan to combine techniques we developed in recent related work [48] to automatically generate meaningful pull request descriptions.

Acknowledgments. We thank the maintainers for reviewing our patches; and the reviewers of SCAM and JSS for their helpful comments. This work was partially supported by CNPq (#406308/2016-0 and #309032/2019-9); and by the Swiss National Science Foundation (SNSF) grant *Hi-Fi* (#200021_182060).

References

[1] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, 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, ICPC '19, IEEE Press, Piscataway, NJ, USA, 2019, pp. 209–219. doi:10.1109/ICPC.2019.00040.
URL https://doi.org/10.1109/ICPC.2019.00040

[2] A. Habib, 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, ASE 2018, ACM, New York, NY, USA, 2018, pp. 317–328. doi:10.1145/3238147.3238213.
URL http://doi.acm.org/10.1145/3238147.3238213

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

[4] K. Liu, A. Koyuncu, D. Kim, 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), 2019, pp. 1–12. doi:10.1109/SANER.2019.8667970.

[5] T. Barik, Y. Song, B. Johnson, 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), 2016, pp. 211–221. doi:10.1109/ICSME.2016.63.

[6] K. F. Tómasdóttir, M. Aniche, A. van Deursen, Why and how JavaScript developers use linters, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 578–589. doi:10.1109/ASE.2017.8115668.

[7] M. Beller, R. Bholanath, S. McIntosh, 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, 2016, pp. 470–481. doi:10.1109/SANER.2016.105.

[8] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix, Using static analysis to find bugs, IEEE Software 25 (5) (2008) 22–29. doi:10.1109/MS.2008.130.

[9] D. Marcilio, C. A. Furia, R. Bonifácio, G. Pinto, Automatically generating fix suggestions in response to static code analysis warnings, in: 19th IEEE

International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, USA, September 30–October 1, 2019, IEEE, 2019, pp. 34–44. doi:10.1109/SCAM.2019.00013.
URL https://doi.org/10.1109/SCAM.2019.00013

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

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

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

[13] S. Cherem, L. Princehouse, 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, PLDI '07, ACM, New York, NY, USA, 2007, pp. 480–491. doi:10.1145/1250734.1250789.
URL http://doi.acm.org/10.1145/1250734.1250789

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

[15] T. Rausch, W. Hummer, P. Leitner, 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, 2017. doi:10.1109/msr.2017.54.
URL https://doi.org/10.1109/msr.2017.54

[16] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, 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, 2017. doi:10.1109/msr.2017.2.
URL https://doi.org/10.1109/msr.2017.2

[17] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, Y. Le Traon, Mining fix patterns for FindBugs violations, IEEE Transactions on Software Engineering (2018) 1–1doi:10.1109/TSE.2018.2884955.

[18] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, 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), 2018, pp. 153–163. doi:10.1109/SANER.2018.8330205.

[19] E. Aftandilian, R. Sauciuc, S. Priya, 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, 2012, pp. 14–23. doi:10.1109/SCAM.2012.28.

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

[21] C. Liu, J. Yang, L. Tan, M. Hafiz, R2fix: Automatically generating bug fixes from bug reports, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 282–291. doi:10.1109/ICST.2013.24.

[22] R. Rolim, G. Soares, R. Gheyi, L. D'Antoni, Learning quick fixes from code repositories, CoRR abs/1803.03806 (2018). arXiv:1803.03806.
URL http://arxiv.org/abs/1803.03806

[23] Y. Padioleau, J. Lawall, R. R. Hansen, G. Muller, Documenting and automating collateral evolutions in linux device drivers, SIGOPS Oper. Syst. Rev. 42 (4) (2008) 247–260. doi:10.1145/1357010.1352618.
URL https://doi.org/10.1145/1357010.1352618

[24] R. Just, D. Jalali, 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, ISSTA 2014, ACM, New York, NY, USA, 2014, pp. 437–440. doi:10.1145/2610384.2628055.
URL http://doi.acm.org/10.1145/2610384.2628055

[25] M. Monperrus, Automatic software repair: A bibliography, ACM Comput. Surv. 51 (1) (2018) 17:1–17:24. doi:10.1145/3105906.
URL http://doi.acm.org/10.1145/3105906

[26] L. Gazzola, D. Micucci, L. Mariani, Automatic software repair: A survey, IEEE Trans. Software Eng. 45 (1) (2019) 34–67. doi:10.1109/TSE.2017.2755013.
URL https://doi.org/10.1109/TSE.2017.2755013

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

[28] D. Kim, J. Nam, J. Song, S. Kim, Automatic patch generation learned from human-written patches, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, 2013, p. 802–811.

[29] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, A. Zeller, Automated fixing of programs with contracts, in: Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA), ACM, 2010, pp. 61–72.

[30] Z. Qi, F. Long, S. Achour, M. Rinard, An analysis of patch plausibility and correctness for generate-and-validate patch generation systems, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA), ACM, 2015, pp. 24–36.

[31] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, A. Scott, Sapfix: Automated end-to-end repair at scale, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 269–278. doi:10.1109/ICSE-SEIP.2019.00039.

[32] M. Martinez, M. Monperrus, Coming: A tool for mining change pattern instances from git commits, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2019, pp. 79–82. doi:10.1109/ICSE-Companion.2019.00043.

[33] J. Bader, A. Scott, M. Pradel, S. Chandra, Getafix: Learning to fix bugs automatically, Proc. ACM Program. Lang. 3 (OOPSLA) (Oct. 2019). doi:10.1145/3360585.
URL https://doi.org/10.1145/3360585

[34] F. Long, M. Rinard, Automatic patch generation by learning correct code, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 298–312. doi:10.1145/2837614.2837617.
URL https://doi.org/10.1145/2837614.2837617

[35] F. Long, P. Amidon, M. Rinard, Automatic inference of code transforms for patch generation, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, 2017, p. 727–739. doi:10.1145/3106237.3106253.
URL https://doi.org/10.1145/3106237.3106253

[36] R. Bavishi, H. Yoshida, M. R. Prasad, Phoenix: Automated data-driven synthesis of repairs for static analysis violations, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, ACM, New York, NY, USA, 2019, pp. 613–624. doi:10.1145/3338906.3338952.
URL http://doi.acm.org/10.1145/3338906.3338952

[37] B. Johnson, Y. Song, E. R. Murphy-Hill, 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.

[38] A. Brito, L. Xavier, A. Hora, 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.

[39] P. Klint, T. Van Der Storm, 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.

[40] R. Dantas, A. Carvalho, D. Marcílio, L. Fantin, U. Silva, W. Lucas, 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), 2018, pp. 497–501. doi:10.1109/SANER.2018.8330247.

[41] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, An in-depth study of the promises and perils of mining GitHub, Empirical Software Engineering 21 (5) (2015) 2035–2071. doi:10.1007/s10664-015-9393-5.
URL https://doi.org/10.1007/s10664-015-9393-5

[42] Y. Tao, D. Han, S. Kim, Writing acceptable patches: An empirical study of open source project patches, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 271–280. doi:10.1109/ICSME.2014.49.

[43] A. Ram, A. A. Sawant, M. Castelluccio, 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. doi:10.1145/3236024.3236080.
URL https://doi.org/10.1145/3236024.3236080

[44] Y. Yu, H. Wang, G. Yin, T. Wang, Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?, Inf. Softw. Technol. 74 (2016) 204–218.

[45] V. Lenarduzzi, F. Lomio, H. Huttunen, D. Taibi, Are SonarQube rules inducing bugs?, in: IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, https://arxiv.org/abs/1907.00376.

[46] J. Kim, D. Batory, D. Dig, M. Azanza, Improving refactoring speed by 10x, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 1145–1156. doi:10.1145/2884781.2884802.

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

[48] A. Carvalho, W. Luz, D. Marcílio, R. Bonifacio, G. Pinto, E. D. Canedo, C-3PR: A bot for fixing static analysis violations via pull requests, in: IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020.