# Static Analysis Warnings and Automatic Fixing: A Replication for C# Projects

Martin Odermatt
Software Institute
USI Università della Svizzera italiana
Lugano, Switzerland
martin.odermatt@usi.ch

Diego Marcilio
Software Institute
USI Università della Svizzera italiana
Lugano, Switzerland
dvmarcilio.github.io

Carlo A. Furia
Software Institute
USI Università della Svizzera italiana
Lugano, Switzerland
bugcounting.net

***Abstract*—Static analyzers have become increasingly popular both as developer tools and as subjects of empirical studies. Whereas static analysis tools exist for disparate programming languages, the bulk of the empirical research has focused on the popular Java programming language.**

**In this paper, we investigate to what extent some known results about using static analyzers for Java change when considering C#—another popular object-oriented language. To this end, we combine two replications of previous Java studies. First, we study which static analysis tools are most widely used among C# developers, and which warnings are more commonly reported by these tools on open-source C# projects. Second, we develop and empirically evaluate EagleRepair: a technique to automatically fix code in response to static analysis warnings; this is a replication of our previous work for Java [20].**

**Our replication indicates, among other things, that 1) static code analysis is fairly popular among C# developers too; 2) ReSharper is the most widely used static analyzer for C#; 3) several static analysis rules are commonly violated in both Java and C# projects; 4) automatically generating fixes to static code analysis warnings with good precision is feasible in C#. The EagleRepair tool developed for this research is available as open source.**

## I. Introduction

Static analysis tools (SATs) are nowadays regularly used by developers to help them detect defects and to enforce a consistent programming style throughout the development lifecycle. Reflecting this increasing uptake by practitioners, SATs have also become a popular target for software engineering research (see Sec. VIII)—both to understand how developers use them and to improve their overall usability. Most of the empirical work on SATs targets the Java programming language. This paper is a replication of some of these studies that targets instead the C# programming language.

C# is a popular[a] object-oriented programming language, which shares several similarities with Java but also some interesting differences. Its user community, in particular, is somewhat narrower than Java, as it revolves around the .NET framework—which was initially focused on the Windows operating system, but has gradually become available in other systems as well. To our knowledge, no previous studies looked into how SATs are used by C# developers nor how to automatically fix some of their common warnings. This paper fills this knowledge gap, and contributes a tool that can automatically fix SAT warnings flagged in C# projects.

More precisely, our replication is in two parts, each targeting different previous work about Java. First, we consider studies of how SATs are used in Java projects [9], [14], [18]: what SATs are more popular among practitioners, and what warnings they commonly report. Second, we consider our previous work [19] introducing SpongeBugs: a technique to automatically fix warnings reported by SonarQube (a popular SAT), and analyzing its effectiveness on real-world Java projects.

In the present paper, we perform a partial replication of the first line of work: we present an empirical study of SonarQube and ReSharper (two of the SATs used by C# developers) where we ran them on 100 open-source C# projects to determine which defects they most commonly detect in these projects. Following up on these results, we perform a full replication of the second line of work: we present the EagleRepair technique, which automatically builds suggestions for source-code fixes of some static-analysis rule violations that commonly occur in C# projects. We implemented the EagleRepair technique as a tool, and validated its effectiveness by running it on the same 100 projects identified in the first part of this study.

The main results of our replication are as follows. Regarding the first part, we confirm that SATs are used in C# projects as well; comparing the most frequent warnings, we found several rules that are frequently violated both in Java and in C#. Regarding the second part, EagleRepair demonstrates that automatically fixing SAT warnings is possible in C# too. In our experiments, EagleRepair achieved precision and recall (78% and 58%) that are lower than those of SpongeBugs (the Java technique we replicated), but still good enough to be practically useful. Like with SpongeBugs, we also confirmed the acceptability of EagleRepair's fixes by submitting 281 of them as pull requests—89% of which were accepted by the project developers.

In summary, the main contributions of this paper are:

- A partial replication—on 100 C# projects—analyzing which SATs are more widely used, and which warnings are more frequently raised.
- A full replication in C# of a technique to automatically fix SAT warnings.

[a] C# ranks 6th in the 2021 IEEE Spectrum ranking;[1] Java ranks 2nd.

- A dataset [21] with 4.6 M warnings flagged by SonarQube and ReSharper on 100 open-source C# projects.[2]
- An open-source implementation of the EagleRepair technique supporting 20 ReSharper and SonarQube rules available at https://marodev.github.io/EagleRepair/.

## II. Background: Static Analysis Tools

Static analyzers are automatic tools that process a project's source code and report anomalies, bad smells, and stylistic issues that are often indicative of bad programming practices.

SATs like SonarQube or ReSharper offer a range of *rules* that users can select for checking. Each rule corresponds to a pattern whose violations the SATs can identify. For example, SonarQube's rule 1186[3] requires that "Methods should not be empty"; whenever SonarQube finds a method with a completely empty body, it will report a *violation* of rule 1186. A SAT's rule violation report is also called a *warning* (or an issue); therefore, each warning corresponds to the violation of a certain rule at a certain location in the source code. *Fixing* a warning means modifying the source code so that the SAT no longer triggers that warning. For example, a fix for a violation of rule 1186 consists of adding some executable code or some comments to fill the empty method body.

## III. Original Studies

### A. Frequency of Static Analysis Warnings in Java

Most recent empirical studies of SAT usage in Java target the SonarQube static analyzer, which gained significant popularity among Java developers [14]. Therefore, the first part of our study (Sec. IV-B,V-B) performs, on C# projects, similar analyses as 3 recent studies [9], [14], [18] of SonarQube usage on Java projects. Lenarduzzi et al. [14] ran SonarQube on all commits of 33 Java projects from the Apache Software Foundation, collecting—among other things—1.8 M SonarQube warnings. Digkas et al. [9] analyzed how the maintainers of 57 open-source Java projects fix SonarQube issues. Marcilio et al. [18] mined SonarQube usage in 246 projects—including both large open-source projects and private government ones. Among their many findings, we focus on those about the rules that are most frequently violated in Java projects, which we recall in Sec. V-B.

### B. Automatically Fixing Static Analysis Warnings in Java

In previous work [19][b], we investigated the feasibility of automatically generating fixes that address static analysis warnings in Java projects. To this end, we designed and implemented SpongeBugs, a technique that can automatically fix 11 commonly used SonarQube rules.

*1) Original Approach:* SpongeBugs targets SonarQube rules that developers are more likely to violate and fix [9], [16], [18].

SpongeBugs's implementation uses Rascal, a domain-specific language for meta-programming. This dependency implies that SpongeBugs can only analyze source code compatible with Java version 8. SpongeBugs uses Rascal's pattern matching features to detect rule violations; therefore, it does not use SonarQube's output (and may detect rule violations slightly differently). More precisely, SpongeBugs performs matching in two steps: first, it does a textual "linear" matching that is fast but imprecise; then, it confirms any textual matching with a more accurate AST matching. To fix any detected violations, it instantiates pre-defined rule-specific fix templates using the information collected during matching.

*2) Original Research Questions:* SpongeBugs's evaluation targets three main research questions:[c]

*ORQ1.* How widely applicable is SpongeBugs?
*ORQ2.* Does SpongeBugs generate fixes that are acceptable?
*ORQ3.* How efficient is SpongeBugs?

ORQ1 looks into how many rule violations SpongeBugs detects and fixes; ORQ2 evaluates how many of fix suggestions, submitted as pull requests, were accepted by project maintainers; ORQ3 measures SpongeBugs's scalability in terms of running time on large codebases.

*3) Original Study Design:* SpongeBugs's evaluation targets 15 large open-source Java projects, selected using standard criteria [13] such as active projects, with several open issues, stars, and contributors. In addition, it only includes projects registered in SonarCloud (a cloud service used to run SonarQube), so as to ensure that SonarQube is routinely used in those projects.

To answer ORQ1, [19] first ran SonarQube on each selected project, collecting the violations of the 11 rules that SpongeBugs can fix. Then, it ran SpongeBugs on the same projects, collecting all the generated fixes. Finally, it ran SonarQube again on the project modified by applying all the fixes generated automatically by SpongeBugs, counting how many original warnings no longer appeared. By manually analyzing a sample of these experiments' output, the original study evaluated the correctness of fixes produced by SpongeBugs, and the false positive and false negative rates of its violation detection—using SonarQube's detection as ground truth.

To answer ORQ2, [19] identified which projects were interested in fixing SonarQube violations (12 out of 15 projects). It then submitted 38 pull requests to these projects, including 946 SpongeBugs-generated fixes complemented with manually-written natural language descriptions of the violated rules.

To answer ORQ3, [19] measured the running time of SpongeBugs on each project.

*4) Original Results:* Considering SpongeBugs's *applicability* (ORQ1), the original study found that it could automatically fix 85% of all violations detected by SonarQube (for the supported rules). All of SpongeBugs's fixes are syntactically correct (they compile without errors). Only 0.6% of all rule violations reported by SpongeBugs are false positives that do not correspond to SonarQube warnings (i.e., precision is 99%). In contrast, SpongeBugs misses 15% of all SonarQube

[b] We refer to the JSS journal publication [19], which extends the original study published at SCAM [20].

[c] A fourth research question is specific to the Java benchmark Defects4J [12], and we don't include it in our C# replication.

warnings (false negatives in SpongeBugs's detection, i.e., recall is 85%).

Considering the *acceptability* of SpongeBugs's fixes (ORQ2), the original study reports that the developers accepted 34 out of 38 submitted pull requests (including 825 SpongeBugs-generated fixes out of 946, or 87%).

Considering *scalability* (ORQ3), the original study reports that SpongeBugs takes 1.2 minutes to process 1,000 lines of code on average. Unsurprisingly, very large files and methods may take up a disproportionate amount of running time.

## IV. Our Replication Study Design

The overall aim of our study is replicating for C# some findings about static analysis in Java. Since C# features much less frequently than Java in empirical studies of open-source software, we first have to cover some groundwork to discover what static analysis tools are used by C# projects, which leads to our first research question:

**RQ1.** Which static analysis tools do open-source C# projects commonly use?

After identifying a few popular static analyzers for C#, we should understand which of their rules are more frequently violated in C# projects. This is captured by our second research question:

**RQ2.** What static analysis rules do C# projects more frequently violate?

After establishing this basic knowledge about static analysis tools and warnings in C#, we can address the main research question:

**RQ3.** Is it feasible to automatically generate fixes that address static analysis warnings in C# projects?

As we explain in detail in the rest of the section, answering RQ3 involves: 1) developing a technique to automatically fix static analysis warnings; 2) evaluating it on the same C# projects analyzed to answer RQ1 and RQ2; 3) assessing its applicability (RQ3.1), the acceptability of its fixes (RQ3.2), and its runtime efficiency (RQ3.3).

### A. Project Selection

To study static analysis usage "in the wild", we collect large and popular open-source C# projects. As customary in experiments based on mining software repositories, we assume that prominent open-source projects are more likely to be realistic and mature, following practices that are representative of professional software development—including using static analysis tools.

We first used the GHS online tool [7] to broadly query GitHub and retrieve 44,848 C# projects.[d] We then filtered the projects to select those with least 100 stars (a common heuristic to select "popular" projects [10]), and at least one of the following: issues, pull requests, forks. This led to 34,430 projects. We then selected non-archived projects with at least 15 contributors and some recent activity in the third quartile of the following GitHub metrics: commits, watchers, stars, total issues, forks, and pull requests. This led to 1,810 projects. Finally, we kept only projects that could be compiled in our build environment, that is need no more than 10 GB of disk space and 6 GB of RAM, and can be fully analyzed in no more than 6 hours. This finally led to 1,050 projects, from which we randomly selected 100 for the rest of the study. These 100 projects total over 31 M LOC; the average project size is 316 K LOC, the smallest project is 18 K LOC and the largest is 3.2 M LOC.

### B. Static Analysis Tools and Warnings

To answer RQ1, we searched the 100 projects' repositories for build configuration files that may indicate the regular use of static analysis tool: continuous integration files (e.g., `ci.yml`, `travis.yml`), build scripts (e.g., `build.sh`), and configuration files of static analysis tools (e.g., SonarQube's `sonar-project.properties`).

As Sec. V-A discusses in more detail, the analysis identified ReSharper and SonarQube as suitable subjects. In our experiments, we ran both tools with default settings on the latest commit all projects. At the time of this research, SonarQube for C# checks 269 rules when run in SonarCloud;ReSharper version 2021.2 checks over 700 rules for C#.[4]

To answer RQ2, we collected statistics about which rules were violated more frequently in the source code of the 100 C# projects. We paid heed to the static analyzers' classification of their many rules: ReSharper groups rules into 9 categories (BestPractice, CodeSmell, ConstraintViolation, DeclarationRedundancy, CodeRedundancy, LanguageUsage, CodeStyleIssues, FormattingIssues, and CompilerWarning),[5] whereas SonarQube groups rules into 3 categories (CodeSmell, Vulnerability, and Bug).[6]

### C. Automatically Fixing Violations

To address RQ3, we designed and implemented EagleRepair, a technique to automatically fix static analysis warnings in C# projects. EagleRepair can detect and fix violations of 15 ReSharper rules and 5 SonarQube rules.

Several ReSharper and SonarQube rules are similar, but there is no one-to-one mapping because the two tools use rules of different granularity. The 5 SonarQube rules roughly correspond to 12 ReSharper rules: in particular, 7 different ReSharper rules capture a series of issues that can affect LINQ expressions; SonarQube lumps all of these issues in 1 rule. To uniformly detect and fix warnings from both static analyzers, EagleRepair summarizes these $17 = 5 + 12$ rules by means of 9 *fix templates*, which capture both the violations of a rule (or group of similar rules) and the modifications needed to fix such violations.

When run on a project, EagleRepair processes all source code referenced in the project's `.sln` *solution file*. After generating an AST of the source code by compilation, EagleRepair performs the following step for each template:

1) It finds all template matches, which correspond to rule violations;

[d] Sample collected in March 2021.

2) It uses the matching information to modify the AST so that it fixes the violation;
3) It checks that the fixes are well-formed.

A fix is well-formed if it corresponds to code that compiles correctly. EagleRepair's final output consists of the project's source code patched with all fixes that pass the checks.

EagleRepair's implementation uses the Roslyn C# Compiler;[7] this entails that every C# program that can be compiled with the standard compiler can also be analyzed by EagleRepair. Similarly to SpongeBugs, EagleRepair does not use the output of ReSharper or SonarQube but independently and directly checks for violations of their rules. As we observed in SpongeBugs, this means that EagleRepair's detection may incur false positives and false negatives (with respect to ReSharper's or SonarQube's).

To evaluate EagleRepair, we ran it on the 100 C# project that we also used to study static-analysis warnings. In each run, we collected EagleRepair's generated fixes and we compared them to the warnings reported by SonarQube and ReSharper on the same projects. More precisely, we address the following research questions, which mirror those used in SpongeBugs's empirical evaluation Sec. III-B.

*Applicability:* To answer RQ3.1:

**RQ3.1** How widely applicable is EagleRepair?

we manually analyzed a sample of its fixes to determine their correctness, and how many false positives and false negatives it incurs (using SonarQube and ReSharper as ground truth).

*Acceptabilty:* To answer RQ3.2:

**RQ3.2** Does EagleRepair generate fixes that are acceptable?

we selected 281 automatically-generated fixes and submitted them as 27 pull requests of the corresponding projects. This indicates whether developers find EagleRepair's fixes meaningful and of quality sufficient to be included in their codebase.

*Efficiency:* To answer RQ3.3:

**RQ3.3** How efficient is EagleRepair?

we measured the running time of EagleRepair on the 100 projects. We used C#'s Stopwatch[8] to measure running time.

## V. Replication Results: Static Analysis Tools and Warnings in C#

### A. Static Analysis Tools Used by C# Projects

Tab. I shows the results of our analysis of which SATs are used by the 100 C# projects we selected. ReSharper is clearly the most widely used, with 28 projects that include a configuration file for running it. ReSharper's popularity is consistent with the findings of a survey of 375 Microsoft developers [6], which indicated ReSharper as the most widely used code analyzer.

CodeQL[9], Dependabot,[10] and StyleCop[11] are considerably less used than ReSharper but still significantly so. CodeQL and Dependabot are part of GitHub's ecosystem,[12,13] and hence they are easy to integrate into GitHub pipeline workflows; since we only considered projects on GitHub, this may be part of the reason for their significant usage. Compared to tools like ReSharper, CodeQL and Dependabot are more specialized to find specific issues: CodeQL automates security checks, whereas Dependabot detects insecure and out-of-date dependencies. StyleCop[14] is a static analyzer that focuses on style consistency rules (as the name suggests); ReSharper includes most of StyleCop's rules together with many others that are not just about stylistic conventions.[15] CredScan[16] is another specialized tool (developed by Microsoft), which detects sources of credential leaks in source code and configuration files; it was used by 3 projects in our sample. Finally, Java developers' favorite SonarQube was definitely used by only 4 C# projects; this may indicate that it is not as popular among C# developers as it is in Java.

| **Tool** | **% of Projects Using Tool** |
|---|---:|
| ReSharper | 28 % |
| CodeQL | 10 % |
| Dependabot | 10 % |
| StyleCop | 8 % |
| SonarQube | 4 % |
| CredScan | 3 % |

TABLE I: For each static analysis tool, the percentage of the 100 C# projects used in our replication that use that tool.

> *ReSharper is the most widely used static analyzer in our C# projects. SonarQube is considerably less popular in C#.*

Following these results, we will focus on ReSharper and SonarQube for the rest of our study. We consider ReSharper because it's clearly the most popular tool among C# projects; but we also include SonarQube so that we can more directly compare our results to those about Java. In contrast, the other tools used by our projects are too specialized to support a meaningful comparison with SonarQube's usage in Java.

### B. Static Analysis Warnings in C#

*Warnings:* As shown in Tab. II, running ReSharper and SonarQube reported millions of warnings for our 100 projects. ReSharper reported more than 9 times more warnings than SonarQube (3.8 M vs. 413 K), which may partly be due to the fact that ReSharper's rules are designed specifically for C#, whereas SonarQube aims at being a multi-language tool. The distribution of warnings per projects has a long tail of 6–8 standard deviations for both tools.

Unsurprisingly, a project's size is often a proxy for how many warnings the SATs will report. Take QuickFIXn[17] as an example: it is the largest project in our dataset (2.3 M LOC), and also first in terms of number of ReSharper warnings (over 939 K) and fifth in terms of number of SonarQube warnings (over 23 K). Upon closer inspection, we found that a disproportionate fraction of all warnings in this project (97% of ReSharper's and 96% of SonarQube's warnings) are due to 585 files (such as `Fields.cs`[18]) that repeat the same rule violations over and over. The comments and structure of all these files clearly indicate that they were automatically generated using inconsistent naming. Symmetrically, projects that trigger a small number of warnings tend to be smaller, such as project aliyun-open-api-netsdk[19], which is the second smallest in our dataset and also second in terms of the fewest warnings from ReSharper.

In contrast, projects that diligently heed the recommendations of static analyzers as an integral part of their development process tend to trigger fewer warnings, even relative to their size. For instance, MaterialDesignInXamlToolkit's[20] contribution guidelines prominently mention the usage of ReSharper;[21] it is then unsurprising that its size approximately matches the 80% percentile of all project sizes, but triggers the smallest number of ReSharper warnings (and the 5th smallest of SonarQube warnings) among all projects.

TABLE II: Warnings reported by ReSharper and SonarQube on the 100 C# projects. All numbers are in *thousands*.

| Tool | Mean | Std | Min | Med | Max | Total |
|---|---|---|---|---|---|---|
| **Per Project** | | | | | | |
| ReSharper | 39.5 | 116.4 | 0.236 | 13.0 | 939.0 | 3800.0 |
| SonarQube | 4.1 | 8.9 | 0.048 | 1.4 | 58.3 | 413.6 |
| **Per Rule** | | | | | | |
| ReSharper | 8.0 | 50.3 | 0.001 | 0.2 | 881.7 | 3800.0 |
| SonarQube | 1.9 | 4.2 | 0.001 | 0.2 | 22.8 | 413.6 |

*ReSharper vs. SonarQube Rules:* If we break down the warnings by violated rule (in all projects), we find that ReSharper reported warnings violating 481 of its over 700 default rules, and SonarQube reported reported warnings violating 211 rules of its 269 default rules.

ReSharper offers more rules compared to SonarQube because its rules feature a combination of finer granularity and more extensive checks. Finer granularity means that ReSharper often breaks down a single SonarQube rule into multiple more specialized rules. For instance, SonarQube rule S2971[22] checks several features of LINQ `IENumerable` queries that should be simplified; ReSharper breaks down the same checks into 7 different rules, each targeting one feature that can be simplified (e.g., `Where`, `Count`, etc.).

More extensive checks means that ReSharper sometimes lifts to a broader scope some checks that SonarQube only performs locally. For example, SonarQube's rule S1144[23] checks that each source file includes no unused private types or members; this check is performed locally on each file independent of the others. ReSharper includes two rules, `UnusedMember.Local`[24] and `UnusedMember.Global`,[25] that correspond to the same check that there are no unused types or members. `UnusedMember.Local` has the same single-file scope as SonarQube's matching rule; `UnusedMember.Global` applies the same check to non-private types or members and across all files. Note that using more extensive rules give more flexibility but can also backfire: for example, ReSharper's documentation remarks that rule `UnusedMember.Global` may be inapplicable to libraries or APIs that export members to public clients (but have no clients within the project). These global rules offer annotations to suppress warnings in these cases;[26] indeed, we found 10 projects (e.g., eddi[27]) using these annotations.

To perform a systematic mapping between SonarQube and ReSharper rules, we first considered all warnings reported by the two analyzers that flag lines that are very close in the project's source code (precisely, they flag lines $\ell_1, \ell_2$ such that $|\ell_1 - \ell_2| \leq 2$). Then, we went through several hundred of such pairs of warnings, and read the warnings' natural-language messages to determine if they correspond to a violation of the same condition. With this process, we managed to map 35 ReSharper rules to 21 SonarQube rules: 1 SonarQube rule corresponds to 7 ReSharper rules; each of 2 SonarQube rules corresponds to 5 ReSharper rules; the remaining 18 $(= 21-1-2)$ SonarQube rules match 1-to-1 the remaining 18 ReSharper rules $(= 35 - 1 \cdot 7 - 2 \cdot 5)$.

> *ReSharper's rules often are finer-grained and more extensive than SonarQube's. 35 ReSharper rules correspond to 21 SonarQube rules.*

### C. Static Analysis Warnings: C# vs. Java

SonarQube's rules are classified into 4 categories: Bugs, Vulnerabilities, Security Hotspots, and Code Smells. Previous work [9], [14], [18] (see Sec. III-A) consistently found that Code Smells rules are disproportionately violated in Java projects: 96% of all warnings in 33 Apache projects correspond to violations of Code Smells rules [14]; 55% of the fixes analyzed in [9] and 80% of the fixes analyzed in [18] fix violations of Code Smells rules.

In order to understand whether a similar finding applies to C# projects, we looked at the top-10 most frequently violated rules in C# project. All top-10 most frequently violated SonarQube rules (Tab. IV) indeed belong to category Code Smells. As for the top-10 most frequently violated ReSharper rules (Tab. III), we tried to find which of them to SonarQube's Code Smells, which are "a maintainability-related issue[s] in the code". ReSharper has its own categorization of rules into 9 categories, one of which is indeed called "CodeSmell"; one of the rules belong to this category. In addition, we further investigated the other 9 rules in Tab. III: 6 are equivalent to SonarQube rules that belong to category Code Smells; 4 have no direct SonarQube equivalent rule, but all correspond to checks that SonarQube commonly classifies as Code Smells (redundant qualifiers, style and formatting issues).

Since SonarQube was used in our study as well as in the original studies [14], [18]—albeit on projects written in different languages—we can directly see which of the most frequently violated rules in our C# experiments also feature prominently in the Java experiments.[f] Tab. V includes all SonarQube rules that were violated at least once in our experiments, as well as in the Java studies [14], [18]. While the absolute frequencies can vary considerably, there is overall a remarkable similarity between the frequency of violations of the same rules in different languages.

Exceptions to this trend are also interesting to discuss. Violations of three SonarQube rules (S1199, S1481, and S1905 in Tab. V) feature proportionally much more frequently in C# than in Java. Two of them ("nested blocks should not be

[f] Note that SonarQube for Java's default ruleset includes 485 rules, significantly more than C#'s 269 rules.

TABLE III: Top-10 most frequently violated ReSharper rules in C# projects: number of warnings (**Counts**) and percentage of all warnings (**%**).

| **TypeId** | **CategoryId** | **Severity** | **Description** | **Counts** | **%** |
|---|---|---|---|---|---|
| RedundantNameQualifier | Code Redundancy | Warning | Redundant name qualifier | 881,541 | 25.46 |
| UnusedMember.Global | Declaration Redundancy | Suggestion | Type member is never used: Non-private accessib... | 421,576 | 12.17 |
| InconsistentNaming | Constraint Violation | Warning | Inconsistent Naming | 346,990 | 10.02 |
| MemberCanBePrivate.Global | Best Practice | Suggestion | Member can be made private: Non-private accessi... | 187,061 | 5.40 |
| ArrangeAccessorOwnerBody | Code Style Issues | Suggestion | Use preferred body style: Convert to property, ... | 144,293 | 4.17 |
| BadControlBracesIndent | Formatting Issues | Suggestion | Incorrect indent: Around statement braces | 126,693 | 3.66 |
| RedundantAssignment | Code Redundancy | Warning | Assignment is not used | 112,507 | 3.25 |
| RedundantUsingDirective | Code Redundancy | Warning | Redundant using directive | 86,862 | 2.51 |
| SuggestVarOrType_BuiltInTypes | Code Style Issues | Hint | Use preferred 'var' style: For built-in types | 81,457 | 2.35 |
| VirtualMemberCallInConstructor | Code Smell | Warning | Virtual member call in constructor | 55,096 | 1.59 |
| **Total** | | | | **2,444,076** | **70.58** |

TABLE IV: Top-10 most frequently violated SonarQube rules in C# projects: number of warnings (**Counts**) and percentage of all warnings (**%**).

| **Squid** | **Severity** | **Type** | **Title** | **Counts** | **%** |
|---|---|---|---|---|---|
| **S3776** | Critical | CS[e] | Cognitive Complexity of methods should not be t... | 22,836 | 5.52 |
| S1104 | Minor | CS | Fields should not have public accessibility | 22,466 | 5.43 |
| S1905 | Minor | CS | Redundant casts should not be used | 19,553 | 4.73 |
| **S125** | Major | CS | Sections of code should not be commented out | 18,007 | 4.35 |
| S101 | Minor | CS | Types should be named in PascalCase | 17,084 | 4.13 |
| S4136 | Minor | CS | Method overloads should be grouped together | 16,818 | 4.07 |
| **S112** | Major | CS | General exceptions should never be thrown | 15,660 | 3.79 |
| S2933 | Major | CS | Fields that are only assigned in the constructo... | 15,486 | 3.74 |
| S1481 | Minor | CS | Unused local variables should be removed | 13,418 | 3.24 |
| S927 | Critical | CS | Parameter names should match base declaration a... | 12,656 | 3.06 |
| **Total** | | | | **173,984** | **42.06** |

used" and "redundant casts should not be used") may simply reflect different conventions in the usage of some language features; but "unused local variables should be removed" seems sound advice in any language—and hence it's unclear why C# programmers seem to violate this rule at higher rates.

Conversely, some rules feature more prominently in Java than in C#. SonarQube's Java rule "unnecessary imports should be removed" is among the top-10 most frequently violated rules in [18] but this rule[g] is not not checked by SonarQube on C# projects by default, and hence it doesn't feature in our experiments. ReSharper does check an equivalent rule `RedundantUsingDirective`, which is in fact the 8th most frequently violated in our dataset. Another interesting example concerns the keyword `var`, which declares variables with implicit typing in both C# (since version 3 of the language) and Java (since version 10). SonarQube for Java includes by default a rule[28] that signals cases where introducing `var` can be useful, which is triggered frequently [18]. In contrast, SonarQube developers intentionally did not provision a similar rule for C#,[29] claiming that Visual Studio already performs a similar check. Nevertheless, ReSharper offers a similar rule `SuggestVarOrType`, which is indeed the 9th most frequently violated rules in our experiments.

In all, there is a clear similarity between the static analysis rules that are most frequently violated by C# projects and those violated by Java projects.

*The most frequently violated kinds of static analysis rules are similar in Java and C# and often correspond to maintainability-related issues of the code ("code smells").*

## VI. Replication Results: Automatically Fixing Static Analysis Warnings in C#

This section describes the empirical evaluation of EagleRepairon the 100 C# projects in our dataset (Sec. IV-A), thus answering RQ3 and, specifically, RQ3.1–3 (Sec. IV-C).

### A. Fixed Rules

EagleRepair implements the 9 fix templates in Tab. VI, which it uses to detect and fix violations to 15 ReSharper rules and 5 SonarQube rules. We selected the static analysis rules to fix among those that were violated in our experiments (Sec. V-B), including several rules that both ReSharper and SonarQube support. In addition, we only considered rules that can be fixed with modifications local to the source file where the rule is violated, and do not entail modifying program behavior. These criteria were also at the basis of SpongeBugs, since they allow us to select rules that are amenable to being fixed correctly and automatically. Our selection of rules to fix is neither exhaustive nor unique, but it corresponds to a reasonably varied selection that demonstrates fixing different kinds of rules.

### B. RQ3.1: Applicability

We assess EagleRepair's applicability in terms of the correctness, precision, and recall of its fixes. An EagleRepair fix is correct if it modifies a program's source code so that

[g] More precisely, its C# counterpart "unused `using` should be removed".

TABLE V: SonarQube rules that were violated at least once in both our study and in the Java studies [14], [18]. Percentages correspond to the percentage of all warnings in each dataset; rule descriptions are from SonarQube's C# documentation.

| Squid | Description | [18] (Java) % | [14] (Java) % | Our study (C#) % |
|---|---|---|---|---|
| S1066 | Collapsible "if" statements should be merged | 0.511 | 0.607 | 2.083 |
| S1118 | Utility classes should not have public construc... | 0.801 | 0.513 | 0.510 |
| S1125 | Boolean literals should not be redundant | 0.074 | 0.555 | 2.145 |
| S1134 | Track uses of "FIXME" tags | 0.234 | 0.093 | 0.080 |
| S1135 | Track uses of "TODO" tags | 1.900 | 1.147 | 2.343 |
| S1155 | "Any()" should be used to test for emptiness | 0.500 | 1.142 | 0.151 |
| S1163 | Exceptions should not be thrown in finally blocks | 0.045 | 0.049 | 0.006 |
| S1168 | Empty arrays and collections should be returned... | 0.381 | 0.339 | 0.494 |
| S1172 | Unused method parameters should be removed | 1.056 | 0.647 | 1.024 |
| S1185 | Overriding members should do more than simply c... | 0.165 | 0.066 | 0.082 |
| S1186 | Methods should not be empty | 1.634 | 1.769 | 1.894 |
| S1199 | Nested code blocks should not be used | 1.634 | 0.812 | 2.950 |
| S1206 | "Equals(Object)" and "GetHashCode()" should be ... | 0.086 | 0.039 | 0.110 |
| S1210 | "Equals" and the comparison operators should be... | 0.019 | 0.058 | 0.083 |
| S1215 | "GC.Collect" should not be called | 0.006 | 0.101 | 0.020 |
| S1479 | "switch" statements should not have too many "c... | 0.067 | 0.003 | 0.034 |
| S1481 | Unused local variables should be removed | 0.329 | 0.499 | 3.240 |
| S1764 | Identical expressions should not be used on bot... | 0.021 | 0.020 | 0.040 |
| S1862 | Related "if/else if" statements should not have... | 0.005 | 0.002 | 0.005 |
| S1905 | Redundant casts should not be used | 0.241 | 0.146 | 4.727 |
| S2178 | Short-circuit logic should be used in boolean c... | 0.010 | 0.005 | 0.083 |
| S2183 | Integral numbers should not be shifted by zero ... | 0.015 | 0.052 | 0.013 |
| S2184 | Results of integer division should not be assig... | 0.199 | 0.294 | 0.034 |
| S2225 | "ToString()" method should not return null | 0.004 | 0.026 | 0.007 |
| S2275 | Composite format strings should not lead to une... | 0.101 | 0.004 | 0.002 |
| S2326 | Unused type parameters should be removed | 0.054 | 0.029 | 0.051 |
| S2386 | Mutable fields should not be "public static" | 0.282 | 0.118 | 1.285 |
| **Median** | | 0.165 | 0.101 | 0.083 |
| **Total** | | 10.4 | 9.1 | 23.5 |

TABLE VI: The 9 fix templates implemented by EagleRepair correspond to 15 ReSharper rules and 5 SonarQube rules.

| # | ReSharper | SonarQube |
|---|---|---|
| R1 | MergeSequentialChecks | |
| R2 | MergeSequentialChecks | S4201 |
| R3 | ReplaceWith*[LINQ feature]* (8× ) | S2971 |
| R4 | MergeCastWithTypeCheck | S3247 |
| R5 | UseMethodAny | S1155 |
| R6 | UseNullPropagation | |
| R7 | UsePatternMatching | |
| R8 | UseStringInterpolation | |
| R9 | ReplaceWithStringIsNullOrEmpty | S3256 |

TABLE VII: EagleRepair's precision and recall in our sample for each ReSharper and SonarQube rule its fix templates cover.

| Template | ReSharper/SonarQube Rule | Precision | Recall |
|---|---|---|---|
| R4 | MergeCastWithTypeCheck | 95 % | 37 % |
| R1–2 | MergeSequentialChecks | 67 % | 33 % |
| R3 | ReplaceWithOfType | 100 % | 5 % |
| R3 | ReplaceWithSingleCallToAny | 100 % | 68 % |
| R3 | ReplaceWithSingleCallToCount | 10 % | 78 % |
| R3 | ReplaceWithSingleCallToFirst | 90 % | 48 % |
| R3 | ReplaceWithSingleCallToFirstOrDefault | 100 % | 68 % |
| R3 | ReplaceWithSingleCallToLast | 100 % | 50 % |
| R3 | ReplaceWithSingleCallToSingle | 100 % | 23 % |
| R3 | ReplaceWithSingleCallToSingleOrDefault | 90 % | 63 % |
| R9 | ReplaceWithStringIsNullOrEmpty | 30 % | 37 % |
| R5 | UseMethodAny | 70 % | 88 % |
| R6 | UseNullPropagation | 76 % | 35 % |
| R7 | UsePatternMatching | 76 % | 74 % |
| R8 | UseStringInterpolation | 85 % | 84 % |
| R5 | S1155 | 100 % | 94 % |
| R3 | S2971 | 100 % | 75 % |
| R4 | S3247 | 62 % | 77 % |
| R9 | S3256 | 20 % | 24 % |
| R2 | S4201 | 90 % | 79 % |
| **Total** | | **78 %** | **58 %** |

it no longer violates a static analysis rule and its behavior does not change. A false positive is a warning reported (and fixed) by EagleRepair that is reported by neither ReSharper nor SonarQube. EagleRepair's precision is thus the percentage of its fixes that are *not* false positives. A false negative is warning reported by ReSharper or SonarQube that EagleRepair missed (and hence didn't fix). EagleRepair's recall is thus the percentage of all ReSharper or SonarQube warnings that are *not* false negatives. We measure precision and recall for each of EagleRepair's *fix templates*, which correspond to one or several SonarQube and ReSharper rules.

In our experiments, EagleRepair produced a total of 31,394 fixes for the 100 analyzed projects. We manually analyzed a sample of 460 of these fixes to estimate correctness and precision. This sample size is sufficient to estimate precision with up to 5% error and 95% probability with the most conservative (i.e., 50%) a priori assumption [8]. We selected the 460 fixes using stratified sampling: first, we classified all fixes into 9 strata according to the fix template they correspond to; then, we sampled randomly in each stratum a number of fixes proportional to the relative size of the stratum (compared to the total number of fixes) with a lower bound of 10 fixes per template; finally, we added a few fixes to the sample to include at least 1 fix for every project with any fixes.

Thanks to its precise matching and well-formedness checks (see Sec. IV-C), EagleRepair achieves a high *correctness*: only 16 (3%) of the 460 manually inspected fixes were incorrect, that is alter program behavior in unintended ways.

EagleRepair's incomplete handling of string interpolation is responsible for 4 of these incorrect fixes, generated by fixing template R8 (which was correctly applied in 105 other cases in our sample). For simplicity, EagleRepair fills in the formal parameters in a format string in the same order as the parameters appear in the string, without processing the parameters' name; in expressions like `string.Format("{1},{0}")`, the first actual parameter would fill in the first formal parameter `{1}` even though it should fill in the second one `{0}`. Another source of a few incorrect fixes is when EagleRepair matches fix template R9 too eagerly, and replaces a check `s != ""` with `!string.IsNullOrEmpty(s)` which is equivalent to the stronger check `s != null && s != ""`. In a few cases, the program really did intend to allow `null` as a valid value for `s`; in these cases, EagleRepair's fixes modify program behavior. It's interesting that this kind of incorrect fix is similar to some produced by SpongeBugs for Java's SonarQube rule "Strings should be compared using equals()", which may also change program behavior when `null` is involved.

Tab. VII summarizes EagleRepair's precision and recall for each rule. The overall precision is 78%, which indicates that EagleRepair's templates match SonarQube's and ReSharper's detection fairly accurately. ReSharper and SonarQube reported 48,999 warnings; EagleRepair managed to detect and fix 28,417 of them, which gives an overall recall of 58%. When EagleRepair incurs a false positive, it means it will generate a fix that is not needed according to ReSharper's or SonarQube's detection algorithm; it does *not* mean that it introduces a bug: since most of EagleRepair's fixes do not alter behavior (they are correct), it simply refactors the code in a way that may not be a stylistic improvement. On the other hand, when EagleRepair incurs a false negative, it simply means that it won't automatically fix a static analysis warning.

Let's now look into some of the rules where recall or precision are lower than average. ReSharper's rule `ReplaceWithOfType` has the lowest recall of 5%: EagleRepair only detected and fixed 1 of 20 violations reported by ReSharper. This rule is combined with 7 other similar rules in EagleRepair's fix template R3, which deals with anti-patterns in LINQ expressions. Since the rule isn't violated very often compared to others, we overlooked some possible cases when implementing EagleRepair's template, which is the one reason for this low recall. Other similar rules (all corresponding to template R3) also have lower recall than most other rules; this is often due to limitations in EagleRepair's matching capabilities, which we occasionally focused on the most frequently occurring features. For instance, this snippet is a violation of rule `ReplaceWithSingleCallToSingle`

```
return fields.Where(f => f.Name == valueName)
    .Where(f => f.age >= threshold)
    .Single();
```

which EagleRepair didn't detect because it cannot match expressions with several chained method calls.

Another limitation of EagleRepair explains its low recall with fix template R9, corresponding to ReSharper rule `ReplaceWithStringIsNullOrEmpty` and SonarQube rule S3256. The static analyzers can detect violations of this rule (which essentially boils down to using C#'s static method `!string.IsNullOrEmpty(s)` instead of the expression `s != null && s != ""`) even when they involve constants declared in a different module, such as in the conditional expression `if (!_graphvizdir.Equals(String.Empty))`, where `String.Empty` is a synonym of `""`. EagleRepair, in contrast, does not follow references to constants declared in other classes, and hence fails to detect such instances.

When EagleRepair has a low precision, it may be due to overlapping static analysis rules. For instance, Boolean expression `cad != null && cad.MethodInfo != null` violates ReSharper rules `UsePatternMatching` and `MergeSequentialChecks`. ReSharper only reports a violation of the latter, whereas EagleRepair detects and fixes a violation of the former. As a result, this is both a false positive and a false negative for EagleRepair using ReSharper's output as ground truth.

Another limitation of EagleRepair that is a source of false positives is that it does not take the language version into account to perform its checks. For example, ReSharper only checks rule `UsePatternMatching` (corresponding to EagleRepair's template R7) in projects written in C# version 7 or higher, since pattern matching features where introduced in that version of the language. EagleRepair checks the same rule even with older C# projects, which determines false positives (and, in a couple of cases, incorrect fixes).

> *EagleRepair automatically fixed 31,394 ReSharper and SonarQube warnings, with overall (sampled) correctness of 97%, precision of 78%, and recall of 58%.*

It's questionable that EagleRepair's fixing results can be quantitatively compared to SpongeBugs's, since the two tools have been evaluated on completely different projects and languages. Nevertheless, we note that EagleRepair's precision and recall are lower than SpongeBugs (99% precision and 85% recall). EagleRepair's difference is mainly due to a few rules whose detection is tricky, often because they combine SonarQube and ReSharper similar rules that are detected differently by the two SATs; SpongeBugs exclusive focus on SonarQube may have helped in this regard.

### C. *RQ3.2: Acceptability*

We assess the acceptability of EagleRepair's fixes as done in SpongeBugs's original study: we submit some of the fixes as pull requests and record how many developers accept.

Tab. VIII summarizes the content of the 27 pull requests collecting 281 EagleRepair fixes to the source code of various projects. We selected projects and fixes to submit in order to include popular and mature C# repositories (including several components of .NET's core framework), and all of EagleRepair's fix templates.

Developers accepted and merged 24 of the 27 pull requests (including 250 fixes out of 281); the remaining 3 pull requests were ignored by the project maintainers. All accepted pull requests were accepted without modifications, except for PR19, where the project maintainers asked us to remove a single

TABLE VIII: Summary of the 27 pull requests collecting several of EagleRepair's fixes submitted to various C# projects.

| | Category | Organization | #Stars | Pull Request URI | Fixed template | #Fixes | Status |
|---|---|---|---|---|---|---|---|
| PR1 | Framework | ABP | 6,200 | abpframework/abp/pull/9032 | R1, R6, R7, R8 | 5 | Merged |
| PR2 | Search Engine | Apache Foundation | 1,604 | apache/lucenenet/pull/488 | R5 | 3 | Merged |
| PR3 | SDK | Dropbox | 277 | dropbox/dropbox-sdk-dotnet/pull/240 | R6, R8 | 2 | Merged |
| PR4 | Backup Solution | Duplicati | 6,200 | duplicati/duplicati/pull/4511 | R8 | 9 | Merged |
| PR5 | Database | EventStore | 4,183 | EventStore/EventStore/pull/2961 | R1 | 9 | Merged |
| PR6 | Platform | GrandNode | 39 | grandnode/grandnode2/pull/10 | R4 | 1 | Merged |
| PR7 | SDK | Microsoft | 387 | microsoft/ApplicationInsights-dotnet/pull/2268 | R7 | 29 | Merged |
| PR8 | Concurrency Tool | Microsoft | 944 | microsoft/coyote/pull/173 | R1, R6, R7 | 5 | Merged |
| PR9 | Service for MS Azure | Microsoft | 687 | microsoft/fhir-server/pull/1942 | R1, R5, R6, R7, R8 | 8 | Merged |
| PR10 | Code Analyzer | Microsoft | 341 | microsoft/infersharp/pull/60 | R7 | 1 | Merged |
| PR11 | Library | Microsoft | 904 | microsoft/RulesEngine/pull/137 | R9 | 1 | Merged |
| PR12 | SDK | Microsoft Azure | 389 | Azure/azure-cosmos-dotnet-v3/pull/2470 | R1, R4, R6 | 21 | Merged |
| PR13 | IoT | Microsoft Azure | 1,189 | Azure/iotedge/pull/4991 | R3 | 18 | Merged |
| PR14 | Database | RavenDB | 2,718 | ravendb/ravendb/pull/12157 | R4 | 26 | Merged |
| PR15 | Video Game | Regalis11 | 605 | Regalis11/Barotrauma/pull/5772 | R6 | 48 | Merged |
| PR16 | Video Game | ServUO | 390 | ServUO/ServUO/pull/4935 | R9 | 25 | Merged |
| PR17 | Framework | .NET Foundation | 3,865 | akkadotnet/akka.net/pull/5013 | R3 | 1 | Merged |
| PR18 | Framework | .NET Foundation | 1,355 | dotnet/infer/pull/341 | R3 | 2 | Merged |
| PR19 | Database | .NET Foundation | 10,200 | dotnet/efcore/pull/24951 | R2, R8 | 15 | Merged |
| PR20 | Framework | .NET Foundation | 9,900 | dotnet/maui/pull/1059 | R2 | 3 | Merged |
| PR21 | .NET extensions | .NET Foundation | 4,941 | dotnet/reactive/pull/1543 | R6 | 4 | Merged |
| PR22 | Compiler Platform | .NET Foundation | 1,048 | dotnet/roslyn-analyzers/pull/5120 | R1, R3, R6 | 5 | Merged |
| PR23 | Framework | .NET Foundation | 7,600 | dotnet/orleans/pull/7080 | 9 | 3 | Merged |
| PR24 | Library | ElasticSearch | 3,070 | elastic/elasticsearch-net/pull/5691 | R8 | 6 | Merged |
| PR25 | Concurrency Tool | Microsoft | 376 | microsoft/AMBROSIA/pull/119 | R7, R8, R9 | 8 | Inactive |
| PR26 | Service for MS Azure | Microsoft | 1,174 | microsoft/azure-pipelines-agent/pull/3404 | R6 | 12 | Inactive |
| PR27 | Entertainment | Sonarr | 6,400 | Sonarr/Sonarr/pull/4490 | R6 | 11 | Inactive |

fix that targeted a method that could misbehave because it was dynamically interpreted rather than statically compiled; according to the maintainers, this peculiar usage could have interfered with the fix to generate unintended side effects.

An interesting case is PR2, which included 3 fixes to Apache's Lucene search engine. The repository is a C# port of the main Java version; therefore, the developers were initially hesitant to modify the C# port independent of the Java "parent" version. Nevertheless, they eventually accepted the pull request since EagleRepair's fixes provide a "real performance benefit".[30]

*Developers accepted 89% of all 281 EagleRepair fixes submitted as pull requests to their respective projects.*

### D. RQ3.3: Efficiency

To assess EagleRepair's efficiency we simply measure its running time in relation to the projects' size. Fig. 1 visualizes the measured running time in each project, and relates it to the project's size (bubble size) and number of detected and fixed issues (vertical coordinate). Most projects, including some very large ones such as QuickFIXn[31] (2.4 M LOC) and nHapi[32] (2.1 M LOC), were processed within 100 seconds. The project taking the most time was, however, ServUO,[33] which is a fairly small project in our selection (50 K LOC) but took nearly 425 seconds to be analyzed with EagleRepair.

We found that the best predictor of EagleRepair's running time on a project is the number of semantic model accesses that are required by the parsing step (see Sec. IV-C). This is a step that affects all project's code; after parsing is completed, EagleRepair's fix templates can be matched quickly and locally to each component.

*EagleRepair scales to projects of realistic size, which it can process in 100–400 seconds.*

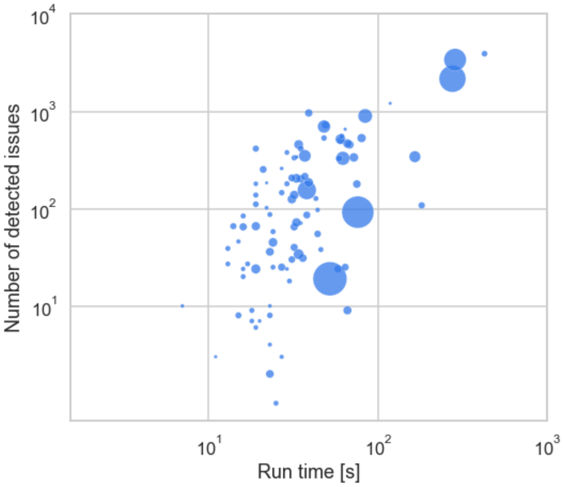

Fig. 1: EagleRepair's running time on the 100 selected projects. Each bubble corresponds to a project in the dataset; the bubble's size is proportional to the project size (as measured by SonarCloud); the bubble's $x$ position measures EagleRepair's runtime on the project; the bubble's $y$ position measures the number of detected and fixed warnings.

## VII. Threats to Validity

Our study of SAT usage and warnings in C# is a *partial* replication, which only determined which SATs are used in C# open-source projects and which warnings they more frequently triggered. We did not try to replicate several other results of the original Java studies [9], [14], [18] such as which static analysis warnings developers manually *fix* more frequently. We included SonarQube for C# in our comparison to meaningfully compare some of the results for Java (which also target SonarQube). However, SonarQube's popularity in C# seems considerably lower than its popularity in Java, which may limit the generality of some of our comparative findings.

Our selection of 100 popular open-source C# projects may not be representative of the whole C# ecosystem. A larger-scale analysis, including proprietary software, may reveal different SAT usage habits. Our detection of SAT usage may have missed projects that use non-standard configuration files.

Our analysis of EagleRepair's applicability was based on a sample of the generated fixes. We followed a multistage sampling strategy [2] (a stratified random sampling combined with a statistically significant sample size) to make the sample representative. While we double-checked our manual analysis, we cannot exclude that some errors in the correctness analysis remain; given the very low number of wrong fixes that we found, we doubt they would significantly affect the results. Our analysis of acceptability did not interview or survey developers to better understand the usefulness of EagleRepair's submitted fixes. Our analysis of EagleRepair's efficiency is simply based on running time; however, we did not find any glaring scalability problem.

The original study [19] that we replicated for C# was done by a similar team: this paper's second and third authors are also authors of [19]. A fully independent replication may have warranted a wider generalizability of the results.

## VIII. Related Work

We summarize the main related work in two areas: empirical studies of practitioners' usage of SATs (besides the three studies [9], [14], [18] that we partially replicated); and automatically generating fix suggestions.

### A. *Practical Usage of SATs*

One of the few multi-language study of SAT usage in open-source projects [5] targeted 122 popular open-source Java, JavaScript, Ruby, and Python projects. By manually analyzing the projects' websites and configuration files, the study found that almost half of the projects use at least one SAT. Since it is multi-language (and hence multi-tool), Beller et al.'s work [5] faces a similar problem we had, namely mapping similar rules across languages and tools. In our case, we could start from SonarQube's rules (many of which are available both in Java and in C#) and then match them to ReSharper's.

Continuous integration (CI) pipelines offer a convenient environment to empirically study SAT adoption. Rausch et al. [22] found that it's not uncommon that a single project uses multiple SATs; as a result, low code quality measures—typically measured by SATs—are often correlated with build failures. Zampetti et al. [28] report similar findings that correlate static analysis checks and build failures. Furthermore, they surveyed the SATs used by 20 GitHub Java projects with a CI pipeline; Vassallo et al. [26], [27] also report SonarQube as the most used SAT for Java; their findings are used on a combination of analysis of CI pipelines and interviews with industry practitioners. The methodological suggestion that we took from all these studies for our replication is to use configuration and build files to find which projects likely use SATs as part of their development process.

SATs' uptake by practitioners happened gradually and over recent years, and still has potential to further develop. In their 2013 work, Johnson et al. [11] noted that several factors limit a more widespread usage of SATs despite their undeniable benefits, and especially their sometimes logorrheic and imprecise (i.e., false positives) output, and the absence of automatic fix suggestions. Their observations motivated follow-up work to better characterize which SAT warnings developers find more useful, and how to help the process of fixing warnings. Our replication also contributes to this line of work.

As we mentioned in Sec. III-A, there is very little empirical work on SAT usage for C#. The only (partial) exception we could find is an empirical study of C# code smells [24], which however focuses on *design* anti-patterns rather than the more "syntactic" flaws that ReSharper and SonarQube (and hence EagleRepair) primarily target.

### B. *Automatic Fix Suggestions*

EagleRepair, like SpongeBugs, targets fixes that only involve changes that should not affect program behavior. EagleRepair and SpongeBugs's approach is based on bespoke template transformations, which generate fixes that are often correct-by-construction. Other approaches to generate syntactic fixes learn templates or other transformations from examples, such as in bug reports [15], commits, or bug fixes [23].

A few alternative approaches have been proposed to provide fix suggestions for SAT warnings; the bulk of the work targets Java and the SAT FindBugs [1]; note that SonarQube supersedes FindBugs by supporting many of its rules [25]. In contrast to the work described above, these works focus on violations of rules that are "semantics", and hence require to change program behavior to be fixed. Historical patterns of programmer-written fixes provide useful information to learn fix patterns [16], [17]. Barik et al. [3]'s approach uses a sort of differential testing, which compares the effect of different possible fixes for the same warning. Phoenix [4] is another system that learns fix patches for violations detected by FindBugs; it combines the fix pattern ingredients mined automatically from many revisions of 517 open-source projects to generate new fix suggestions to faults that have similar characteristics. Like EagleRepair, Phoenix's acceptability of fixes was assessed by submitting as pull requests a manual selection of fix suggestions generated by the tool.

## IX. Conclusions

We presented a replication for C# of studies of static analysis usage and automatic fixing in Java. Among other things, we confirmed that SATs are fairly used in C# open-source projects; that the most frequent static analysis warnings have to do with "code smells" which are similar in Java and C# projects; and that automatically fixing violations of static analysis rules is possible in C# as well with high correctness and good precision.

# References

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

[2] Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines, 2021. arXiv:2002.07764.

[3] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 211–221, Oct 2016. doi:10.1109/ICSME.2016.63.

[4] Rohan Bavishi, Hiroaki Yoshida, and Mukul 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, pages 613–624, New York, NY, USA, 2019. ACM. URL: http://doi.acm.org/10.1145/3338906.3338952, doi:10.1145/3338906.3338952.

[5] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy 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)*, volume 1, pages 470–481. IEEE, 2016.

[6] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2970276.2970347.

[7] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in GitHub for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 560–564. IEEE, 2021. doi:10.1109/MSR52588.2021.00074.

[8] Wayne W. Daniel. *Biostatistics: A Foundation for Analysis in the Health Sciences*. Wiley, 7 edition, 1999.

[9] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou. How do developers fix issues and pay back technical debt in the Apache ecosystem? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 153–163, March 2018. doi:10.1109/SANER.2018.8330205.

[10] Junxiao Han, Shuiguang Deng, Xin Xia, Dongjing Wang, and Jianwei Yin. Characterization and prediction of popular projects on GitHub. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, volume 1, pages 21–26. IEEE, 2019.

[11] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[12] René Just, Darioush Jalali, and Michael 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, pages 437–440, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2610384.2628055, doi:10.1145/2610384.2628055.

[13] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering*, 21(5):2035–2071, sep 2015. doi:10.1007/s10664-015-9393-5.

[14] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. The technical debt dataset. In *15th Conference on Predictive Models and Data Analytics in Software Engineering*, January 2019.

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

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

[17] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandè. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12, Feb 2019. doi:10.1109/SANER.2019.8667970.

[18] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo 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, pages 209–219, Piscataway, NJ, USA, 2019. IEEE Press. doi:10.1109/ICPC.2019.00040.

[19] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. Spongebugs: Automatically generating fix suggestions in response to static code analysis warnings. *J. Syst. Softw.*, 168:110671, 2020. doi:10.1016/j.jss.2020.110671.

[20] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. Automatically generating fix suggestions in response to static code analysis warnings. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 34–44, 2019. doi:10.1109/SCAM.2019.00013.

[21] Martin Odermatt, Diego Marcilio, and Carlo A. Furia. Static Analysis Warnings and Automatic Fixing: A Replication for C# Projects : Dataset. doi:10.5281/zenodo.5838454.

[22] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan 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)*, pages 345–355. IEEE, 2017.

[23] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 404–415, May 2017. doi:10.1109/ICSE.2017.44.

[24] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. House of cards: Code smells in open-source C# repositories. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 424–429. IEEE, 2017.

[25] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C Gall. Continuous code quality: are we (really) doing that? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 790–795, 2018.

[26] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49, 2018. doi:10.1109/SANER.2018.8330195.

[27] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of ci build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193, 2017. doi:10.1109/ICSME.2017.67.

[28] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di 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)*, pages 334–344, 2017. doi:10.1109/MSR.2017.2.

# URL References

1. https://spectrum.ieee.org/top-programming-languages/
2. https://github.com/marodev/usi-2021-automated-fixing-of-static-code-analysis-warning tree/main/resharper
3. https://rules.sonarsource.com/csharp/RSPEC-1186
4. https://www.jetbrains.com/help/resharper/Reference__Code_Inspections_CSHARP.html
5. https://www.jetbrains.com/help/resharper/Code_Analysis__Code_Inspections.html#categories
6. https://docs.sonarqube.org/latest/user-guide/issues/
7. https://github.com/dotnet/roslyn
8. https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-5.0
9. https://codeql.github.com/
10. https://dependabot.com/
11. https://github.com/StyleCop/StyleCop
12. https://github.blog/2020-09-30-code-scanning-is-now-available/
13. https://dependabot.com/blog/hello-github/

14. https://github.com/StyleCop/StyleCop
15. https://www.jetbrains.com/help/resharper/StyleCop_Styles.html
16. https://secdevtools.azurewebsites.net/helpcredscan.html
17. http://quickfixn.org/
18. https://github.com/connamara/quickfixn/blob/63f7e97f3f0353559becfcc2fc989246a4bc09c5/QuickFIXn/Fields/Fields.cs
19. https://github.com/aliyun/aliyun-openapi-net-sdk
20. https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkit
21. https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkit/blob/690f999772690b0e7e895eb939672bf34178a32d/.github/CONTRIBUTING.md#coding-standards
22. https://rules.sonarsource.com/csharp/type/Code%20Smell/RSPEC-2971
23. https://rules.sonarsource.com/csharp/type/Code%20Smell/RSPEC-1144
24. https://www.jetbrains.com/help/resharper/UnusedMember.Local.html
25. https://www.jetbrains.com/help/resharper/UnusedMember.Global.html
26. https://www.jetbrains.com/help/resharper/Reference__Code_Annotation_Attributes.html
27. https://github.com/EDCD/EDDI
28. https://rules.sonarsource.com/java/RSPEC-6212
29. https://github.com/SonarSource/sonar-dotnet/issues/865
30. https://github.com/apache/lucenenet/pull/488
31. http://quickfixn.org/
32. https://github.com/nhapinet/nhapi
33. https://github.com/ServUO/ServUO