Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube

Diego Marcilio$^{\phi}$, Rodrigo Bonifácio$^{\phi\psi}$, Eduardo Monteiro$^{\delta}$, Edna Canedo$^{\phi}$, Welder Luz$^{\phi}$ and Gustavo Pinto$^{\omega}$
$^{\phi}$Computer Science Department, University of Brasília, Brasília, Brazil
$^{\psi}$Paderborn University, Paderborn, Germany
$^{\delta}$Department of Statistics, University of Brasília, Brasília, Brazil
$^{\omega}$Faculty of Computing, Federal University of Pará, Belém, Brazil

Abstract—**The use of automatic static analysis tools (ASATs) has gained increasing attention in the last few years. Even though available research have already explored ASATs issues and how they are fixed, these studies rely on revisions of the software, instead of mining real usage of these tools and real issue reports. In this paper we contribute with a comprehensive, multi-method study about the usage of SonarQube (a popular static analysis tool), mining 421,976 issues from 246 projects in four different instance of SonarQube: two hosted in open-source communities (Eclipse and Apache) and two hosted in Brazilian government institutions (Brazilian Court of Account (TCU) and Brazilian Federal Police (PF)). We first surveyed team leaders of the analyzed projects and found that they mostly consider ASATs warning messages as relevant for overall software improvement. Second, we found that both Eclipse and TCU employ highly customized instance of SonarQube, with more than one thousand distinct checkers–though just a subset of these checkers actually led to issues' reports. Surprisingly, we found a low resolution rate per project in all organizations–on average, 13% of the issues have been solved in the systems. We conjecture that just a subset of the checkers reveal real design and coding flaws, and this might artificially increase the technical debt of the systems. Nevertheless, considering all systems, there is a central tendency (median) of fixing issues after 18.99 days they had been reported, faster than the period for fixing bugs as reported in previous studies.**

I. Introduction

The rapid growth of software development activities over the past decade has increased the focus on the reliability and quality of software systems, which also incurs in associated costs to ensure these characteristics [1]. The use of Automatic Static Analysis Tools (ASATs) is a prominent approach to improve internal quality attributes, as they reveal recurrent code violations without having the cost of running the program [2]. By mainly leveraging heuristic pattern matching approaches to scan source/binary code, these tools can be used for a variety of purposes [3], [4], [5], such as automatically identify refactoring opportunities [6], detect security vulnerabilities [7], highlight performance bottlenecks [8], and bad programming practices, such as code smells [9].

One organization can leverage the benefits of using static analysis tools when they are integrated in the development pipeline—for instance, through the adoption of Continuous Integration (CI) practices [4], [5], [10]. An important principle of CI is continuous inspection, which includes static analysis of source code, among other types of assessments, on every change of the software [10]. Even though the use of ASATs provide several benefits, developers still face challenges when using them [2], [3]. One common reason is the high number of false positive violations, which can reach the thousands as reported by Johnson et al. [2]. Another related barrier is filtering through warnings to find defects that are worth fixing, as violations are often ignored [3].

Previous works have already investigated how open-source software (OSS) projects take advantage of ASATs (e.g., [11], [12], [13]). For instance, Beller et al. [11] found that ASAT tools are widely used, with most projects relying on a single ASAT tool. Recent studies [12], [13] have focused on what kind of violations developers tend to fix. We challenge this perspective as researchers had to run the static analysis tools themselves on multiple revisions of the projects. As stated by Liu et al. [13], many developers do not use ASATs as part of their development tool chain. Consequently, a piece of code flagged as a fixed issue by these studies may never have been perceived as a violation, and thus fixed unintentionally. We argue that this fact has a significant impact on how developers react to violations. Furthermore, the studies restricted their analysis to OSS projects.

In this work we present the results of an in-depth, multi-method study that aims to increase the comprehension of how developers respond to violations reported by ASATs. To achieve this goal, we first conduct a survey with practitioners, in order to better understand the relevance of using static analysis tools and the general procedures developers take to deal with the reported issues. We found that developers consider the use of static analysis tools relevant for improving software quality. Developers also use the outcomes of these tools to decide about postponing a release or accepting / rejecting source code contributions. We then curate and mine a dataset of issues reported from both OSS and industrial projects that actually use the SonarQube ASAT, the leading product for continuous code inspection, used by more than 85,000 organizations.

Our study comprehends 373,413 non-fixed violations and 36,974 fixed violations spanning from 246 Java projects distributed in four distinct SonarQube instances, two from Eclipse (EF) and Apache foundations (ASF)—both well-known Java ecosystems [12], [14]—and two from Brazilian government institutions, the Brazilian Federal Court of Accounts (TCU) and the Brazilian Federal Police (PF). Altogether, in this work we answer questions related to (a) *the perceptions of the reported issues* and (b) *the practices for fixing them.* Accordingly, we present the following contributions:

- We present how experienced practitioners use the reports of a static analysis tool.
- We report the results of an in depth analysis of issues and fixes from four different instances of SonarQube.
- We implement and make available an approach for mining issues from SonarQube.
- We make available an extensive dataset[1] of issues from open-source Java projects.

II. Background

Static analysis tools analyze source code without having to run the program [15]. They aim to capture defects in source code in an anticipated manner, helping ensure higher quality software during its development process [2]. ASATs can identify important classes of defects that are frequently not found by neither unit tests nor manual inspection [16]. Moreover, ASATs can be integrated to the development pipeline by a variety of ways, such as on demand, just in time before the source code is stored in a source management system, or continuously during software development activities [2]. The latter can be achieved through the adoption of continuous integration (CI) practices, specifically by continuous inspection, which includes static analysis of source code [10].

Several tools integrate static analysis into development workflows, including SonarQube. SonarQube [17] is one of the most adopted code analysis tool in the context of CI environments [10], [4]. It supports more than 25 languages and is used by more than 85,000 organizations. SonarQube includes its own rules and configurations, but new rules can be added. Notably, it incorporates popular rules of other static and dynamic code analysis tools, such as FindBugs and PMD [10].

SonarQube considers rules as coding standards. When a piece of code violates a rule, an issue is raised. SonarQube classifies issues by type and severity. Issues' types are related to the code itself [12]. There are three broad kinds of issues on SonarQube. A **Bug** occurs when an issue is related to a piece of code that is demonstrably wrong. A **Vulnerability** occurs when a piece of code could be exploited to cause harm to the system. Finally, a **Code smell** occurs when an issue represent instances of improper code, which are neither a bug nor a vulnerability.

The severities of issues can also be categorized by their possible impact, either on the system or on the developer's productivity. **Blocker** and **critical** issues might impact negatively the system, with blocker issues having a higher probability compared to critical ones. SonarQube recommended to fix these kind of issues as soon as possible[2]. **Major** issues can highly impact the productivity of a developer, while **minor** ones have little impact. Finally, **info** issues represent all issues that are neither a bug nor a quality flaw. In SonarQube, issues flow through a lifecycle, taking one of multiple possible statuses, namely: **open**, which is set by SonarQube on new issues; **resolved**, set manually to indicate that an issue should be closed; **closed**, which is set automatically by SonarQube when an issue is fixed.

[1]https://doi.org/10.5281/zenodo.2602038

[2]https://docs.sonarqube.org/latest/user-guide/issues/

III. Study Settings

In this section we describe the settings of our study. We first state the goal of our investigation, and then we present details about the research questions we address and the procedures we take to conduct the study and collect issues from the SonarQube instances.

A. Research Goal

The main goal of this research is to build a broad comprehension about how developers use the static analysis SonarQube tool, as well as to characterize how they respond to the warnings reported by these tools. Differently from previous works [12], [13], here we focus on both open-source and private organizations.

B. Research Questions

We conduct a multi-method study to investigate the following research questions:

(RQ1) What are the practitioners' perceptions about the use of static analysis tools?
(RQ2) How often developers fix issues found in open-source and private SonarQube instances?
(RQ3) What are the SonarQube issues that developers fix more frequently?
(RQ4) How is the distribution of the SonarQube issues? That is, do 20% of the issues correspond to 80% of the fixes? Do 20% of the files lead to 80% of the issues?

To answer RQ1 we use a *survey* approach. We explore whether the use of ASATs is relevant to improve software quality, considering the perspective of practitioners. We also use the answers to RQ1 to support the discussion about the results of the second study.

To answer the remaining questions we use a *mining software repository* approach. The goal in this case is to comprehend the dynamics for fixing issues reported by SonarQube. The last research question might help practitioners to configure static analysis tools properly, and thus avoid a huge number of false-positives. Moreover, it might also help developers plan their activities in a more effective way, reducing the efforts to improve the internal quality of the systems.

We consider different perspectives to answer these questions, including the characteristics of the systems (e.g., legacy or greenfield systems, private or open-source systems) and the type and severity of the issues. The datasets we use in the investigation include issues from four SonarQube instances, two publicly available, and two private ones.

TABLE I
SURVEY QUESTIONS ANSWERED BY ALL 18 PARTICIPANTS

ID	Question
Q1	Do you agree that warning messages reported by ASATs are relevant for improving the design and implementation of software?
Q2	How do you fix the issues reported by Automatic Static Analysis Tools?
Q3	How often do you use program transformation tools to automatically fix issues reported by Automatic Static Analysis Tools?
Q4	How important is the use of program transformation tools to fix issues reported by ASATs?
Q5	How often do you reject pull-requests based on the issues reported by ASATs?
Q6	How often do you postpone a release based on the issues reported by ASATs?

C. Research Methods

To answer the first research question, we conduct an online survey with developers from the four organizations in which we focus our study. We asked 6 closed questions (see Table I) mainly using a Likert scale [18]. For the OSS foundations we asked for participation on mailing lists, while for the private organizations we reached our personal contacts. The survey was available for approximately one month. Participation was voluntary and all the participants allowed the researcher to use and disclose the information provided while conducting the research. The estimated time to complete the survey was 12-15 minutes. 18 developers, from 81 unique visits (completion rate of 23%), answered all questions of our questionnaire.

The majority of the participants identified themselves, although it was not mandatory. Among the respondents, 50% have more than ten years of experience in software development, 27.77% have between four and ten years, and 22.23% have under four years. Regarding the time using ASATs, 33.33% have more than four years, and the remaining 66.67% have under than four years.

To investigate the practices for fixing issues (RQ2) – (RQ4), we mine four different SonarQube repositories. We focus on projects from Eclipse Foundation (EF) and Apache Software Foundation (ASF). This decision is based on the work of Izquierdo and Cabot [14], which analyzes 89 software foundations in OSS development and both EF and ASF were the largest in terms of projects they support (216 for EF and 312 for ASF). Moreover, their projects are known for high quality and wide adoption in the OSS community [12].

Our private datasets from Brazilian government institutions are selected not only due to convenience (we got permission to mine their issue databases), but also because they represent a heterogeneous context. TCU does inhouse development whereas PF mostly outsources. More important to this work, they both enforce conformity to SonarQube quality checks in their development processes. We restricted our analysis to the Java programming language, since it is the programming language used in the majority of projects available in the selected OSS foundations [19] and is also the primary programming language used in the private datasets. In addition, SonarQube has a very mature analysis for Java projects, with more than 525 rules.

We leverage statistical techniques during the analysis of this study, including exploratory data analysis (considering plots and descriptive statistics) and hypothesis testing methods. In particular, we use the non-parametric Kruskal-Wallis hypothesis testing [20] to understand whether or not the severity of a given issue influences the interval in days of the fix. We also use the Dunn test [21] to conduct a multiple comparison of the means. We chose non-parametric methods because our data does not follow a normal distribution. As such, we used the Spearman's rank correlation test [22] to investigate the correlation between variables.

D. SonarQube Data Collection

We implement a tool[3] that is able to extract several data from SonarQube instances. The data collection is done by querying the API provided in the instance itself. One challenge hidden in this activity is to deal with distinct versions of SonarQube, as parameters and responses differ from versions with large disparities. We found that OSS projects rely on older versions of SonarQube: EF uses 4.5.7 (major version from September, 2014) and ASF uses 5.6.3 (major version from June, 2016). Interestingly, those are Long Term Support (LTS) versions. The private instances rely on newer versions (the 7.x, released after 2018). None of them is a LTS version though, although they can be queried in the same fashion. The data collection took place during the months of November / December of 2018, though we updated the datasets also in January 2019.

For each SonarQube instance, we gather data for rules, projects, and their issues. As aforementioned, rules indicate whether instances use customized rules or not. Even though SonarQube encompasses rules from other ASATs, such as FindBugs and CheckStyle, we found that EF and TCU use a significant number of customized rules from these ASATs. We filter out 28 projects to remove branches that are considered as projects in SonarQube, a situation particular in the TCU's repository. The next step collects issues: open, fixed, won't fix, and false-positives. To filter out non-desired projects, such as toy projects, inactive and demos [23], we apply a filter to consider only projects with at least one Java fixed issue. We removed 31 projects from EF, 21 from ASF, 62 from PF and 157 from TCU when applying this filter. Table II presents an overview of the whole dataset.

Overall we collected data from 246 Java software projects. Altogether, these software projects employed 4,319 rules (2,086 distinct ones). Still, these projects had reported a total

[3]https://github.com/dvmarcilio/sonar-issues-miner

TABLE II
OVERVIEW OF THE COLLECTED DATA. THE LAST COLUMN INDICATES QUANTITY OF (W)ON'T (F)IX AND (F)ALSE-(P)OSITIVE ISSUES.

Org.	Rules	Projects	Filtered Projects	Open Issues	Fixed Issues	WF + FP
EF	1,493	95	64	29,547	6,993	952
ASF	397	48	27	136,235	16,731	10,168
TCU	1,998	283	98	165,304	10,021	467
PF	530	119	57	42,327	3,229	2
Total	2,086[a]	545	246	373,413	36,974	11,589

[a] Distinct rules

of 421,976 issues (373,413 labeled as open, 36,974 as fixed, and 11,589 as won't fix or false positive).

IV. STUDY RESULTS

In this section we present the main findings of our research, answering the general research questions we investigate.

A. What are the practitioners' perceptions about the use of static analysis tools? (RQ1)

The findings from our survey indicate that: (a) developers consider ASATs warnings as relevant for overall software improvement (Q1), and (b) developers typically fix the issues along the implementation of bug fixes and new features (Q2), i.e., without creating specific tasks / branches for fixing the reported issues. Perhaps due to the small effort to fix part of the issues, this task does not become a first class planned activity, and thus may not require their own development branches.

More than 80% of the respondents said that they agree or strongly agree that the issues reported by ASAT tools are relevant for improving the design and implementation of software (Q1), as shown in Figure 1. Moreover, as Figure 2 shows, only 22.22% reject pull-requests based on the issues reported (Q5), and 50% never or rarely postpone a release based on the issues reported (Q6).

We find an apparent contradiction between the importance that developers claim ASATs and program transformation tools to automatically fix issues have (Q4), and how these tools are used by them during the software development process (Q3), we detail it in the next two paragraphs. Regarding the use of transformation tools to automatically fix issues reported by ASATs, more than 60% said that they consider them important or very important (see Figure 3). But, when asked about how often they actually use this type of tool for this purpose, only 22.22% answered very often, against 66.66% never or rarely, as shown in Figure 2. This might indicate that issues are not always solved in batch, which would benefit from the use of automatic program transformation tools. This also suggests that it would be worth to develop program transformation tools that address issues reported by static analysis tools and that could be integrated into continuous integration workflows.

TABLE III
DESCRIPTIVE STATISTICS RELATED TO THE NUMBER OF DAYS FOR FIXING ISSUES

Organization	Median	Mean	Sd
EF	24.59	299.11	435.19
ASF	6.67	222.16	298.73
TCU	47.14	282.60	435.43
PF	153.81	216.60	241.31

B. How often developers fix issues found by SonarQube? (RQ2)

To answer this research question we first investigate the interval in days between the date a given issue was created and the date it was closed. Here we only consider the 36,974 fixed issues (8.76% of all issues). The issues that developers quickly fix might represent relevant problems that should be fixed in a short period of time or could be a potential target for automation.

Figure 4 presents some descriptive statistics, considering the four organizations. Considering all projects, the median interval in days for fixing an issue is 18.99 days. There is a (median) central tendency of ASF developers to fix issues in 6.67 days (more details in Table III). Interestingly, the industrial projects studied take much longer to fix the SonarQube reported issues, when compared to the OSS studied projects. In a previous work, Panjer and colleagues reported that Eclipse bugs are fixed in 66.2 days on average [24]. Here we found that developers spend on average more time to fix issues reported by SonarQube (see Table III). In addition, we found 10,749 issues (29% of them) that were fixed after one year of the report—many of them are considered **major** issues (see Figure 7). These "long to be fixed" issues were found across all four organizations (ASF: 4,760, EF: 2,310, TCU: 2,954, PF: 726). It is also important to note that almost 50% of the open issues have been reported more than one year ago.

Altogether, our first conclusion is that the number of fixes is relatively small (8.77% of the total of issues), a lower value than reported by previous research. Developers often fix the issues in a reasonable time frame (median is 18.99 days), also a shorter period than previously reported periods for both bugs and ASATs violations. In addition, we found that almost one-third of the fixes occurs after one year the issue had been reported.

We used the approach proposed by Giger et al. [25] for classifying the resolutions time. It has only two status: *Fast* (fix interval $\leq$ median time to fix) and *Slow* (fix interval $>$ median time to fix). Since the median interval for fixing an issue is 18.99 days, we used this information to characterize our dataset of fixed issues in the plot of Figure 5. As one could see, most fixes (55%) related to ASF projects present a *Fast* resolution while the PF organization presents the slowest scenario, with 66% of its resolutions being considered *Slow*.

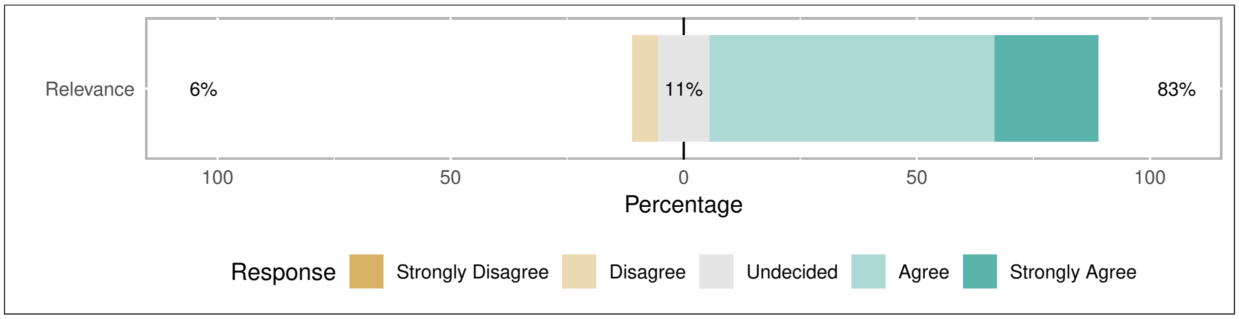

Fig. 1. Do you agree that warning messages reported by ASAT tools are relevant for improving the design and implementation of software? (Q1)

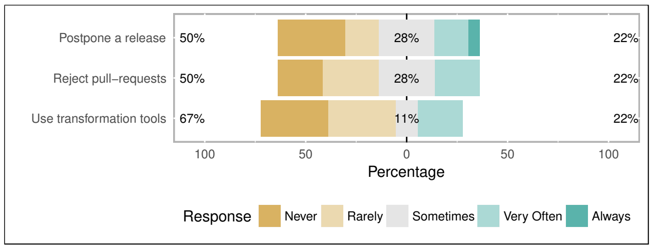

Fig. 2. Survey responses on whether respondents postpone a release (Q6), reject pull-requests (Q5), or use transformation tools (Q3).

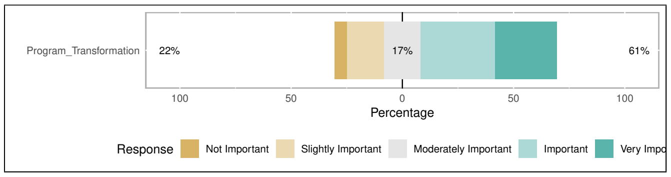

Fig. 3. How important is the use of program transformation tools to fix issues reported by ASATs? (Q4)

To better understand why the reported issues are taking too long to be fixed at PF, we present a closer look at PF issues resolutions in Table IV.

In our collaboration with PF, we found out that they work on two lines of software projects: one that maintains and evolves legacy software systems (mainly developed in Java 6), and another one that develops greenfield software systems, working with newer technologies (e.g., Java 8). This particular greenfield line of work also follows agile practices with monthly deliveries. Our analysis confirms the intuition that greenfield projects fix issues faster, and also have more issues fixed in total (1,862 for 26 greenfield projects *vs* 1,367 for 31 legacy projects). Conversely, legacy projects have a significantly larger number of open issues (27,599 *vs* 14,728 from greenfield projects).

Moreover, we also investigate whether or not the "severity" of the issues influences the interval in days for fixing the reported problems (see the boxplots in Figure 8). To further investigate this aspect, we executed the non-parametric *Kruskal-*

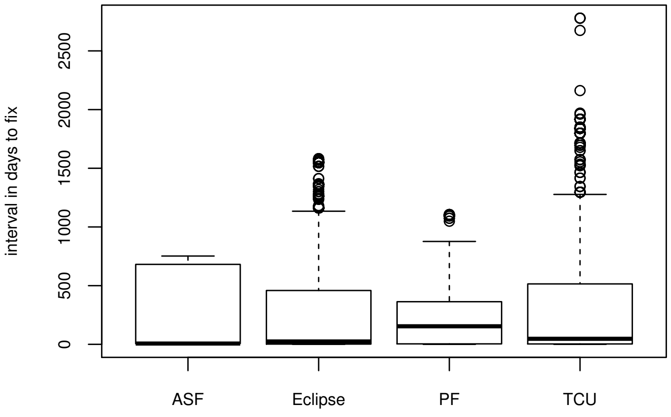

Fig. 4. Descriptive statistics with the interval in days to fix reported issues (grouped by organization)

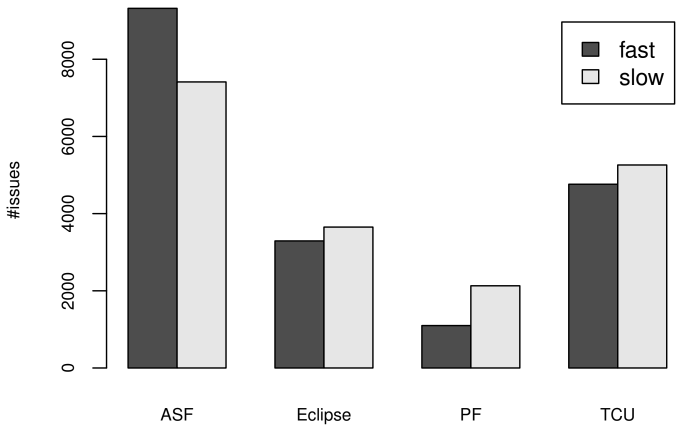

Fig. 5. Speed resolutions of the organizations

TABLE IV
SPEED RESOLUTIONS FOR PF'S DIFFERENT LINES OF SOFTWARE DEVELOPMENT PROJECTS

Projects category	Fix Class	Total
Greenfield[a]	Fast Resolutions	794 (24,6%)
Greenfield	Slow Resolutions	1068 (33,1%)
Legacy[b]	Fast Resolutions	305 (9,4%)
Legacy	Slow Resolutions	1062 (32,9%)

[a] 26 projects
[b] 31 projects

Wallis test and the *Dunn test* method for comparing mean differences, which (a) reveal that the severity factor influences the time for fixing issues (p–value $< 2.2e^{-16}$), and (b) give evidence that the Blocker and Minor severity categories are fixed in less time than the other categories. Figure 6 shows the outcomes of the *Dunn test*. We actually found surprising the observation that Minor issues have been fixed faster than Major and Critical issues. A possible reason might be that Minor issues are simpler to solve than the other issues.

```
Col Mean-|
Row Mean | BLOCKER   CRITICAL       INFO       MAJOR
---------+------------------------------------------
CRITICAL | -10.48
         |  0.0000*
         |
    INFO | -9.83       -0.32
         |  0.0000*    0.3717
         |
   MAJOR | -3.19       23.13      14.30
         |  0.0007*    0.0000*    0.0000*
         |
   MINOR | -0.69       29.67      18.60      12.48
         |  0.2445     0.0000*    0.0000*   0.0000*

alpha = 0.05
Reject Ho if p <= alpha/2
```

Fig. 6. Mean differences of the interval in days to fix issues, considering the severity of the issues

Blocker and Minor issues are solved faster than the other categories.

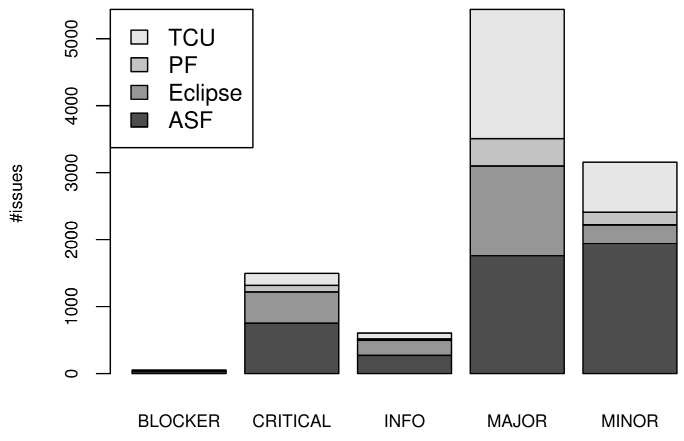

Fig. 7. Number of issues fixed after one year they had been reported

In order to further investigate our first research question, we also considered the frequency in which developers fix the issues reported by SonarQube. To this end, we computed a number of metrics for each project P.

MinDate(P) The first date an issue have been reported for a project P.

MaxDate(P) The last date an issue have been reported for a project P.

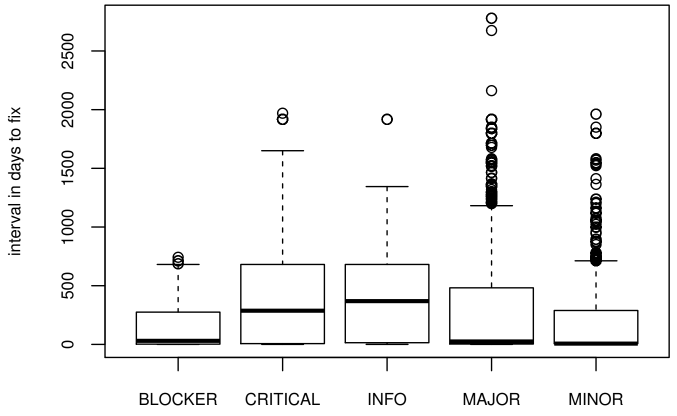

Fig. 8. Descriptive statistics with the interval in days to fix reported issues (grouped by severity)

Interval(P) The difference between the *max* and *min* dates for a project P (computed as $MaxDate(P) - MinDate(P) + 1$).

ODD(P) The number of distinct dates in which at least one issue has been opened in a project P.

FDD(P) The number of distinct dates in which at least one issue has appeared as fixed in a project P.

OpenFreq(P) The frequency in the interval where at least one issue have been opened in a project P (computed as $ODD(P) * 100/Interval(P)$).

FixFreq(P) The frequency in the interval where at least one issue have been fixed in a project P (computed as $FDD(P) * 100/Interval(P)$).

Table V summarizes some descriptive statistics related to these metrics. Based on **Interval(P)**, it is possible to realize that most projects in our dataset are using SonarQube for at least one year (median of **Interval(P)** is 415 days). Considering the full interval in days where the projects were using SonarQube, on average, issues have been reported in 15.47% of the days. A possible explanation is that, when a project starts using a static analysis tool (like SonarQube), several issues are reported at once. After that, while the development of a system makes progress, the frequency in which new flaws are introduced and reported becomes sparse (with seasonal peaks where many flaws are reported). Surely, this might also indicates either that SonarQube is not integrated into the development processes or that there is a lack of activity.

More interesting is that issues are less frequently fixed than they are reported—that is, on average, we found fixes in 9.91% of the days between **MinDate(P)** and **MaxDate(P)** for a given project P. We investigate the correlation (using the Spearman's method) between the total of distinct days the issues have been fixed (**FDD(P)**) and the total number of fixed issues of a project P. We find a moderate correlation within all organizations (EF with the least has 0.59, whereas ASF had the maximum of 0.69). This does not support the argument that issues are often fixed in "batch". Batch examples in the ASF ecosystem are projects *LDAP_API*, with 15 distinct dates and 6,832 fixed issues, and *Myfaces*, with 7 distinct dates and

3,856 fixed issues.

TABLE V
SOME DESCRIPTIVE STATISTICS RELATED TO THE FREQUENCY BASED METRICS

Metric	Median	Mean	SD
Interval(P)	415 days	667 days	772.71
OpenFreq(P)	4.71%	15.47%	26.07%
FixFreq(P)	0.67%	9.91%	25.61%

Therefore, the frequency in which new issues are either reported or fixed is relatively sparse, which might indicates that (1) SonarQube is not part of continuous integration workflows or that (2) developers do not act immediately when a new issue arises. Based on our findings, we can conclude that developers rarely fix issues reported by SonarQube.

Finally, we also investigate if there are specific days in which developers fix more issues. To this end, we collect the total number of issues fixed in each day of the week for each organization. Table VI reports the results. We found many fixes of TCU appearing on Saturdays (22.5% of them) and many fixes of the Eclipse Foundation appearing on Sundays (22.8%). Overall, 12.4% of the fixes occurred during the weekends (EF: 40%, ASF: 3.3%, TCU: 25.9%, and PF: 0%). Contrasting to the other organizations whose fixes appear more frequently during the weekdays. Actually, PF does not have any issues fixed on the weekends.

TABLE VI
ISSUES FIXED PER DAY OF THE WEEK

Org.	Sun	Mon	Tue	Wed	Thu	Fri	Sat
EF	1,294	1,284	605	1,132	923	1,031	699
ASF	383	724	7,470	1,520	5,515	961	158
TCU	218	1,015	1,390	1,874	2,253	1,428	1,843
PF	0	949	78	1,400	219	583	0
Total	1,895	3,972	9,543	5,926	8,910	4,003	2,700

C. What are the SonarQube issues that developers fix more frequently? (RQ3)

In our study we mined 36,974 fixed issues from all organizations. Even though the number of fixed violations is significantly low in comparison to the number of open violations, we still find some open issues that are worth fixing. To answer this research question, we studied the rules associated to the reported issues. Eclipse employs a deprecated rule by the SonarQube team which says that *Cycles between packages should be removed*. Since this rule is frequently fixed on EF's dataset, we introduce the type *Deprecated* to classify it. We found that both EF and ASF modify configurations for rules, either activating/deactivating rules, or changing rules severities. For instance, EF deactivates the rule *Useless imports shoud be removed*, which is the 10th most fixed rule in our dataset. As a result, this rule is not related to any violations in EF.

TABLE VII
MOST FREQUENTLY FIXED ISSUES BY TYPE IN EACH ORGANIZATION

Org.	Code Smell	Vulnerability	Bug	Deprecated
EF	2,533	399	116	3,945
ASF	15,077	322	1,332	–
TCU	8,837	677	507	–
PF	2,968	60	201	–
Total	29,415 (79.6%)	1,458 (3.9%)	2,156 (5.8%)	3,945 (10.7%)

At first, we quantify all fixed issues from all organizations' projects. We then classify them by their type, as shown in Table VII. It is possible to see that *Code Smells* are responsible for a high percentage (almost 80%) of all fixed issues. As we show in Table VIII, *Minor* issues are responsible for 21% of the fixed issues, and *Info* issues for 2.5%. We also find contrasting results regarding vulnerabilities as in our study: they represent approximately 4% of the total number of fixes, when compared to 0.2% and 0.5% reported in [12] and [13], respectively.

TABLE VIII
MOST FREQUENTLY FIXED ISSUES CLASSIFYING SEVERITY TO TYPE

Severity	Code Smell	Vulnerability	Bug	Deprecated	Total
Major	**19,732**	496	**972**	**3,945**	25,145 (68%)
Minor	7,683	53	91	–	7,827 (21.2%)
Critical	943	**883**	697	–	2,523 (6.8%)
Info	944	6	–	–	950 (2.6%)
Blocker	113	20	396	–	529 (1.4%)

Table IX presents the ten most frequently fixed issues. It is worthy noting that the five most fixed *Minor* issues correspond to almost 17% of the total fixed issues. Not surprisingly, *Code Smells* and *Major* issues are prevalent in the selection. Although code smells is the most fixed issues type, it is also responsible for the ten most frequent open issues, with six of them having a Major severity. Since Major issues can highly impact developers of a system (see II) and also represent a predominantly large portion (68%) of the fixed issues, we question whether ASATs issue prioritization is as ineffective as reported in related works [13], [26].

We find that a frequently fixed issue does not incur in high fixing rate. We found that two issues, *Sections of code should not be commented out* and *Generic exceptions should never be thrown*, are also present in the ten most common opened issues. If we consider 20 issues for both most fixed and most opened, there are 9 common issues between the two lists.

Finally, we investigate the occurrence of *Won't fix* and *False-positive* issues. We argue that marking an issue as one of these resolutions is similar to the process of fixing a violation. The developer has to filter the specific issue among all others, assess if it truly represents a quality flaw worthy of fixing, and then she must take an action. With that in mind, developers do tend to flag issues as won't fix and/or false-positive. Apache's projects flagged a total of 10.168 issues with these resolutions. We encounter similar findings when comparing ASF's ten most issues flagged as won't fix/false-positive and the foundation's ten most opened issues. There is a common subset of 5 issues among the two. These findings suggests that no rule is always fixed, regardless of context. Developers seem to consider other factors to decide whether to fix an issue or not.

Code smells and Major issues are highly prevalent among most of all issues' types and severities. We found common issues among the top ten most fixed, won't fix / false-positive, and opened issues. This suggests that developers consider a variety of factors when deciding whether to fix an issue.

EF and TCU SonarQube instances have a large number of customized rules. When comparing those rules to rules that are available in a fresh SonarQube installation, EF has 1.163 additional rules, and TCU has 1.533. Nonetheless, we found that just a subset of these rules actually lead to issues' reports. Overall, 122 unique rules are associated to fixed issues in EF, with 104 custom rules, which represents 7% of the total of custom rules. In TCU 250 unique rules are associated to fixed issues, with 141 custom rules, or 9% of TCU's custom rules.

D. How is the distribution of the SonarQube issues? (RQ4)

Taking into account the results of the previous section, here we answer our fourth research question, which investigates the concentration of the rules (20% of the rules correspond to 80% of the fixes) and the concentration of the files (20% of the files concentrate 80% of the issues). Answering to this question might help practitioners (a) to select a subset of rules that should be fixed (for instance, due to its relevance for source code improvement or easiness of fixing) or (b) to concentrate quality assurance activities in certain files of a project.

Considering all projects, we found a total of 412 rules having at least one fix. In this way, we consider the 82 most frequent fixed rules to answer RQ4—where 82 corresponds to 20% of the 412 rules. These 82 rules are related to 32,717 fixes. Since our dataset comprises 36,959, the 20% most frequent fixed rules correspond to 88.52% of all fixed issues. We publish this list of most frequent fixed rules in the paper's website (omitted here due to the blind review process).

We further analyse our dataset to verify which projects follow the distribution *20% of the rules correspond to 80% of the fixes)*. To avoid bias due to a small number of fixes, we constrain our analysis to projects having at least 16 fixes and 190 files (the median number of fixes and files per project, respectively), leading to a total of 80 projects. We found 62 projects (77.5%) in the rule 20% of the rules correspond to 80% of the fixes, which suggests that it would be possible to reduce the number of reported issues (and avoid false-positives and issues that would not be fixed) by correctly configuring SonarQube to report a relatively small subset of all rules—those issues that are more likely to be fixed.

Another recurrent question that arises in the literature [27], [28], [29] is whether or not 20% of the modules (files) are responsible for 80% of the issues (bugs in the existing literature). Investigating this issue might not only help managers to concentrate quality assurance activities on a subset of the modules of a project, but also might open new research directions to predict which files are more expected to present design flaws. More precisely, here we investigate if 20% of the files of each project (with at least 16 fixes and 190 files in our dataset) concentrate at least 80% of the issues. Interesting, we did not find any project satisfying this distribution. Considering the median statistic, the top 20% of files containing more issues represent 35.79% of all issues of a project (mean: 37.23 and max: 63.37). Comparing with the literature aforementioned, which suggests a higher concentration of bugs, we can conclude that static analysis issues are more widespread throughout the modules of a system than bugs.

We found that 20% of the rules correspond to 80% of the fixes, and that the issues reported by static analysis tools are not localized in a relatively small subset of the files of the projects.

V. Discussion

Our findings show contrasting results at first. Practitioners find ASATs reports relevant to the software development process, and, in some situations, reject pull-requests or even postpone the release of a software based on the outcomes of these tools.

Our investigation also reveals that the resolution time for fixing issues is faster than the time previously reported for fixing bugs. Although these results strongly support that developers indeed use ASATs and take their warnings in consideration, we find that fixed issues only represent 8.76% of the 421,976 mined issues, which suggests that not all issues are relevant to developers, as supported by the finding that 20% of the rules correspond to 80% of the fixes.

Our results also indicate that practitioners can greatly benefit from the usage of ASATs if they properly configure them to mostly consider rules that they find relevant and are more likely to fix. This might help to control the pressure related to the technical debt of the systems, often calculated using ASAT reports. Developers could also benefit from tool support to fix ASATs issues, since most of them consider important the use of tools that provide automatic fixes, but at the same time most never or rarely use them. We envision that our findings, such as the big prevalence of fixed *Code Smells* and Major issues, can provide insights to tools developers.

Our findings still unfold several unanswered observations pertaining to the comprehension on how developers fix and perceive ASAT issues. An organization, or a particular team or project, might have a policy to fix all major issues, thus impacting on which kind of violations are fixed. In cases that

TABLE IX
MOST FREQUENTLY FIXED ISSUES IN ALL ORGANIZATIONS

Issue	Type	Severity	Count	EF	ASF	PF	TCU
Cycles between packages should be removed	D[a]	Major	3,945 (10.7%)	3,945 (100%)	0 (0%)	0 (0%)	0 (0%)
Checked exceptions should not be thrown	CS[b]	Major	2,053 (5.5%)	0 (0%)	0 (0%)	0 (0%)	2,053 (100%)
Sections of code should not be commented out	CS	Major	1,903 (5.1%)	182 (9.6%)	1,014 (53.3%)	364 (19.1%)	343 (18%)
The diamond operator should be used	CS	Minor	1,871 (5%)	0 (0%)	1,716 (91.7%)	155 (8.3%)	0 (0%)
Nested code blocks should not be used	CS	Minor	1,380 (3.7%)	0 (0%)	1,374 (99.6%)	6 (0.4%)	0 (0%)
throws declarations should not be superfluous	CS	Minor	1,352 (3.6%)	0 (0%)	623 (46.1%)	77 (5.7%)	652 (48.2%)
Generic exceptions should never be thrown	CS	Major	1,334 (3.6%)	59 (4.4%)	129 (9.7%)	20 (1.5%)	1,126 (84.4%)
Redundant pairs of parentheses should be removed	CS	Major	961 (2.6%)	0 (0%)	741 (77.1%)	109 (11.3%)	111 (11.6%)
Local variable and method parameter names should comply with a naming convention	CS	Minor	823 (2.2%)	16 (2%)	774 (94%)	33 (4%)	0 (0%)
Useless imports should be removed	CS	Minor	784 (2.1%)	0 (0%)	517 (66%)	180 (23%)	87 (11%)

[a] Deprecated
[b] Code Smell

ASATs are integrated in the development workflow, several reasons for developers not fixing issues are possible, such as lack of configuration, unawareness on how to perform fixes, value of fixing an issue, or even time pressure. These open questions suggest future research focused on organizations, and/or teams / projects, that use ASATs as part of their workflow.

Finally, mining issues from SonarQube can be challenging, specially when considering different instance versions and different host organizations. To help further research aiming at mining SonarQube issues, we recommend researchers to mine rules for the chosen language(s). As an example, EF most fixed rule was a custom one, that would not be analyzed if rules were not mined or a revision approach was used.

VI. THREATS TO VALIDITY

Identifying if an issue is fixed intentionally is a commonly reported threat among studies that analyze ASATs [12], [13], [26]. Common mitigation strategies involve mining source code management repositories to look for patterns in commit messages[5], [13], [26], or to find references for bug reports identifiers, such as #4223, that are hosted on issue management platforms [26]. Although in our study we did not mine commits, we minimize the threat on our private dataset, as we know for sure, by means of our collaborations, that PF and TCU developers use SonarQube. Regarding OSS projects, we have indication that SonarQube is used, as for ASF we had respondents in our survey, and for EF, SonarQube is mandatory for projects that aim to achieve a higher maturity assessment. We believe that the projects we studied in this work are well suited for our analyses. Our results might be partially generalized to companies and OSS projects. We study 246 projects, 155 from large Brazilian government institutions, and 91 OSS projects from two well-known open-source foundations. However, since Eclipse Foundation and Apache Software Foundation are well matured foundations, with well defined standards and practices, they may not represent the general OSS community. Our choice of SonarQube as the targeted ASAT for this study might not properly contextualize general usage of ASATs. We believe that this threat is minimized as SonarQube is used by more than 85,000 organizations, and also encompasses rules from other ASATs, such as FindBugs, and PMD [10].

Our technical decisions might also introduce some threats to internal validity. That is, since we mined data from different versions of SonarQube instances, with different APIs, our approach for data extraction and filtering might have errors. However, we manually verified parts of our data, and in some cases we verified both our data and findings with collaborators from TCU and PF. Another major threat is the reliance on measuring issues' open and creations dates only from the data extracted from SonarQube. It is possible that an issue fix, for example, may have happened in a different moment than the tool was run, and thus the date reported by SonarQube might not reflect a precise date/time on when the issue was fixed. As we observed in PF, this limitation may be minimized by *nightly builds* (tools are executed automatically at the end of each day). We tried to mitigate several conclusion threats by running all statistic analysis in pairs. At least two authors of this paper double-checked the procedures, statistical methods we use, and results.

VII. RELATED WORK

Beller et al. [11] performed a large-scale evaluation on how nine different ASATs are used. They investigate how ASATs are configured by analyzing 168 214 OSS projects, for Java and other three popular programming languages, and reported that the default configurations of most tools fits the needs of the majority of projects–custom rules are used in less than 5% of the cases. We find diverging results as we argue that developers should choose more carefully the rules ASATs check, and a big portion of EF and TCU fixed rules are related to custom rules. Regarding ASAT usage in a CI context, Rausch et al. [30] performed an in-depth analysis of the build failures of 14 projects, and found that 10 of these projects present a history of build failures related to violations reported by ASATs. Zampetti et al. [5] analyze 20 Java OSS projects and report that build breakages are mainly related to adherence to coding guidelines, while build failures originated by potential bugs or vulnerabilities do not occur frequently.

Our results confirm that a large portion of fixed issues are related to code smells, which cover coding standards and other aspects.

Vassalo et al. [10] conducted a study on 119 OSS projects, mined from SonarCloud (cloud service based on SonarQube), and concluded that developers check code quality only at the end of a sprint, contrary to CI principles. In this study we find that developers tend to fix issues from 216.60 days to 299.12 on average, after they had been reported. However, we find in ASF's projects a central tendency to fix issues in 6.67 days after the report. Kim and Ernest [26] observed warnings by three distinct ASATs, finding that no more than 9% are removed during fix changes. They suggest that issues' prioritization given by ASATs are ineffective. We question ASAT's ineffectiveness on prioritizing issues, largely due to developers mainly fixing Major issues, which are supposed to highly impact developers' productivity.

Recently, studies have focused on what kind of violations developers tend to fix. Liu et al. [13] collected and tracked a large number of FindBugs fixed and unfixed violations across revisions of 730 Java projects. They observed cases that warnings are systematically ignored, and violations categories that are frequently fixed. Furthermore, they found that most developers do not use FindBugs as part of their development tools. Digkas et al. [12] analyzed 57 Java projects from Apache Software Foundation to find what kind of SonarQube issues are fixed and the amount of Technical Debt that is paid over time. Their findings revealed that a small subset of issues is responsible for a large portion of fixes. They extract several statistics from fixed violations, such as most frequent types resolved for all categories and issues with highest and lowest fixing rate. Their study was restricted to only issues with Blocker, Critical and Major severities.

Our study differs from [13] and [12] by mining data from real AST (SonarQube) usage. Both studies mention the challenges of analyzing violations from revisions of project source code, more importantly in regard to the very high cost to run the tools in this fashion, which can be done either by analyzing all revisions [13] or limiting the analysis to weekly revisions [12]. The last approach can harm analyses that investigate time sensitive data, such as our research question RQ2. Another problem with this approach is that the lack of build information frequently leads to false positives [12] (e.g., a Java 8 rule might trigger violations in a Java 7 project). Moreover, custom configurations and rules are ignored as they run fresh installations of the ASATs. We found in EF and ASF that some rules had their severities changed, and others were deactivated. Lastly, issues that are resolved as either won't fix, or false-positive, would never be identified in this setting. Not only our study avoid all these issues, we are assured that the entirety of the projects that compose our dataset are configured to use SonarQube. We argue that this fact has significant impact on how developers fix and perceive issue reports from ASATs.

VIII. Conclusion

In this work we reported the results of a multi-method study about how developers use SonarQube (one of the most used tools for static quality assurance). We first collected the perceptions of 18 developers from different organizations, regarding the use of static analysis tools. Most respondents of the survey agree that these tools are relevant for the overall improvement of software quality. By mining four instances of SonarQube, we built a general comprehension of the practices for fixing issues that this tool reports. We found a low rate of fixed issues and that one-third of the fixes occurs after one year of the issue's report. In addition, we showed evidences that 20% of the violation rules correspond to 80% of the fixes, which can assist practitioners to properly select a subset of rules that are relevant, and discard rules that are rarely fixed.

References

[1] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar," in *Proceedings of the 15th International Conference on Mining Software Repositories - MSR'18*. ACM Press, 2018. [Online]. Available: https://doi.org/10.1145/3196398.3196473

[2] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 672–681.

[3] J. Wang, S. Wang, and Q. Wang, "Is there a "golden" feature set for static warning identification?" in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM'18*. ACM Press, 2018. [Online]. Available: https://doi.org/10.1145/3239235.3239523

[4] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. 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)*. IEEE, mar 2018. [Online]. Available: https://doi.org/10.1109/saner.2018.8330195

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

[6] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of android asynchronous programming (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 224–235.

[7] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, 2018, pp. 181–191.

[8] G. Pinto, A. Canino, F. Castor, G. H. Xu, and Y. D. Liu, "Understanding and overcoming parallelism bottlenecks in forkjoin applications," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 765–775.

[9] M. F. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, "Code smells for model-view-controller architectures," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2121–2157, 2018.

[10] C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall, "Continuous code quality: are we (really) doing that?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. ACM Press, 2018. [Online]. Available: https://doi.org/10.1145/3238147.3240729

[11] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, mar 2016. [Online]. Available: https://doi.org/10.1109/saner.2016.105

[12] 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)*. IEEE, mar 2018. [Online]. Available: https://doi.org/10.1109/saner.2018.8330205

[13] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. L. Traon, "Mining fix patterns for FindBugs violations," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018. [Online]. Available: https://doi.org/10.1109/tse.2018.2884955

[14] J. L. C. Izquierdo and J. Cabot, "The role of foundations in open source projects," in *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Society - ICSE-SEIS'18*. ACM Press, 2018. [Online]. Available: https://doi.org/10.1145/3183428.3183438

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

[16] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, p. 92, dec 2004. [Online]. Available: https://doi.org/10.1145/1052883.1052895

[17] SonarSource S.A., "Sonarqube," 2019, [accessed 18-January-2019]. [Online]. Available: https://www.sonarqube.org

[18] I. E. Allen and C. A. Seaman, "Likert scales and data analyses," *Quality progress*, vol. 40, no. 7, p. 64, 2007.

[19] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013. [Online]. Available: https://doi.org/10.1109/icsm.2013.39

[20] M. Hollander and D. Wolfe, *Nonparametric Statistical Methods*, ser. Wiley Series in Probability and Statistics. Wiley, 1999.

[21] O. J. Dunn, "Multiple comparisons among means," *Journal of the American statistical association*, vol. 56, no. 293, pp. 52–64, 1961.

[22] M. M. Mukaka, "A guide to appropriate use of correlation coefficient in medical research," *Malawi Medical Journal*, vol. 24, no. 3, pp. 69–71, 2012.

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

[24] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, May 2007, pp. 29–29.

[25] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 52–56. [Online]. Available: http://doi.acm.org/10.1145/1808920.1808933

[26] S. Kim and M. D. Ernst, "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 Press, 2007. [Online]. Available: https://doi.org/10.1145/1287624.1287633

[27] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, Aug 2000.

[28] P. Runeson and C. Andersson, "A replicated quantitative analysis of fault distributions in complex software systems," *IEEE Transactions on Software Engineering*, vol. 33, pp. 273–286, 2007.

[29] N. Walkinshaw and L. L. Minku, "Are 20% of files responsible for 80% of defects?" in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*. ACM, 2018, pp. 2:1–2:10.

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