



Are automated static analysis tools worth it? An investigation into relative warning density and external software quality on the example of Apache open source projects

Alexander Trautsch¹ · Steffen Herbold¹ · Jens Grabowski²

Accepted: 3 February 2023 / Published online: 17 April 2023
© The Author(s) 2023

Abstract

Automated Static Analysis Tools (ASATs) are part of software development best practices. ASATs are able to warn developers about potential problems in the code. On the one hand, ASATs are based on best practices so there should be a noticeable effect on software quality. On the other hand, ASATs suffer from false positive warnings, which developers have to inspect and then ignore or mark as invalid. In this article, we ask whether ASATs have a measurable impact on external software quality, using the example of PMD for Java. We investigate the relationship between ASAT warnings emitted by PMD on defects per change and per file. Our case study includes data for the history of each file as well as the differences between changed files and the project in which they are contained. We investigate whether files that induce a defect have more static analysis warnings than the rest of the project. Moreover, we investigate the impact of two different sets of ASAT rules. We find that, bug inducing files contain less static analysis warnings than other files of the project at that point in time. However, this can be explained by the overall decreasing warning density. When compared with all other changes, we find a statistically significant difference in one metric for all rules and two metrics for a subset of rules. However, the effect size is negligible in all cases, showing that the actual difference in warning density between bug inducing changes and other changes is small at best.

Keywords Static code analysis · Quality evolution · Software metrics · Software quality

Communicated by: Gabriele Bavota

Alexander Trautsch
alexander.trautsch@uni-passau.de

Steffen Herbold
steffen.herbold@uni-passau.de

Jens Grabowski
grabowski@cs.uni-goettingen.de

¹ University of Passau, Passau, Germany

² Institute of Computer Science, University of Göttingen, Göttingen, Germany

1 Introduction

Automated Static Analysis Tools (ASATs) or linters are programs that perform rule matching of source code via different representations, e.g., Abstract Syntax Trees (ASTs), call graphs or bytecode to find potential problems. Rules are predefined by the ASAT and based on common coding mistakes and best practices. If a rule is matched, a warning is generated for the developer who can then inspect the given file, line number and rule. Common coding best practices involve ASATs use in different contexts (Vassallo et al. 2020), e.g., as part of Continuous Integration (CI), within IDEs, or to support code reviews. Developers also think of these tools as quality improving when used correctly (Christakis and Bird 2016; Vassallo et al. 2020; Devanbu et al. 2016; Querel and Rigby 2021). However, due to their rule matching nature, ASATs are prone to false positives, i.e., warnings about code that is not problematic (Johnson et al. 2013). This hinders the adoption of these tools and their usefulness, as developers have to inspect every warning that is generated, whether it is a false positive or not. As a result, research into classifying ASAT warnings into true and false positives or actionable warnings is conducted, e.g., Heckman and Williams (2009), Kim and Ernst (2007), and Koc et al. (2017). Due to these two aspects, ASATs are perceived as quality improving while at the same time require manual oversight and corrections.

Due to this manual effort, we want to have a closer look on the impact of ASATs on measurable quality. Previous research regarding the impact on quality measures either builds predictive models, e.g., Nagappan and Ball (2005), Plosch et al. (2008), Rahman et al. (2014), Querel and Rigby (2018), Lenarduzzi et al. (2020), Trautsch et al. (2020), and Querel and Rigby (2021) or investigates bug fixing commits, e.g., Vetro et al. (2011) and Thung et al. (2012) or Habib and Pradel (2018).

In contrast to the previous work, we combine multiple factors in this study. First, we include manually validated data from a large-scale validation study (Herbold et al. 2022a). This allows us to use only manually validated bug fixing lines to evaluate the impact on quality instead of keyword approaches. Second, we focus on a general purpose ASAT, which allows us to investigate a broad range of static analysis warnings, e.g., readability and maintainability warnings. Third, we investigate a long term perspective by including the ASAT warning density of a file over its history. This combination allows us not only to study whether current static analysis warnings in a file has an impact, but also longer term effects of a general purpose ASAT by including its history.

In our previous work, we found that static analysis warnings are evolving over time and that we cannot just use the density or the sum of warnings (Trautsch et al. 2020a). Therefore, we use an approach that is able to produce a current snapshot view of the files we are interested in by measuring the file that induces a bug and, at the same time, all other files of the project. This ensures that we are able to produce time and project independent measurements. The drawback of this approach is that it requires a large computational effort, as we have to run the ASAT under study on every file in every revision of all study subjects. However, the resulting empirical data yields insights for researchers and practitioners.

The research question that we answer in our case study is:

- Do bug inducing files contain more static analysis warnings than other files?

Answering this question can help us to determine whether an ASAT has an impact on software quality. If bug inducing changes contain more static analysis warnings we can use an ASAT to flag high risk changes which could decrease bug inducing changes in the project.

We apply a modified fine-grained just-in-time defect prediction data collection method to extract software evolution data, including bug inducing file changes and static analysis warnings from a general purpose ASAT. We chose PMD¹ as the general purpose ASAT as it has been available for a long time and provides a good mix of available rules. Using this data and a warning density based metric calculation, we investigate the differences between bug inducing files and the rest of the studied system at the point in time when the bug is introduced. In summary, this article contains the following contributions.

- A unique and simple approach to measure impact of ASATs that is independent of differences between projects, size and time.
- Complete static analysis data for PMD for 23 open source projects for every file in every commit.
- An investigation into relative warning density differences within bug inducing changes.

The main findings of our case study are:

- Bug inducing files do not contain higher warning density than the rest of the project at the time when the bug is introduced.
- When comparing bug inducing warning density with all other changes we can measure higher warning density on a subset of PMD warnings that is a popular default for two metrics and for all available rules for one metric.

The rest of this article is structured as follows. Section 2 lists previous research related to this article and discusses the differences. Section 3 describes the case study setup, methodology, analysis procedure and the results. Section 4 discusses the results of our case study and relates them to the literature. Section 5 lists and discusses threats to validity we identified for our study. Section 6 concludes the article with a short summary. Section 7 provides a short outlook.

2 Related Work

In this article, we explore a more general view of ASATs and the warning density differences of bug inducing changes. This can be seen as a mix of a direct and indirect impact study. Therefore, we describe related work for both direct and indirect impact studies within this section.

The direct impact is often evaluated by exploring if bugs that are detected in a project are fixed by fixing ASAT warnings, i.e., did the warning really indicate a bug that needed to be fixed later.

Thung et al. (2012) investigate bug fixes of three open source projects and three ASATs: PMD, JLint, and FindBugs. The authors look at how many defects are found fully and partially by changed lines and how many are missed by the ASATs. Lines that are changed as part of a bug fix are compared with lines reported by the ASAT. Moreover, the authors describe the challenges of this approach: not every line that is changed is really a fix for the bug, therefore the authors perform manual investigation on a per-line level to identify the lines. They were able to find all lines responsible for 200 of 439 bugs. In addition, the authors find that PMD and FindBugs perform best, however their warnings are often very generic.

¹<https://pmd.github.io/>

Habib and Pradel (2018) perform an investigation of the capabilities to find real world bugs via ASATs. The authors used the Defects4J dataset by Just et al. (2014) with an extension² to investigate the number of bugs found by three static analysis tools, SpotBugs, Infer and error-prone. The authors show that 27 of 594 bugs are found by at least one of the ASATs.

In contrast to Thung et al. (2012) and Habib and Pradel (2018), we only perform an investigation of PMD. However, due to our usage of SmartSHARK (Trautsch et al. 2017), we are able to investigate 1,723 bugs for which at least three researchers achieved consensus on the lines responsible for the bug. Moreover, as PMD includes many rules related to readability and maintainability, we build on the assumption that while they are not directly indicating a bug, resolving these warnings improves the quality of the code and may prevent future bugs. This extends previous work by taking possible long term effects of ASAT warnings into account.

Indirect impact is explored by using ASAT warnings as features for predictive models and providing a correlation measure of ASAT warnings to bugs.

Nagappan and Ball (2005) explore the ability of ASAT warnings to predict defect density in modules. The authors found in a case study with Microsoft, that static analysis warnings can be used to predict defect density, therefore they can be used to focus quality assurance efforts on modules that show a high number of static analysis warnings. In contrast to Nagappan and Ball (2005), we are exploring open source projects. Moreover, we explore warning density differences between files and the project they are contained in.

Rahman et al. (2014) compare static analysis and statistical defect prediction. They find that FindBugs is able to outperform statistical defect prediction, while PMD does not. Within our study, we focus on PMD as a general purpose ASAT. Instead of a comparison with statistical defect prediction we explore, whether we can measure a difference of ASAT warnings between bug inducing changes and other changes.

Plosch et al. (2008) explores a correlation between ASAT warnings as features for a predictive model and the dependent variable, i.e., bugs. They found that static analysis warnings may improve the performance of predictive models and that they are correlated with bugs. In contrast to Plosch et al. (2008), we are not building a predictive model. We are exploring whether we can find an effect of static analysis tools without a predictive model with multiple features, instead we strive to keep the approach as simple as possible.

Querel and Rigby (2018) improve the just-in-time defect prediction based commit guru (Rosen et al. 2015) by adding ASAT warnings to the predictive model. The authors show, that just-in-time defect prediction can be improved by adding static analysis warnings. This means that there should be a connection between external quality in the form of bugs and static analysis warnings. In a follow-up study (Querel and Rigby 2021) the authors found that while there is an effect of ASAT warnings the effect is likely small. In our study, we explore a different view on the data. We explore warning density differences between bug inducing files and the rest of the project.

Lenarduzzi et al. (2020) investigate SonarQube as an ASAT and whether the reported warnings can be used as features to detect reported bugs. The authors are combining direct with indirect impact, but are more focused on predictive model performance measures. In contrast to Lenarduzzi et al. (2020), we are mainly interested in the differences in warning density between bug inducing files and the rest of the project. We are also investigating an influence, but in contrast to Lenarduzzi et al., we are comparing our results for bug fixing

²<https://github.com/rjust/defects4j/pull/112>

changes to all other changes to determine whether what we see is really part of the bug fixing change and not a general trend of all changes.

3 Case Study

The goal of the case study is to find evidence if usage of ASATs have a positive impact on the external software quality of our case study subjects. In this section, we explain the approach and ASAT choice. Moreover, we explain our study subject selection and describe the methodology and analysis procedure. At the end of this section, we present the results.

3.1 Static Analysis

Static analysis is a programming best practice. ASATs scan source code or byte code and match against a predefined set of rules. When a rule matches, the tool creates a warning for the part of the code that matches the rule.

There are different tools for performing static analysis of source code. For Java these would be, e.g., Checkstyle, FindBugs/SpotBugs, PMD, or SonarQube. In this article, we focus on Java as a programming language because it is widely used in different domains and has been in use for a long time. The static analysis tool we use is PMD. There are multiple reasons for this. PMD does not require the code to be compiled first as, e.g., FindBugs does. This is an advantage, especially with older code that might not compile anymore due to missing dependencies (Tufano et al. 2017). PMD supports a wide range of warnings of different categories, e.g., naming and brace rules as well as common coding mistakes. This is an advantage over, e.g., Checkstyle which mostly deals with coding style related rules. This enables PMD to give a better overview of the quality of a given file instead of giving only probable bugs within it. The relation to software quality that we expect of PMD stems directly from its rules. The rules are designed to make the code more readable, less error prone and overall more maintainable.

3.2 Just-in-Time Defect Prediction

The idea behind just-in-time defect prediction is to assess the risk of a change to an existing software project (Kamei et al. 2013). Previous changes are extracted from the version control system of the project and, as they are in the past, it is known whether the change induced a bug. This can be observed by subsequent removal or alteration of the change as part of a bug fixing commit. If the change was indeed removed or altered as part of a bug fixing operation it is traced back to its previous file and change and labeled as bug inducing, i.e., it introduced a bug that needed to be fixed later. In addition to these labels, certain characteristics of the change are extracted as features, e.g., lines added or the experience of the author to later train a model to predict the labels correctly for the commits. The result of the model is then a label or probability whether the change introduces a bug, i.e., the risk of the change.

However, ASATs are working on a file basis and we also want to investigate longer-term effects of ASATs. This means we need to track a file over its evolution in a software project. To achieve this, we are building on previous work by Pascarella et al. (2019) which introduced fine-grained just-in-time defect prediction. In a previous study, we improved the concept by including better labels and static analysis warnings as well as static code metrics as features (Trautsch et al. 2020). Similar to Pascarella et al. (2019), we are building upon

Table 1 Study subjects in our case study

Project	#commits	#file changes	#issues	Time frame
ant-ivy	1,647	7,860	296	2005-2017
commons-bcel	850	9,604	27	2001-2017
commons-beanutils	561	2,648	28	2001-2017
commons-codec	810	2,062	21	2003-2017
commons-collections	1,687	11,296	32	2001-2017
commons-compress	1,401	3,566	87	2003-2017
commons-configuration	1,659	4,177	97	2003-2017
commons-dbcp	729	2,211	39	2001-2017
commons-digester	1,131	3,750	11	2001-2017
commons-io	985	2,781	51	2002-2017
commons-jcs	774	7,775	37	2002-2017
commons-lang	3,028	6,312	109	2002-2017
commons-math	4,135	21,440	190	2003-2017
commons-net	1,076	4,666	96	2002-2017
commons-sxml	469	1,774	39	2005-2017
commons-validator	557	1,324	37	2002-2017
commons-vfs	1,098	7,209	67	2002-2017
giraph	819	7,715	109	2010-2017
gora	464	2,256	38	2010-2017
opennlp	1,166	9,679	82	2010-2017
parquet-mr	1,053	5,957	69	2012-2017
santuario-java	1,177	8,503	41	2001-2017
wss4j	1,711	12,218	120	2004-2017
Sum	28,987	146,783	1,723	

PyDriller (Spadini et al. 2018). In this article, we build upon our previous work and include not only counts of static analysis warnings but relations between the files, e.g., how different is the number of static analysis warnings in one file from the rest of the project. We also include aggregations of warnings with and without a decay over time.

3.3 Study Subjects

Our study subjects consist of 23 Java projects under the umbrella of the Apache Software Foundation³ previously collected by Herbold et al. (2022b). Table 1 contains the list of our study subjects. We only use projects which contain fully validated bug fixing on a line-by-line level collected in a crowd sourcing study (Herbold et al. 2022a). Every line in our data was labeled by four researchers. We only consider bug fixing lines for which at least three researchers agree that it fixes the considered bug. This naturally restricts the number of available projects, but improves the noise to signal ratio of the data.

We now give a short overview what the potential problems are and how we mitigate them. When we look at external quality, we want to extract data about defects. However, there are

³<https://www.apache.org>

several additional restrictions we want to apply. First, we want to extract defects from the Issue Tracking System (ITS) of the project and link them to commits in the Version Control System (VCS) to determine bug fixing changes. Several data validity considerations need to be taken into account here. The ITS usually has a kind of label or type to distinguish bugs from other issues, e.g., feature requests. However, research shows that this label is often incorrect, e.g., Antoniol et al. (2008), Herzig et al. (2013), and Herbold et al. (2022b). Moreover, with this kind of software evolution research, we are interested in bugs existing in the software and not bugs which occur because of external factors, e.g., new environments or dependency upgrades. Therefore, we are only considering intrinsic bugs (Rodríguez-Pérez et al. 2020).

The next step is the linking between the issue from the ITS and the commit from the VCS. This is achieved via a mention of the issue in the commit message, e.g., fixes JIRA-123. While this seems straightforward, there are certain cases where this can be problematic. The simplest one being that there is a typo in the project key, e.g., JRIA-123.

Moreover, not all changes within bug fixing commits contribute to bug fixes. Unrelated changes can be tangled with the bug fix. The restriction of all data to only changes that directly contribute to the bug fix further reduces noise in the data. We are only interested in the lines of the changes that contribute to the bug fix. This is probably the hardest to manually validate.

This was achieved in a prior publication (Herbold et al. 2022b) which served as the base for the publication which data we use in this article (Herbold et al. 2022a). In Herbold et al. (2022a) a detailed untangling is performed by four different persons for each change that fixes a bug that meets our criteria. The untangling allows for focusing on the changes that are relevant to the bug fix only without including other changes, e.g., refactoring or documentation changes. Each bug fixing change is displayed in a code diff view with syntax highlighting where each participant of the study assigns a label for each line of the change. If at least three participants agree that the line contributed to the bug fix it is considered as part of the bug fix.

3.4 Replication Kit

We provide all data and scripts as part of a replication kit.⁴

3.5 Methodology

To answer our research question, we extract information about the history of our study subjects including bugs and the evolution of static analysis warnings. While the bulk of the data is based on Herbold et al. (2022a) we include several additions necessary for answering our research question.

To maximize the relevant information within our data, we include as much information from the project source code repository as possible. After extracting the bug inducing changes, we build a commit graph of all commits of the project and then find the current main branch, usually master. After that, we find all orphan commits, i.e., all commits without parents. Then we discard all orphans that do not have a path to the last commit on the main branch, this discards separate paths in the graph, e.g., gh-pages⁵ for documentation.

⁴https://github.com/atrautsch/emse2021a_replication

⁵<https://docs.github.com/en/pages>

As we also want to capture data on release branches which are never merged back into the main branch, we add all other branches that have a path to one of our left over orphan commits. The end result is a connected graph which we traverse via a modified breadth first search. We take the date of the commit into account while we traverse the graph.

The traversal is an improved version of previous work (Trautsch et al. 2020). In addition to the previously described noise reduction via manual labeling, we additionally restrict all files to production code. One of the results of Herbold et al. (2022a) is that non-production code is often tangled with bug fixing changes. Therefore, we only add files that are production files to our final data analogous to Trautsch et al. (2020a). This also helps us to provide a clearer picture of warning density based features as production code may have a different evolution of warning density than, e.g., test or example code.

In our previous study (Trautsch et al. 2020a) we found that static analysis warnings are correlated to Logical Lines of Code (LLOC). This is not surprising as we are observing large portions of our study subjects code history. Large files that are added and removed have an impact on the number of static analysis warnings. While we do not want to discard this information, we also want to avoid the problem of large changes overshadowing information in our data. Therefore, like in our previous study, we are using warning density as a base metric in this study analogous to prior studies, e.g., Aloraini et al. (2019) and Penta et al. (2009).

Warning density (wd) is the ratio of the number of warnings and the size of the analyzed part of the code.

$$wd = \frac{\text{Number of static analysis warnings}}{\text{Product size}} \quad (1)$$

Product size is measured in LLOC. If we measure the warning density of a system $wd(s)$, we sum warnings and LLOC for each file. If we measure the warning density of a file $wd(f)$, we restrict the number of warnings and the LLOC to that file.

While this measure provides a size independent metric, we also need to take differences between projects into account. Warning density can be different between projects and for different points in time for each project. The median warning density range in our study subjects is between 0.4 and 0.8. In absolute number of warnings the difference is between 2,604 and 26,854 warnings. The median difference in warning density in the first commit is 0.7 and the last commit is 0.5. In absolute number of warnings, the difference is between 923 and 8,963 warnings. To be able to use all available data we account for these differences by using differences in warning density between the files of interest and the rest of the project under study (the system) at the specific point in time.

We calculate the warning density difference between the file and the system $fd(f_t)$.

$$fd(f_t) = wd(f_t) - wd(s_t) \quad (2)$$

If the file f at time t contains less static analysis warnings per LLOC than the system s at time t the value is negative and if it contains more it is positive. We can use this metric to investigate bug inducing commits and determine whether the files responsible for bugs contain less or more static analysis warnings per LLOC than the system they belong to.

While this yields information corrected for size, project differences, and time of the change, we also want to incorporate the history of each file. Therefore, we also sum this difference in warning density for all changes to the file. We assume that recent changes are more important than old changes, especially considering that the file history can reach back multiple years. Therefore, we introduce a decay in our warning density derived features.

$$dfd(f_t) = \sum_{j=1}^{j=t} \frac{wd(f_j) - wd(s_j)}{t - j + 1} \quad (3)$$

For the decayed file system warning density delta $dfd(f_t)$ we compute the decayed, cumulative sum of the difference between the warning density of the file ($wd(f_t)$) and the warning density of the system ($wd(s_t)$). The rationale is that if a file is constantly better, regarding static analysis warnings, than the mean of the rest of the system, this should have a positive effect. While other decay mechanisms, e.g., quadratic or logarithmic would be possible we decided on a simple linear mechanism due to the absence of data regarding what would be an ideal decay mechanism in this case. As the static analysis rules are diverse, this can be improved readability, maintainability or robustness due to additional null checks. Within our study, we explore if this effect has a measurable effect on bugginess, i.e., the lower this value is the less often the file should be part of bug inducing commits.

Instead of using all warnings for warning density, we can also restrict these warnings to a smaller set to see if this has an effect. While we do not want to choose multiple subsets to avoid false positive findings, we have to investigate whether our approach to use all available warnings just waters down the ability to indicate files which may contain bugs. To this end, we also investigate the warning density, consisting only of PMD warnings that are enabled by default by the maven-pmd plugin⁶ which we denote as *default* rules. This restricts the number of warnings that are the basis of the warning density calculation to a subset of 49 warnings that are generally considered relevant in comparison to the total number of 3.14 warnings. Their use as default warnings serves to restrict this subset to generally accepted important warnings.

To answer our research question, we compare the warning density for each bug inducing file against the project at the time before and after the bug inducing change. If the difference is positive, this means that the file had a higher warning density than the rest of the project and negative vice versa. We plot the difference in warning density in a box plot for all bug inducing files to provide an overview over all our data.

As this is influenced by a continuously improving warning density, we also measure the differences between bug inducing file changes and all other file changes. We first perform a normality test and find that the data is not normal in all cases. Thus, we apply a Mann-Whitney U test (Mann and Whitney 1947) with H_0 that there is no difference between both populations and H_1 that bug inducing files have a different warning density. We set a significance level of 0.05. Additionally, we perform a Bonferroni (Abdi 2007) correction for 8 normality tests as prerequisite for all populations for the 4 Mann-Whitney U tests. Therefore, we reject H_0 at $p < 0.0042$. If the difference is statistically significant, we calculate the effect size with Cliff's δ (Cliff 1993).

3.6 Results

We now present the results of our study and the answer to our research question whether bug inducing files contain more static analysis warnings than other files. For this, we divide the results into three parts. First, we look at the warning density via $fd(f)$ at the time before and after a bug is induced and $dfd(f)$ after a bug is induced.⁷ Second, we look at the differences between our study subjects and the prior number of changes for bug inducing file

⁶<https://maven.apache.org/plugins/maven-pmd-plugin/>

⁷before is already part of the formula

changes. Third, we compare bug inducing file changes with all other changes and determine if they are different.

3.6.1 Differences of Warning Density before and after the Bug Inducing Change

Figure 1 shows the difference in warning density between each bug inducing file and the rest of the system at the point in time before inducing the bug and after.

Surprisingly, we see a negative warning density median difference for $fd(f)$. This means that the warning density of the files in which bugs are induced is lower than the rest of the project. The drop in warning density shows that the code before the change had less warning density than after the bug inducing change. This means that code that on average contains more static analysis warnings was introduced as part of the bug inducing change.

Now, we are also interested in whether the history of preceding differences in warning density makes a difference. Instead of using the warning density difference at the point in time of the bug inducing change we use a decayed sum of the warning density differences leading up to the considered bug inducing change.

Figure 1 shows a negative median for $dfd(f)$ as well. The accumulated warning density differences between the file and the rest of the project are therefore also negative. Figure 2 shows $fd(f)$ and $dfd(f)$ for bug inducing changes restricted to default rules. We can see, that the warning density for default only is much lower due to the lower number of warnings that are considered. We can also see, that the same negative median is visible when we restrict the set of ASAT rules to default. Overall, bug inducing changes have lower warning density than the other files of the project at the time the bug was induced. However, as we will see later, this is an effect of overall decreasing warning density of our study subjects.

3.6.2 Differences Between Projects and Number of Changes

Instead of looking at all files combined, we can also look at each project on its own. We provide this data in Fig. 3. However, we note that the number of bug inducing files is low in some projects. Such projects may be influenced by few changes with extreme values. Hence, the results of single projects should be interpreted with caution. Instead, we consider trends visible in the data. While we can combine all our data due to our chosen method of metric calculation we still want to provide an overview of the per-project values. This is shown in Fig. 3 for $dfd(f)$. Figure 3 also demonstrates the difference between projects. For example, the median $dfd(f)$ for comons-codec is positive, i.e., files which induce bugs

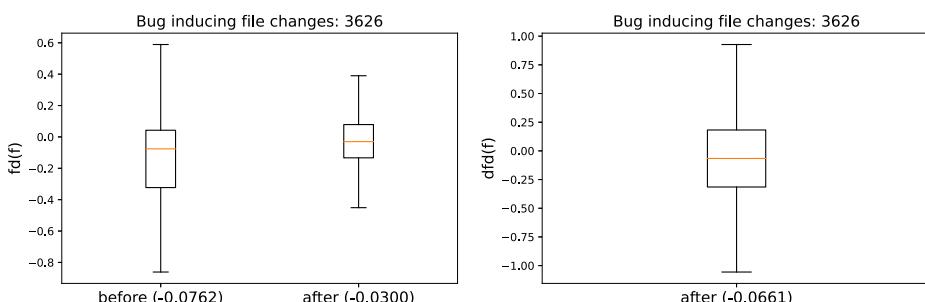


Fig. 1 Box plot of $fd(f)$ for all bug inducing files before and after the bug inducing change and $dfd(f)$ for all bug inducing files after the bug inducing change, median value in parentheses. Fliers are omitted

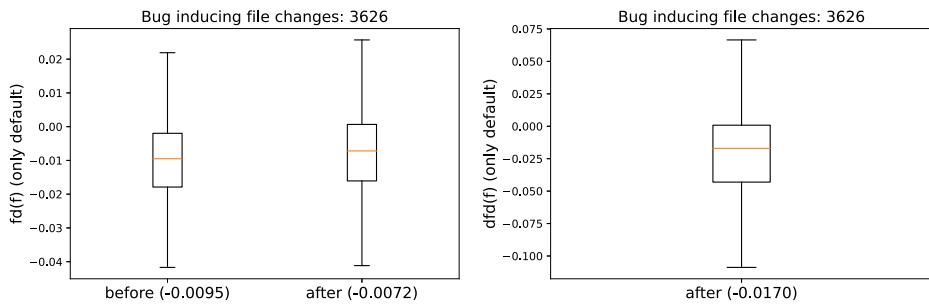


Fig. 2 Box plot of $fd(f)$ for only default warnings of all bug inducing files before and after the bug inducing change, median value in parentheses. Fliers are omitted

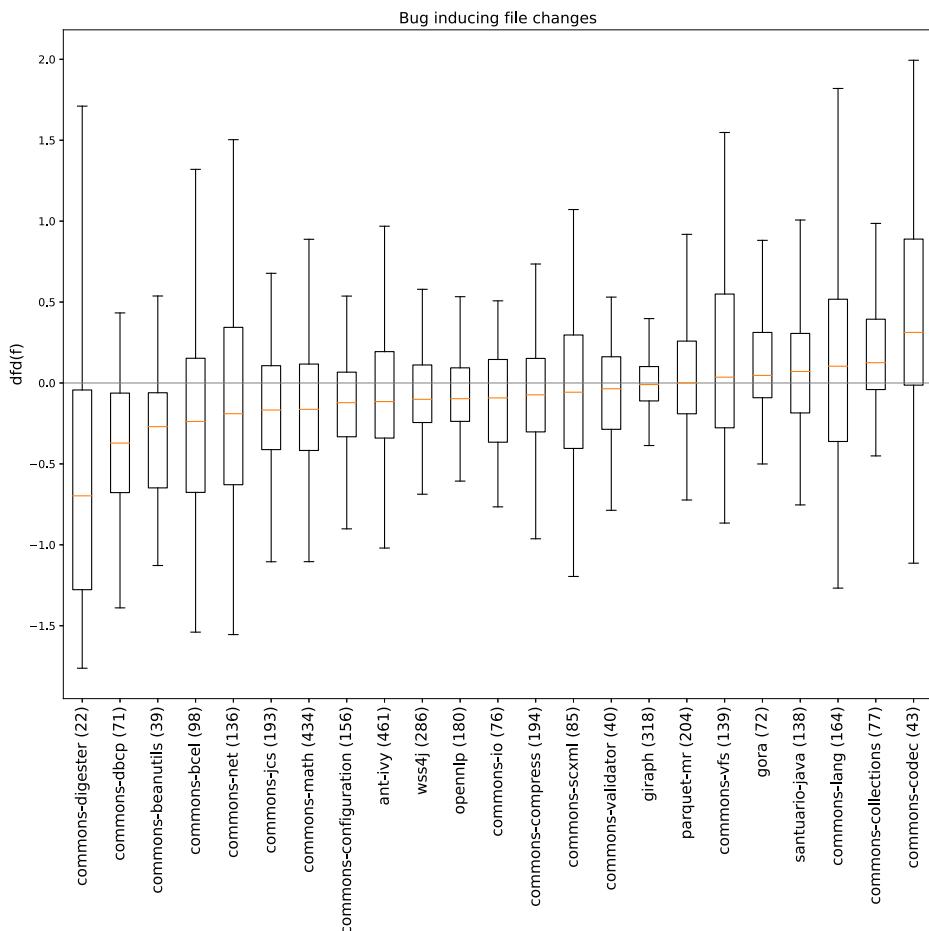


Fig. 3 Box plots of $dfdf(f)$ separately for all study subjects. The number of bug inducing file changes are in parentheses, median value in parentheses. Fliers are omitted

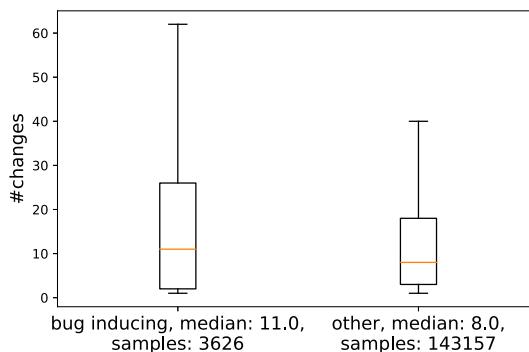


Fig. 4 Number of changes for bug inducing files and other files. Fliers are omitted

contain more warnings. The opposite is the case for, e.g., commons-digester, where the median is negative.

Overall, Fig. 3 shows that the median $dfd(f)$ is negative for 16 of 23 projects. This means that bug inducing changes have less warning density than the rest of the project for most study subjects. A possible explanation for this could be that files which have a lower warning density are changed more often, and those are the same that could be inducing bugs. If we look at the number of changes a file has in Fig. 4, we can see that bug inducing files have a bit more changes. However, the sample sizes for both are vastly different.

3.6.3 Warnings in Bug Inducing Changes

In this section, we present the top 10 absolute numbers of warning types. This offers a perspective on the distribution of warning types in changes in addition to the warning density. This is calculated by summing the delta for each warning type in a bug inducing change over all bug inducing changes. If the value is positive, it means this warning type was added more than removed in bug inducing changes, vice versa if the value is negative the warning

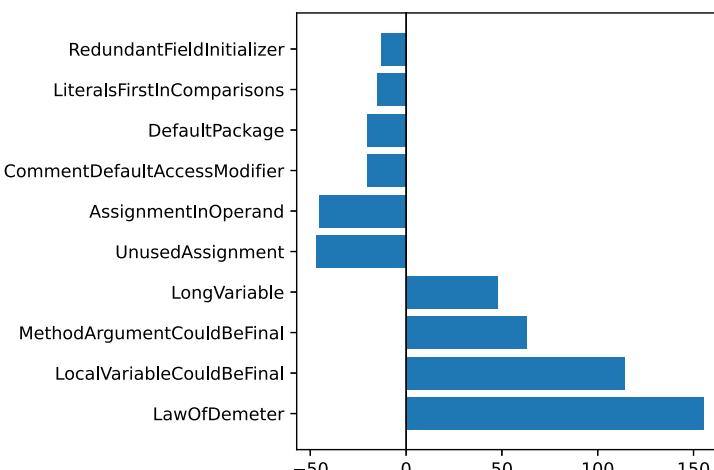


Fig. 5 Top 10 warning types for the number of warning changes in bug inducing changes for all warnings

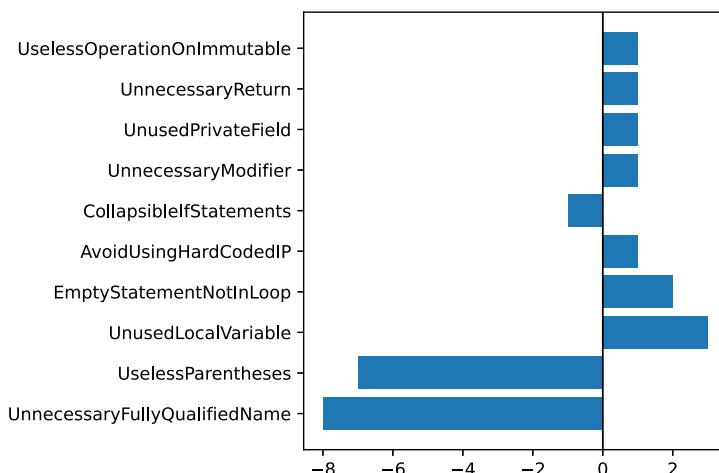


Fig. 6 Top 10 warning types for the number of warning changes in bug inducing changes for default warnings

type was more often removed in bug inducing changes. We present the top 10 absolute numbers of warning types due to the large number of warnings, the whole set is contained in the data of the replication kit. Figure 5 shows the top 10 warning types regarding the change of warnings in bug inducing changes for all possible warnings. We can see that some warning types were removed in bug inducing changes, e.g., RedundantFieldInitializer which warns when a field is initialized with the default value, which is used to initialize it by default. UnusedAssignment is removed most often in bug inducing changes. It warns about variables which are assigned a value which is overwritten later in all cases.

The warning type which is added the most is the LawOfDemeter which aims to reduce the coupling between classes. Second most added warning type is the LocalVariableCouldBeFinal warning which warns when a local variable could be declared final.

Figure 6 shows the top 10 warning types regarding the change of warnings in bug inducing changes for only the default set of warnings. For default warnings, we only see three warning types which are removed. CollapsibleIfStatements which warns about nested if statements that could be combined, UselessParentheses warns about parentheses which are not syntactically required, and finally UnnecessaryFullyQualifiedName which warns about fully qualified names which are used even though an import makes it unnecessary.

The most added warning type from the default warnings is UnusedLocalVariable which warns about a variable which is declared but not used. The second most added warning type is EmptyStatementNotInLoop. It warns about empty statements or single semicolons which are not the sole body of a loop.

A full description for all warning types and examples can be found in the PMD documentation for Java.⁸

3.6.4 Comparison with all other Changes

We now take a look at how warning density metrics differ in bug inducing changes from all other changes. We notice that the median is below zero in all cases. This is due to the

⁸https://pmd.github.io/latest/pmd_rules_java.html

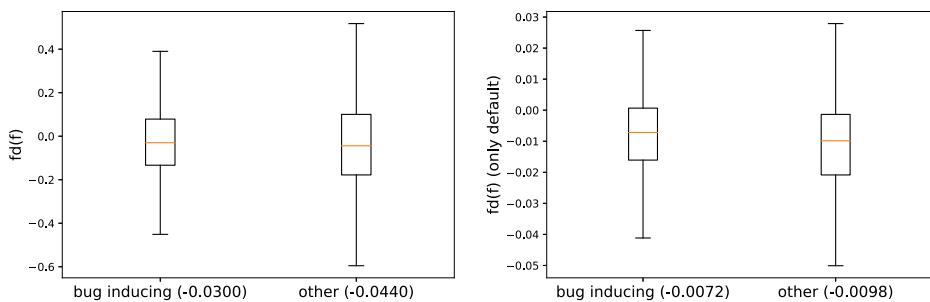


Fig. 7 Box plots of $fd(f)$ for the bug inducing change for all and default only rules for bug inducing and other file changes, median value in parentheses. Fliers are omitted

effect that warning density usually decreases over time (Trautsch et al. 2020a). Therefore, we provide a comparison of bug inducing changes with all other changes.

Figure 7 shows $fd(f)$ for bug inducing and other changes for both all rules and only the default rules. We can see that bug inducing changes have a slightly higher warning density than other changes. If we apply only default rules we see that bug inducing changes are also slightly higher.

Figure 8 shows the same comparison for $dfd(f)$. The difference for all rules is very small. However, the median for bug inducing changes is slightly higher. In contrast, we can see that for default rules the bug inducing changes have a slightly higher warning density than other changes. Table 2 shows the results of the statistical tests for differences between the values for Figs. 7 and 8.

We can see that for all rules $fd(f)$ there is a statistically significant difference. This shows that bug inducing file changes have a higher warning density than other changes.

Overall, we see that there is a significant difference with a negligible effect size for $fd(f)$ and $dfd(f)$ for default rules between bug inducing and other changes. The data shows that in these cases the bug inducing file changes have a higher warning density than other changes. Together with Figs. 7, 8 and Table 2 we can conclude, that bug inducing file changes contain more static analysis warnings than other file changes. Restricting the rules to the default set increases the effect size slightly. However, the effect sizes are still negligible in all cases.

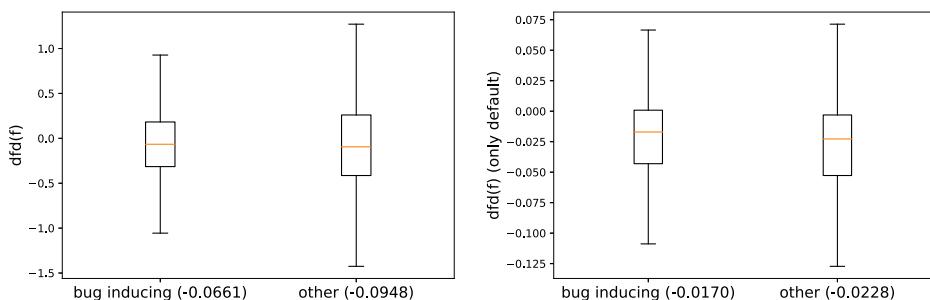


Fig. 8 Box plots of $dfdf(f)$ for all and default only rules bug inducing and other file changes, median value in parentheses. Fliers are omitted

Table 2 Median values, Mann-Whitney U test p-values (statistically significant values are bolded) and effect sizes for all warning density metrics

WD Metric	Median other	Median bug inducing	P-value	Effect size
$fd(f)$	−0.0440	−0.0300	< 0.0001	0.05 (n)
$fd(f)$ (default)	−0.0098	−0.0072	< 0.0001	0.10 (n)
$dfd(f)$	−0.0948	−0.0661	0.0247	—
$dfd(f)$ (default)	−0.0228	−0.0170	< 0.0001	0.07 (n)

3.6.5 Summary

In summary, we have the following results for our research question.

RQ Summary: Do bug inducing files contain more static analysis warnings than other files?

We find that bug inducing files contain fewer static analysis warnings than other files at bug inducing time. However, this is not because these files have a higher quality, but rather because the warning density decreases over time, i.e., most files that are changed have fewer warnings than the rest of the project.

When we compare the differences between bug inducing and other changes, we find that it depends on the applied rules. If we apply all rules we find a statistically significant difference in $fd(f)$. If we apply only default rules we find a statistically significant difference in $fd(f)$ and $dfd(f)$. This indicates that bug inducing file changes have slightly more static analysis warnings than other file changes, although the effect size in all cases is negligible.

4 Discussion

We found that the bug inducing change itself increased the warning density of the code in comparison to the rest of the project as shown in Fig. 1. This means that the actual change in warning density is as we expected, i.e., the change that induces the bug is increasing the warning density in comparison to the rest of the project. This is an indication that warning density related metrics can be of use in just-in-time defect prediction scenarios, i.e., change based scenarios, as also shown by Querel and Rigby (2021) and in our previous work (Trautsch et al. 2020). However, the effect is negligible in our data. This was also the case for predictive models by Querel and Rigby (2021). Thus, any gain in prediction models due to general static analysis warnings is likely very small.

However, when we look at the median difference between bug inducing files and the rest of the project at that point in time we see that bug inducing files contain fewer static analysis warnings. This counterintuitive result can be fully explained by the overall decreasing warning density over time, we found in our previous study (Trautsch et al. 2020a). This finding is highly relevant for researchers, because this shows the importance of accounting for time as confounding factor for the evaluation of the effectiveness of methods. Without the careful consideration of the change over time, we would now try to explain why bug

inducing files have fewer warnings and other researchers may build on this faulty conclusion. Therefore, this part of our results should also be a cautionary tale for other researchers that investigate the effectiveness of tools and methods: if the complete project is used as a baseline, it should always be considered when source code was actually worked on. If parts of the source code have been stable for a long time, they are not suitable for a comparison with recently changed code, without accounting for general changes, e.g., in coding or testing practices, over time.

However, we did find that code with more PMD warnings leads to more bugs when changed. When looking into the differences between bug inducing file changes and all other file changes we find significant differences in 3 of 4 cases. While the effect size is negligible in all cases, using only the default rules yields a higher effect size. These rules were hand-picked by the Maven developers, arguably because of their importance for the internal quality. For practitioners, this finding is of particular importance: not only does it reduce the number of alerts to carefully select ASAT warnings from a large set of candidates, it can also help to reduce general issues that are associated with bugs. While the effect size remains negligible in our findings, we note that we cannot account for possible offline use of static analysis tools. If developers use static analysis tools offline the effect might be larger.

This also has implications for researchers when including warning density based metrics into predictive models. Our data shows that the model might be improved by choosing an appropriate subset of the possible warnings of an ASAT. Using all warnings without considering their potential relation to defects is not a good strategy. Our data also shows that a good starting point might be a commonly used default, e.g., for PMD the maven-pmd-plugin default rules.

5 Threats to Validity

In this section, we discuss the threats to validity we identified for our work. To structure this section we discuss four basic types of validity separately, as suggested by Wohlin et al. (2000).

5.1 Construct Validity

A threat to the relation between theory and observation may occur in our study from the measurement of warning density. We restrict the data to production code to mitigate effects test code has on warning density as it is often much simpler than production code.

As shown in Fig. 3, the projects are not only different regarding the average value of warning density but also regarding the variance of warning density within the project. While the evolution of warning density over time may be the main contributor to the variance as we include the data from all years, there can also be a larger variety between the files of each project regarding the warning density. After the initial analysis we conducted another analysis which instead of comparing a file against the whole project, compared the file only against the files of the project in the same package. This shows a statistically significant effect for both $fd(f)$ and $dfd(f)$ for the set of default warnings, same as our initial analysis. There is a difference regarding all warnings only for $fd(f)$. Although this result does not fully mitigate this threat it could be seen as a hint that the project variability is not that large of a factor.

False positives of warnings, i.e., warnings about code that contains no problems could also be a threat to validity. Vassallo et al. (2020) found that false positives are a prime

concern for developers. Within our study, we mitigate this threat by not investigating single types of warnings for our main research question. Rather, we use the sum of warnings for two sets (all possible warnings and the Maven plugin default warnings). In addition, we use deltas of warnings rather than the absolute values for warning density. This can mitigate the threat of false positives if we assume that they are equally distributed within bug inducing and non bug inducing files. Nevertheless, a high number of false positives could have adverse effects on our results given that the effect size is negligible.

Missing data due to offline use of static analysis tools can influence the results. While we cannot account for this, we note that offline use of ASATs would only influence the results most likely towards an positive effect of static analysis tools as true positives might be directly solved by the developer while false positives might be ignored.

5.2 Internal Validity

A general threat to internal validity would be a selection of static analysis warnings. We mitigate this by measuring the warning density for all warnings and for only default warnings as a common subset for Java projects. Due to the nature of our approach, we mitigate differences between projects regarding the handling of warnings as well as the impact of size.

In order to give more weight to recent instances of files, we add a linear decay in the warning density history for $dfd(f)$. A different decay mechanism, e.g., logarithmic or quadratic could yield different results.

5.3 External Validity

Due to the usage of manually validated data in our study, our study subjects are restricted to those for which we have this kind of data. This is a threat to the generalizability of our findings, e.g., to all Java projects or to all open source projects. Still, as we argue in Herbold et al. (2022a), our data should be representative for mature Java open source projects.

Moreover, we observe only one static analysis tool (PMD). While this may also restrict the generalizability of our study, we believe that due to the large range of rules of this ASAT our results should generalize to ASATs that are broad in scope. ASATs of a different focus, e.g., on coding style (Checkstyle) or directly finding bugs (FindBugs, SpotBugs) may result in different results.

5.4 Conclusion Validity

We report lower warning density for bug inducing files in comparison to the rest of the project at that point in time. While this reflects the difference in warning density between the file and the project, it can be influenced by constantly decreasing warning density. We mitigate this by also including a comparison between bug inducing changes and all other changes.

6 Conclusion

In this article, we provide evidence for a common assumption in software engineering, i.e., that static analysis tools provide a net-benefit to software quality even though they suffer

from problems with false positives. We use an improved state-of-the art approach used for fine-grained just-in-time defect prediction to establish a link between files within commits that induce bugs and measure warning density related features which we aggregate over the evolution of our study subjects. This approach runs on data which allows us to remove several noise factors from our data, wrong issue types, wrong issue links to commits and tangled bug fixes. The analysis approach allows us to merge the available data as it mitigates differences between projects, sizes and to some extent the evolution of warnings over time.

We find that bugs are induced in files which have a comparably low warning density, i.e., less static analysis warnings than the files of the rest of the project at the time the bug was induced. However, this difference can be explained by the fact that the warning density decreases over time. When we compare the bug inducing changes with all other changes, we do find a significant higher warning density when using all PMD rules in one of two metrics. However, the effect size is negligible. When we use a small rule set that restricts the 3.14 PMD warnings to the 49 warnings hand-picked by the Maven developers as default warnings, we find that bug inducing changes have a significant but also negligible larger warning density. However, the effect size increases for the default rule set. Assuming that the smaller rule set was crafted with the intent to single out the most important rules for the quality, this indicates that there is indeed a (weak) relationship between general ASAT tools and bugs. However, we note that this relationship might be stronger due to possible offline use of static analysis tools by the developers, e.g., included in the IDE and used before committing the code.

This is also direct evidence for a common best practice in the use of static analysis tools: Appropriate rules for ASATs should be chosen for the project. This not only reduces the number of alarms, which is important for the acceptance by developers, but also has a better relationship with the external quality of the software measured through bugs.

7 Future Work

Within our study, we investigated the differences in warning density between bug inducing and other changes and found only a negligible difference. A future study could investigate the reasons for this result, e.g., whether all bugs are indicated by static analysis warnings and whether there are bug types which are not indicated by static analysis warnings. This would require a careful analysis of the bug itself and the warning of the static analysis tool if one is generated. The bug itself has to be first untangled because not all lines changed in a bug fix may be part of the bug fix itself. A good starting point would probably be manually validated data like in our previous study (Herbold et al. 2022a) or Defects4J (Just et al. 2014).

Appendix A: Results with Same Package Warning Density

This result was obtained after the initial analysis. Instead of the whole system we compare each file against only the files in the same package. If the file is the only one in the package it is not added to the result.

Table 3 Median values, Mann-Whitney U test p-values (statistically significant values are bolded) and effect sizes for all warning density metrics

WD Metric	Median other	Median bug inducing	P-value	Effect size
$fd(f)$	−0.0161	−0.0130	0.4302	—
$fd(f)(default)$	−0.0036	−0.0030	< 0.0001	0.05 (n)
$dfd(f)$	−0.0311	−0.0299	0.0182	—
$dfd(f)(default)$	−0.0095	−0.0065	< 0.0001	0.05 (n)

All metrics are calculated with the warning density of the same package of each file instead of the whole system

Funding Open Access funding enabled and organized by Projekt DEAL. This work was partly funded by the German Research Foundation (DFG) through the project DEFECTS, grant 402774445.

Data Availability The datasets generated during and/or analysed during the current study are available in the Zenodo repository, <https://doi.org/10.5281/zenodo.7254873>.

Declarations

Competing interests The authors have no competing interests to declare that are relevant to the content of this article.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Abdi H (2007) Bonferroni and Sidak corrections for multiple comparisons. In: Encyclopedia of measurement and statistics. Sage, Thousand Oaks, pp 103–107
- Aloraini B, Nagappan M, German DM, Hayashi S, Higo Y (2019) An empirical study of security warnings from static application security testing tools. J Syst Softw 158:110427. <https://doi.org/10.1016/j.jss.2019.110427>. <http://www.sciencedirect.com/science/article/pii/S0164121219302018>
- Antoniol G, Ayari K, Di PentaM, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement? a text-based approach to classify change requests. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: Meeting of minds, CASCON '08. Association for Computing Machinery, New York. <https://doi.org/10.1145/1463788.1463819>
- Christakis M, Bird C (2016) 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. ACM, New York, pp 332–343. <https://doi.org/10.1145/2970276.2970347>
- Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol Bull 114:494–509
- Devanbu P, Zimmermann T, Bird C (2016) Belief evidence in empirical software engineering. In: 2016 IEEE/ACM 38th international conference on software engineering (ICSE), pp 108–119. <https://doi.org/10.1145/2884781.2884812>
- Habib A, Pradel M (2018) 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, pp 317–328. <https://doi.org/10.1145/3238147.3238213>

- Heckman S, Williams L (2009) A model building process for identifying actionable static analysis alerts. In: 2009 international conference on software testing verification and validation, pp 161–170. <https://doi.org/10.1109/ICST.2009.45>
- Herbold S, Trautsch A, Ledel B, Aghamohammadi A, Ghaleb TA, Chahal KK, Bossenmaier T, Nagaria B, Makedonski P, Ahmadabadi MN, Szabados K, Spieker H, Madeja M, Hoy N, Lenarduzzi V, Wang S, Rodríguez-Pérez G, Colomo-Palacios R, Verdecchia R, Singh P, Qin Y, Chakroborti D, Davis W, Walunj V, Wu H, Marcilio D, Alam O, Aldaeaj A, Amit I, Turhan B, Eismann S, Wickert AK, Malavolta I, Sulir M, Fard F, Henley AZ, Kourtzanidis S, Tuzun E, Treude C, Shamasbi SM, Pashchenko I, Wyrich M, Davis J, Serebrenik A, Albrecht E, Aktas EU, Strüber D, Erbel J (2022a) A fine-grained data set and analysis of tangling in bug fixing commits. *Empir Softw Eng* 27(6):125. <https://doi.org/10.1007/s10664-021-10083-5>
- Herbold S, Trautsch A, Trautsch F, Ledel B (2022b) Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection. *Empir Softw Eng* 27(2):42. <https://doi.org/10.1007/s10664-021-10092-4>
- Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: How misclassification impacts bug prediction. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE Press, pp 392–401
- Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why do software developers use static analysis tools to find bugs? In: Proceedings of the 2013 international conference on software engineering, ICSE '13. IEEE Press, Piscataway, pp 672–681. <http://dl.acm.org/citation.cfm?id=2486788.2486877>
- Just R, Jalali D, Ernst MD (2014) 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. Association for Computing Machinery, New York, pp 437–440. <https://doi.org/10.1145/2610384.2628055>
- Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. *IEEE Trans Softw Eng* 39(6):757–773. <https://doi.org/10.1109/TSE.2012.70>
- Kim S, Ernst MD (2007) Which warnings should I fix first? In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ESEC-FSE '07. ACM, New York, pp 45–54. <https://doi.org/10.1145/1287624.1287633>
- Koc U, Saadatpanah P, Foster JS, Porter AA (2017) Learning a classifier for false positive error reports emitted by static code analysis tools. In: Proceedings of the 1st ACM SIGPLAN international workshop on machine learning and programming languages, pp 35–42. <https://doi.org/10.1145/3088525.3088675>
- Lenarduzzi V, Lomio F, Huttunen H, Taibi D (2020) Are sonarqube rules inducing bugs? 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER), pp 501–511
- Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. *Ann Math Stat* 18(1):50–60
- Nagappan N, Ball T (2005) Static analysis tools as early indicators of pre-release defect density. In: Proceedings of the 27th international conference on software engineering, ICSE '05. ACM, New York, pp 580–586. <https://doi.org/10.1145/1062455.1062558>
- Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. *J Syst Softw* 150:22–36. <https://doi.org/10.1016/j.jss.2018.12.001>, <http://www.sciencedirect.com/science/article/pii/S0164121218302656>
- Penta MD, Cerulo L, Aversano L (2009) The life and death of statically detected vulnerabilities: An empirical study. *Inf Softw Technol* 51(10):1469–1484. <https://doi.org/10.1016/j.infsof.2009.04.013>, <http://www.sciencedirect.com/science/article/pii/S0950584909000500>, source Code Analysis and Manipulation, SCAM 2008
- Plosch R, Gruber H, Hentschel A, Pomberger G, Schiffer S (2008) On the relation between external software quality and static code analysis. In: 2008 32nd annual IEEE software engineering workshop, pp 169–174. <https://doi.org/10.1109/SEW.2008.17>
- Querel L, Rigby PC (2021) Warning-introducing commits vs bug-introducing commits: A tool, statistical models, and a preliminary user study. In: 29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20–21, 2021. IEEE, pp 433–443. <https://doi.org/10.1109/ICPC52881.2021.00051>
- Querel LP, Rigby PC (2018) Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2018. Association for Computing Machinery, New York, pp 892–895. <https://doi.org/10.1145/3236024.3264599>

- Rahman F, Khatri S, Barr ET, Devanbu P (2014) Comparing static bug finders and statistical prediction. In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, New York, pp 424–434. <https://doi.org/10.1145/2568225.2568269>
- Rodríguez-Pérez G, Nagappan M, Robles G (2020) Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. IEEE Trans Softw Eng 1–1. <https://doi.org/10.1109/TSE.2020.3021380>
- Rosen C, Grawi B, Shihab E (2015) Commit guru: Analytics and risk prediction of software commits. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, ESEC/FSE 2015. Association for Computing Machinery, New York, pp 966–969. <https://doi.org/10.1145/2786805.2803183>
- Spadini D, Aniche M, Bacchelli A (2018) PyDriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering - ESEC/FSE 2018. ACM Press, New York, pp 908–911. <https://doi.org/10.1145/3236024.3264598>, <http://dl.acm.org/citation.cfm?doid=3236024.3264598>
- Thung F, Lucia Lo D, Jiang L, Rahman F, Devanbu PT (2012) To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In: 2012 Proceedings of the 27th IEEE/ACM international conference on automated software engineering, pp 50–59. <https://doi.org/10.1145/2351676.2351685>
- Trautsch A, Herbold S, Grabowski J (2020a) A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in apache open source projects. Empirical Software Engineering. <https://doi.org/10.1007/s10664-020-09880-1>
- Trautsch A, Herbold S, Grabowski J (2020) Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In: 36th international conference on software maintenance and evolution (ICSME)
- Trautsch F, Herbold S, Makedonski P, Grabowski J (2017) Addressing problems with replicability and validity of repository mining studies through a smart data platform. Empirical Software Engineering. <https://doi.org/10.1007/s10664-017-9537-x>
- Tufano M, Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD, Poshyvanyk D (2017) There and back again: Can you compile that snapshot? J Softw Evol Process 29(4). <http://dblp.uni-trier.de/db/journals/smr/smr29.html#TufanoPBPOLP17>
- Vassallo C, Panichella S, Palomba F, Proksch S, Gall HC, Zaidman A (2020) How developers engage with static analysis tools in different contexts. Empir Softw Eng 25. <https://doi.org/10.1007/s10664-019-09750-5>
- Vetro A, Morisio M, Torchiano M (2011) An empirical validation of findbugs issues related to defects. In: 15th annual conference on evaluation assessment in software engineering (EASE 2011), pp 144–153. <https://doi.org/10.1049/ic.2011.0018>
- Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.