A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision

Valentina Lenarduzzi[a], Fabiano Pecorelli[b], Nyyti Saarimaki[b], Savanna Lujan[b], Fabio Palomba[c]

[a]*M3S Research Unit - University of Oulu, Finland*
[b]*Clowee Research group - Tampere University, Finland*
[c]*SeSa Lab - University of Salerno, Italy*

Abstract

Background. Developers use Static Analysis Tools (SATs) to control for potential quality issues in source code, including defects and technical debt. Tool vendors have devised quite a number of tools, which makes it harder for practitioners to select the most suitable one for their needs. To better support developers, researchers have been conducting several studies on SATs to favor the understanding of their actual capabilities.
Aims. Despite the work done so far, there is still a lack of knowledge regarding (1) what is their agreement, and (2) what is the precision of their recommendations. We aim at bridging this gap by proposing a large-scale comparison of six popular SATs for Java projects: Better Code Hub, CheckStyle, Coverity Scan, FindBugs, PMD, and SonarQube.
Method. We analyze 47 Java projects applying 6 SATs. To assess their agreement, we compared them by manually analyzing - at line- and class-level - whether they identify the same issues. Finally, we evaluate the precision of the tools against a manually-defined ground truth.
Results. The key results show little to no agreement among the tools and a low degree of precision.
Conclusions. Our study provides the first overview on the agreement among different tools as well as an extensive analysis of their precision that can be used by researchers, practitioners, and tool vendors to map the current capabilities of the tools and envision possible improvements.

1. Introduction

Static analysis tools (SATs) are instruments that analyze source code without executing it, in an effort to discover potential source code quality issues [1]. These tools are getting more popular as they are becoming easier to use—especially in continuous integration pipelines [2]—and there is a wide range to choose from [3]. However, as the number of available tools grows, it becomes harder for practitioners to choose the tool (or combination thereof) that is most suitable for their needs [4].

To help practitioners with this selection process, researchers have been conducting empirical studies to compare the capabilities of existing SATs [5, 6]. Most of these investigations have focused on (1) the features provided by the tools, e.g., which maintainability dimensions can be tracked by current SATs, (2) comparing specific aspects considered by the tools, such as security [7, 8] or concurrency defects [9], and (3) assessing the number of false positives given by the available SATs [10].

Recognizing the effort spent by the research community, which led to notable advances in the way tool vendors develop SATs, we herein notice that our knowledge on the capabilities of the existing SATs is still limited. More specifically, in the context of our research we point out that three specific aspects are under-investigated: (1) which source quality problems can actually be detected by static analysis tools, (2) what is the agreement among different tools with respect to source code marked as potentially problematic, and (3) what is the precision with which a large variety of the available tools provide recommendations. An improved knowledge of these aspects would not only allow practitioners to take informed decisions when selecting the tool(s) to use, but also researchers/tool vendors to enhance the tool and improve the level of support provided to developers.

In this paper, we propose a large-scale empirical investigation into the detection capabilities of six of the most widely used SATs, namely SonarQube, Better Code Hub, Coverity Scan, FindBugs, PMD, and CheckStyle.[1] Specifically, we run the considered tools against a corpus of 47 projects from the Qualitas Corpus dataset, (1) showcasing the functionalities and distribution of source code quality

Email addresses: valentina.lenarduzzi@oulu.fi (Valentina Lenarduzzi), fabiano.pecorelli@tuni.fi (Fabiano Pecorelli), nyyti.saarimaki@tuni.fi (Nyyti Saarimaki), savanna.lujan@tuni.fi (Savanna Lujan), fpalomba@unisa.it (Fabio Palomba)

[1]SATs verify code compliance with a specific set of rules that, if violated, can introduce an issue in the code. This issue can be accounted for as "source code quality issue": as such, in the remaining the paper we use this term when referring to the output of the considered tools.

issues detected by the tools; (2) computing the agreement among the recommendations given by them at line-level; and (3) manually computing the precision of the tools.

The key results of the study shows that, among the considered tools, SonarQube is the one able to detect most of the quality issues that can be detected by the other SATs. However, when considering the specific quality issues detected, there is little to no agreement among the tools, indicating that different tools are able to identify different forms of quality problems. Finally, the precision of the considered tools ranges between 18% and 86%, meaning that the practical accuracy of some tools is seriously threatened by the presence of false positives—this result corroborates and enlarges previous findings [10] on a larger scale and considering a broader set of tools.

To sum up, the main contribution of our work is represented by the largest empirical analysis up to date on the capabilities of existing static analysis tools—a more detailed description of how our work compares to previous studies in the field is reported in the next section. Specifically, we advance the current state of the art in three different manners:

1. By providing an overview of the features and types of source code quality concerns detectable by six popular static analyzers, which may be used by practitioners as a way to select the most suitable tool(s) based on the specific needs of a project;

2. By investigating the agreement among the considered tools, which can inform tool vendors about the limitations of the current solutions available the market, other than making practitioners aware of how to benefit more from the combined capabilities of existing static analysis tool;

3. By providing a quantification of the precision of six static analysis tools, which may used to describe their accuracy in practice, hence alerting developers on the actual effectiveness of those tools.

Structure of the paper. Section 2 discusses the related work in the field of empirical studies on static analysis tools, highlighting how our work advances the state of the art. Section 3 reports on the tools selected for the empirical investigation, while Section 4 presents the specific research questions targeted by our study and the methods employed to address them. The results are presented in Section 5 and further elaborated in Section 6. Section 7 identifies the threats to the validity of our study. Finally, in Section 8 we draw conclusions and provide an outlook on our future research agenda.

2. Related Work

Static analysis tools (SATs) are getting more popular [3, 11] as they are becoming easier to use [2]. The use of static analysis tools has been studied by several researchers in the last years [12, 13, 14, 15]. In this section, we report the relevant work on static analysis tools focusing on their usage [16, 17, 18], rules and the detected problems [19, 20, 21].

SATs has been investigating considering which tools are being used, which types of issues are detected [22, 2], and the effective solving time [23], considering projects developed in different language [24]. Results showed that the most violated rules are related to adherence to coding standards and missing licenses [2]. Looking at the which issues is fixed and the related fixing time in average 13% of the issues have been solved in the systems [23].

Developers can use SATs, such as SonarQube[3.6] and CheckStyle[2], to evaluate software source code, finding anomalies of various kinds in the code [25, 26]. Moreover, SATs are widely adopted in many research studies in order to evaluate the code quality [10, 27, 23] and identify issues in the code [16, 17, 18]. Some studies demonstrated that some rules detected by SATs can be effective for identifying issues in the code [14, 28, 18]. However, evaluating the performance in defect prediction, results are discordant comparing different tools (e.g. FindBugs[3.4] and PMD[3.5]) [29].

Rutar et al. [25] compared five bug-finding tools for Java (Bandera[3], ESC/Java2[4], FindBugs[5], JLint[6], and PMD[7]), that use syntactic bug pattern detection, on five projects, including JBoss 3.2.3[8] and Apache Tomcat 5.019[9]. They focused on the different rules (also called rules) provided by each tool, and their results demonstrate some overlaps among the types of errors detected, which may be due to the fact that each tool applies different trade-offs to generate false positives and false negatives. Overall, they stated that rules provided by the different tools are not correlated with each other. Complementing the work by Rutar et al. [25], we calculated the agreement of SATs on TD identification. In addition, we investigated the precision with which these tools output rules. Finally, we also investigated the types of TD items that can actually be detected by existing SATs.

Tomas et al. [26] performed a comparative analysis by means of a systematic literature review. In total, they compared 16 Java code SATs, including JDepend[10], FindBugs[3.4], PMD[3.5], and SonarQube[??]. They focused on internal quality metrics of a software product and software tools of static code analysis that automate measurement of these metrics. As results, they reported the tools' detection strategies and what they detect. For instance, most of

[2] https://checkstyle.sourceforge.io/
[3] http://bandera.projects.cs.ksu.edu/
[4] https://kindsoftware.com/products/opensource/ESCJava2/
[5] http://findbugs.sourceforge.net/
[6] http://jlint.sourceforge.net/
[7] https://pmd.github.io/
[8] http://www.jboss.org/
[9] http://jakarta.apache.org/tomcat
[10] https://github.com/clarkware/jdepend

them automate the calculation of internal quality metrics, the most common ones being code smells, complexity, and code size [26]. However, they did not investigate agreement between the tools' detection rules.

Avgeriou et al. [30] identified the available SATs for the Technical Debt detection. They compared features and popularity of nine tools investigating also the empirical evidence on their validity. Results can help practitioners and developers to select the suitable tool against the other ones according to the measured information that satisfied better their needs. However, they did not evaluate their agreement and precision in the detection.

Focusing on developers' perception on the SATs usage, they can help to find bugs [10]. However, developers are not sure about the usefulness of the rules [31, 32, 33], they do pay attention to different rules categories and priorities and remove violations related to rules with high severity [32] to avoid the possible risk of faults [31]. Moreover, false positives and the way in which the rules are presented are barriers to their wider adoption [10]. Some studies highlighted the need to reduce the number of detectable rules [34, 35] or summarize them based on similarities [32]. SATs are able to detect many defects in the code. However, some tools do not capture all the possible defect even if they could be detected by the tools [36]. Even if some studies since the beginning of 2010 highlighted the need to better clarify the precision of the tools, differentiating false positives from actionable rules [37, 38], many studies deal with the many false positives produced by different tools, such as FindBugs[3.4] [36, 39, 40], JLint[6], PMD[3.5], CheckStyle[3.2], and JCSC[11] [36].

The two closest works with respect to ours are those by Mantere et al. [5] and Wilander et al [6]. Both of them investigated and compared existing security vulnerability SATs. More specifically, Mantere et al. [5] executed three SATs (Fortify SCA, Splint, and Frama-C) on a project reporting the amount of violated security rules without comparing their detection agreement. Wilander et al [6] compared five SATs (Flawfinder, ITS4, RATS, Splint, and BOON) and verified their detection capability of four security vulnerabilities (Changing the flaw of control, Bugger overflow attacks, Buffer overflow vulnerabilities, and Format string attacks) on 20 vulnerable functions selected from ITS4's vulnerability database. Differently from our work, the two papers just discussed only focused on security vulnerabilities. In addition, none of them performed additional analyses aiming at shedding lights on their detection agreement and precision. As such, our work represents the first attempt to investigate the capabilities of a large amount of static analysis tools under multiple perspectives, providing researchers, practitioners, and tool vendors with insights into (1) the features and types of issues they detect; (2) the potential usefulness given by their combination; and (3) the limitations in terms of precision.

[11]http://jcsc.sourceforge.net

3. Selection of the Static Analysis Tools

In this section, we describe the SATs we selected for this work and their code quality issues detection capability.

In particular, we selected six SATs based on two main observations. First and foremost, we selected the tools that have been previously investigated by researchers in the field with respect to their adoption [3, 32, 30] and were found to be the most widely employed in practice [3]. In the second place, the selected tools were familiar to the authors: such a familiarity allowed us to (1) use/run them better (e.g., by running them without errors) and (2) analyze their results better, for instance by providing qualitative insights able to explain the reasons behind the achieved results. The analysis of other tools is already part of our future research agenda. Table 1 reports the detection capability of each tool in terms of how many rules can be detected, and the classification of internal rules (e.g., type and severity). Moreover, we report the diffusion of the rule in the selected projects.

3.1. Better Code Hub

Better Code Hub[12] is a commonly used static analysis tool that assesses code quality. The analysis is done through the website's API, which analyzes the repository from GitHub. The default configuration file can be modified for customization purposes. Code quality is generally measured based on structure, organization, modifiability, and comprehensibility.

This is done by assessing the code against ten *guidelines*: write short units of code, write simple units of code, write code once, keep unit interfaces small, separate concern in modules, couple architecture components loosely, keep architecture components balanced, keep your code base small, automate tests, and write clean code. Out of the ten *guidelines*, eight *guidelines* are grouped based on type of *severity*: medium, high, and very high. *Compliance* is rated on a scale from 1-10 based on the results[13].

Better Code Hub static analysis is based on the analysis of the source code against heuristics and commonly adopted coding conventions. This gives a holistic view of the health of the code from a macroscopic perspective.

It detects a total of 10 rules, of which 8 are grouped based on type and severity. Better Code Hub categorizes the 8 rules under 3 types: *RefactoringFileCandidateWithLocationList*, *RefactoringFileCandidate*, and *RefactoringFileCandidateWithCategory*. Of these 8 rules, one is of *RefactoringFileCandidateWithLocationList* type, six are of *RefactoringFileCandidate* type, and one is of *RefactoringFileCandidateWithCategory* type. In addition to the types, Better Code Hub assigns three possible severities to the rules: *Medium*, *High*, and *Very High*. Of these eight rules, four were classified as *Medium* severity, four as *High*

[12]`https://bettercodehub.com/`
[13]`https://pybit.es/bettercodehub.html`

Table 1: Detection capability of the six selected SATs. For each tool, the number of supported rules, the number of rules categories, the number of severity levels and the description of severity levels are reported.

Tool	**Detection Capability**			
	# rule	**#Type group**	**#Severity levels**	**Severity levels**
Better Code Hub	10	3	3	Medium, High, and Very High
Checkstyle	173	14	4	Error, Ignore, Info, and Rule
Coverity	130	-	3	Low, Medium, and High
FindBugs	424	9	4	Of concern, Troubling, Scary, and Scariest
PMD	305	8	5	from 1 (most severe) to 5 (least severe)
SonarQube	413	3	5	Info, Minor, Major, Critical, and Blocker

severity, and eight as *Very High* severity. Some of the rules have more than one severity possibly assigned to them.

3.2. Checkstyle

Checkstyle[14] is an open-source tool that evaluates Java code quality. The analysis is done either by using it as a side feature in Ant or as a command line tool. Checkstyle assesses code according to a certain coding standard, which is configured according to a set of *checks*. Checkstyle has two sets of style configurations for standard *checks*: Google Java Style[15] and Sun Java Style[16]. In addition to standard *checks* provided by Checkstyle, customized configuration files are also possible according to user preference.[17] These *checks* are classified under 14 different categories: annotations, block checks, class design, coding, headers, imports, javadoc comments, metrics, miscellaneous, modifiers, naming conventions, regexp, size violations, and whitespace. Moreover, the violation of the *checks* are grouped under two severity levels: error and rule[18], with the first reporting actual problems and the second possible issues to be verified.

It detects a total of 173 rules which are grouped based on type and severity. Checkstyle categorizes the 173 rules under 14 types: *Annotations*, *Block Checks*, *Class Design*, *Coding*, *Headers*, *Imports*, *Javac Comments*, *Metrics*, *Miscellaneous*, *Modifiers*, *Naming Conventions*, *Regexp*, *Size Violations*, and *Whitespace*. Of these 173 rules, 8 are of *Annotations* type, 6 are of *Block Checks* type, 9 are of *Class Design* type, 52 are of *Coding* type, 1 is of *Headers* type, 8 are of *Imports* type, 19 are of *Javac Comments* type, 6 are of *Metrics* type, 16 are of *Miscellaneous* type, 4 are of *Modifiers* type, 16 are of *Naming Conventions* type, 4 are of *Regexp* type, 8 are of *Size Violations* type, and 16 are of *Whitespace* type. In addition to these types, Checkstyle groups these checks under four different severity levels: *Error*, *Ignore*, *Info*, and *rule*. The distribution of the checks with respect to the severity levels is not provided in the documentation.

[14]https://checkstyle.org
[15]https://checkstyle.sourceforge.io/google_style.html
[16]https://checkstyle.sourceforge.io/sun_style.html
[17]https://checkstyle.sourceforge.io/index.html
[18]https://checkstyle.sourceforge.io/checks.html

3.3. Coverity Scan

Coverity Scan[19] is another common open-source static analysis tool. The code build is analyzed by submitting the build to the server through the public API. The tool detects defects and vulnerabilities that are grouped by *categories* such as: resource leaks, dereferences of NULL pointers, incorrect usage of APIs, use of uninitialized data, memory corruptions, buffer overruns, control flow issues, error handling issues, incorrect expressions, concurrency issues, insecure data handling, unsafe use of signed values, and use of resources that have been freed[20]. For each of these *categories*, there are various issue *types* that explain more details about the defect. In addition to issue *types*, issues are grouped based on *impact*: low, medium, and high. The static analysis applied by Coverity Scan is based on the examination of the source code by determining all possible paths the program may take. This gives a better understanding of the control and data flow of the code[21]. Coverity Scan's total scope of detectable rules as well as the classification is not known, since its documentation requires being a client. However, within the scope of our results, Coverity Scan detected a total of 130 rules. These rules were classified under three severity levels: *Low*, *Medium*, and *High*. Of these 130 rules, 48 were classified as *Low* severity, 87 as *Medium* severity, and 12 as *High* severity. Like Better Code Hub, some of Coverity Scan's rules have more than one severity type assigned to them.

3.4. FindBugs

FindBugs[22] is a static analysis tool for evaluating Java code, more precisely Java bytecode. Despite analyzing bytecode, the tool is able to highlight the exact position of an issue if also the source code is provided to the tool [23]. The analysis is done using the GUI, which is engaged through the command line. The analysis applied by the tool is based on detecting *bug patterns*. According to FindBugs, the *bug patterns* arise for the following main reasons: difficult language features, misunderstood API features, misunderstood invariants when code is modified during maintenance, and garden variety mistakes.[24]

[19]https://scan.coverity.com/
[20]https://scan.coverity.com/faq\#what-is-coverity-scan
[21]https://devguide.python.org/coverity
[22]http://findbugs.sourceforge.net
[23]http://findbugs.sourceforge.net/manual/gui.html
[24]http://findbugs.sourceforge.net/findbugs2.html

[25] Such *bug patterns* are classified under 9 different categories: bad practice, correctness, experimental, internationalization, malicious code vulnerability, multithreaded correctness, performance, security, and dodgy code. Moreover, the *bug patterns* are ranked from 1-20. Rank 1-4 is the *scariest* group, rank 5-9 is the *scary* group, rank 10-14 is the *troubling* group, and rank 15-20 is the *concern* group[26]. It detects a total of 424 rules grouped based on type and severity. It categorizes the 424 rules under 9 types: *Bad practice*, *Correctness*, *Experimental*, *Internationalization*, *Malicious code vulnerability*, *Multithreaded correctness*, *Performance*, *Security*, and *Dodgy code.* Of these 424 rules, 88 are of *Bad practice* type, 149 are of *Correctness*, 3 are of *Experimental*, 2 are of *Internationalization*, 17 are of *Malicious code vulnerability*, 46 are of *Multithreaded correctness*, 29 are of *Performance*, 11 are of *Security*, and 79 are of *Dodgy code.* In addition to these types, FindBugs ranks these 'bug patterns' from 1-20. Rank 1-4 is the *scariest* group, rank 5-9 is the *scary* group, rank 10-14 is the *troubling* group, and rank 15-20 is the *concern* group.

3.5. PMD

PMD[27] is a static analysis tool mainly used to evaluate Java and Apex, even though it can also be applied to six other programming languages. The analysis is done through the command line using the tool's binary distributions. PMD uses a set of *rules* to assess code quality according to the main focus areas: unused variables, empty catch blocks, unnecessary object creation, and more. There are a total of 33 different rule set configurations[28] for Java projects. The rule sets can also be customized according to the user preference[29]. These *rules* are classified under 8 different categories: best practices, code style, design, documentation, error prone, multi threading, performance, and security. Moreover, the violations of *rules* are measured on a priority scale from 1-5, with 1 being the most severe and 5 being the least[30].

PMD detects a total of 305 rules which are grouped based on type and severity. PMD categorizes the 305 rules under 8 types: *Best Practices*, *Code Style*, *Design*, *Documentation*, *Error Prone*, *Multithreading*, *Performance*, and *Security.* Of these 305 rules, 51 are of *Best Practices*, 62 are of *Code Style*, 46 are of *Design*, 5 are of *Documentation*, 98 are of *Error Prone*, 11 are of *Multithreading*, 30 are of *Performance*, and 2 are of *Security* type. In addition to the types, PMD categorizes the rules according to five priority levels (from P1 "Change absolutely required" to P5 "Change highly optional"). Rule priority guidelines for default and custom-made rules can be found in the PMD project documentation.[31]

3.6. SonarQube

SonarQube[32] is one of the most popular open-source static code analysis tools for measuring code quality issues. It is provided as a service by the `sonarcloud.io` platform or it can be downloaded and executed on a private server. SonarQube computes several metrics such as number of lines of code and code complexity, and verifies code compliance with a specific set of "coding rules" defined for most common development languages. If the analyzed source code violates a coding rule, the tool reports an "issue". The time needed to remove these issues is called remediation effort.

SonarQube includes reliability, maintainability, and security rules. Reliability rules, also named *Bugs*, create quality issues that "represent something wrong in the code" and that will soon be reflected in a bug. *Code smells* are considered "maintainability-related issues" in the code that decrease code readability and code modifiability. It is important to note that in the category "code smells", SonarQube actually includes some of the code smells proposed by Fowler et al. [41].

SonarQube LTS 6.7.7 detects a total of 413 rules which are grouped based on type and severity. SonarQube categorizes the 413 rules under 3 types: *Bugs*, *Code Smells*, and *Vulnerabilities.* Of these 413 rules, 107 rules are classified as *Bugs*, 272 as *Code Smells*, and 34 as *Vulnerabilities.* In addition to the types, SonarQube groups the rules under 5 severity typers: *Blocker*, *Critical*, *Major*, *Minor*, and *Info.* Considering the assigned severity levels, SonarQube detects 36 *Blocker*, 61 *Critical*, 170 *Major*, 141 *Minor*, and 5 *Info* rules. Unlike Better Code Hub and Coverity Scan, SonarQube has only one severity and classification type assigned to each rule.

4. Empirical Study Design

We designed our empirical study according to the guidelines defined by Wohlin [42]. The following section describes the goals and specific research questions driving our empirical study as well as the data collection and analysis procedures.

4.1. Goal and Research Questions

The *goal* of our empirical study is to compare state-of-the-practice SATs *with the aim of* assessing their capabilities when detecting source code quality issues with respect to (1) the types of problems they can actually identify; (2) the agreement among them, and (3) their precision. Our ultimate *purpose* is to enlarge the knowledge

[25] `http://findbugs.sourceforge.net/factSheet.html`
[26] `http://findbugs.sourceforge.net/bugDescriptions.html`
[27] `https://pmd.github.io/latest/`
[28] `https://github.com/pmd/pmd/tree/master/pmd-java/src/main/resources/rulesets/java`
[29] `https://pmd.github.io/latest/index.html`
[30] `https://pmd.github.io/latest/pmd$_$rules$_$java.html`
[31] `https://pmd.github.io/latest/`
[32] `http://www.sonarsource.org/`

available on the identification of source code quality issues with SATs from the *perspective* of both researchers and tool vendors. The former are interested in identifying areas where the state-of-the-art tools can be improved, thus setting up future research directions, while the latter are instead concerned with assessing their current capabilities and possibly the limitations that should be addressed in the future to better support developers.

It is important to remark that the goal of the empirical study is to show and compare the capabilities of existing, widely used static analysis tools *independently* from the types of analyses they perform while detecting potential concerns in source code. Indeed, we are interested in benchmarking the most popular static analysis tools with respect to their practical support provided by developers, i.e., with respect to the issues they are able to uncover, the potential gain provided by their combination, and their overall precision. Our results are meant to inform practitioners on how to benefit more from the capabilities of existing static analysis tools, other than alerting them on the potential drawbacks of blindly relying on these tools. In addition, our findings might raise limitations that researchers and tool vendors should consider to provide better tools. More specifically, our goal can be structured around three main research questions (**RQ**$_s$). As a first step, we conducted a preliminary investigation aiming at posing the basis for the additional analyses. The goal of our first research question is to (i) analyze the features the various tools make available, and (ii) determine how many issues can be detected on the selected dataset—this is important to understand whether the selected dataset is actually useful to address the other research questions.

RQ$_1$**.** *How do the considered static analysis tools compare in terms of functionalities and distribution of source code quality issues?*

Once we had characterized the tools with respect to what they are able to identify, we proceeded with a finer-grained investigation aimed at measuring the extent to which SATs agree with each other. Regarding this aspect, further investigation would not only benefit tool vendors who want to better understand the capabilities of their tools compared to others, but would also benefit practitioners who would like to know whether it is worth using multiple tools within their code base. Moreover, we were interested in how the issues from different tools overlap with each other. We wanted to determine the type and number of overlapping issues, but also whether the overlapping is between all tools or just a subset.

RQ$_2$**.** *What is the agreement among different Static Analysis Tools when detecting source code quality issues?*

Finally, we focused on investigating the potential accuracy of the tools in practice. While they could output numerous rules that alert developers of the presence of potential quality problems, it is still possible that some of these rules might represent false positive instances, i.e., that they wrongly recommend source code entities to be refactored/investigated. Previous studies have highlighted the presence of false positives as one of the main problems of the tools currently available [10]; our study aims at corroborating and extending the available findings, as further remarked in Section 4.4.

RQ$_3$**.** *What is the precision of Static Analysis Tools?*

All in all, our goal was to provide an updated view on this matter and understand whether, and to what extent, this problem has been mitigated in recent years or whether there is still room for improvement.

4.2. Context of the Study

We selected projects from the Qualitas Corpus collection of software systems (Release 20130901), using the compiled version of the Qualitas Corpus [43]. There are two main reasons leading to the selection of the Qualitas Corpus dataset. First, it provides compiled versions of software systems. This is a key requirement in our case, as most of the static analysis tools considered in our study require compiled code to be executed - the build phase would have been extremely time-consuming and error-prone, should have we relied on different datasets [44]. In addition, the Qualitas Corpus dataset allowed us to perform analyses on a publicly available and well-established source, which has been often used as a benchmark for software quality studies [45, 46, 47]. As such, we preferred to conduct the study on this dataset to provide researchers with insights and findings that can be challenged by other researchers by using the same dataset.

The dataset contains 112 Java systems with 754 versions, more than 18 million LOCs, 16,000 packages, and 200,000 classes. Moreover, the dataset includes projects from different contexts such as IDEs, databases, and programming language compilers. More information is available in [43]. In our study, we considered the recent (“r”) release of each of the 112 available systems. Since two of SATs considered, i.e., Coverity Scan and Better Code Hub, require permissions in the GitHub project or the upload of a configuration file, we privately uploaded all 112 projects to our GitHub account in order to enable the analysis[33].

4.3. Data Collection

This section describes the data collection from each tool and the data collection process. We analyzed a single

[33]The GitHub projects, with the related configuration adopted for executing the tools, will be made public in the case of acceptance of this paper.

snapshot of each project, considering the release available in the dataset for each of 112 systems.

SonarQube. We first installed SonarQube LTS 6.7.7 on a private server having 128 GB RAM and 4 processors. However, because of the limitations of the open-source version of SonarQube, we are allowed to use only one core, therefore more cores would have not been beneficial for our scope. We decided to adopt the LTS version (Long-Time Support) since this is the most stable and best-supported version of the tool.

Coverity Scan. The projects were registered in Coverity Scan (version 2017.07) by linking the GitHub account and adding all the projects to the profile. Coverity Scan was set up by downloading the tarball file from `https://scan.coverity.com/download` and adding the `bin` directory of the installation folder to the path in the `.bash_profile`. Afterwards the building process began, which was dependent on the type of project in question. Coverity Scan requires to compile the sources with a special command. Therefore, we had to compile them, instead of using the original binaries.

Better Code Hub. The `.bettercodehub.yml` files were configured by defining the `component_depth`, `languages`, and `exclusions`. The `exclusions` were defined so that they would exclude all directories that were not source code, since Better Code Hub only analyzes source code.

Checkstyle. The `JAR` file for the Checkstyle analysis was downloaded directly from Checkstyle's website[34] in order to engage the analysis from the command line. The executable `JAR` file used in this case was `checkstyle-8.30-all.jar`. In addition to downloading the `JAR` executable, Checkstlye offers two different types of rule sets for the analysis[34].

FindBugs. FindBugs 3.0.1 was installed by running `brew install findbugs` in the command line. Once installed, the GUI was then engaged by writing `spotbugs`. From the GUI, the analysis was executed through `File` → `New Project`. The classpath for the analysis was identified to be the location of the project directory.

PMD. PMD 6.23.0 was downloaded from GitHub[35] as a zip file. After unzipping, the analysis was engaged by identifying several parameters: project directory, export file format, rule set, and export file name.

More details about the how install and ran the tools are available in our replication package[36].

4.4. Data Analysis

In this section, we describe the analysis methods employed to address our research questions (**RQs**).

[34]`https://checkstyle.org/#Download`

[35]`https://github.com/pmd/pmd/releases/download/pmd$_$releases\%2F6.23.0/pmd-bin-6.23.0.zip`

Source code quality issues identified by the tools (RQ$_1$). To address this RQ, we first provide an overview of the features and the issues detected by the tools. To this aim, we consulted the tools' documentation in order to extract their main features. Specifically, we collected the list of programming languages supported by each tool and the typologies of issues they cover. As for the former, we simply reported programming language support according to the documentation. For the latter case, instead, we had to perform a qualitative analysis process [48] by applying descriptive and pattern coding [49]. First, the documentation of each tool was analyzed in order to extract the categories of issues covered. Then, two of the authors performed a standardization of the categories to ensure consistency and completeness of the coding process. Finally, the categories referring to the same or similar concept were grouped together.

Once having provided an overview of the tools features, in order to determine the tool detection capabilities overlaps, we also calculated how many issues are generated by the rules violated in the reference dataset.

In the reminder of the paper, we will refer to all the categories and types of rules and analysis checks performed by SATs as "rule" and to the instances of rules violated in the code as "issue".

Agreement among the tools (RQ$_2$). In this research question the goal is to inspect whether the tools are similar in terms of the issues they detect. The assumption is similar tools to have similar rules and, further, we expect similar rules to affect the same classes, and the same lines of the code.

Tool similarity. Similar tools were identified by comparing all six tools with each other, meaning $\binom{6}{2} = 15$ tool pair comparisons. To determine the similarity of a tool pair t_1 and t_2, we calculated the percentage of detected issues that in both tools highlighted the same position in the source code. The agreement value for tools t_1 and t_2 is defined below:

$$\textit{tool agreement}(t_1, t_2) = \frac{\#\text{issues in the same position from } t_1 \text{ and } t_2}{\min\left(\#\text{issues from } t_1, \#\text{issues from } t_2\right)} \quad (1)$$

The numerator is the number of times issues from t_1 and t_2 were detected in the same code position. The maximum of this number achieved when all issues generated from one tool are always present with an issue from the other tool, which is possible only for the tool that generated less issues. Therefore, the formula uses minimum in the denominator. In the equation, #issues for t is the total number of issues the tool has identified, therefore, it is the sum of issues generated by all rules in tool t.

The overlap of the detected issues was determined by analyzing all possible rule pairs from each tool pair. For each rule pair (r_m, r_n), we iterated through all detected instances of r_m and checked whether an instance of r_n was affecting the same position. For calculating the tool similarity, this data was aggregated to the tool level. For example, in the data set used in this paper, SonarQube had 275 rules and which generated issues while PMD had 180, resulting in $275 \cdot 180 = 49,500$ possible rule pairs from that tool pair. In total, we analyzed 339,169 rule pairs as similar comparison was made for each tool pair (Table 5).

Rule similarity. In addition to tool similarity, we investigated the similarity of the individual rules from the tools. The similarity of two rules is determined using the percentage of the instances of a rule occurring together with another rule. The agreement value for rule r_m when compared to rule r_n is defined as below:

$$rule\ agreement(r_m, r_n) = \frac{\#\text{times } r_m \text{ and } r_n \text{ violated in the same position}}{\#\text{total violations of } r_m} \quad (2)$$

The data for the numerator was obtained during the data analysis for the tool agreement, and it has been described there.

Note, that the value is calculated separately for both rules in a rule pair, i.e., it is possible that $rule\,agreement_{(r_1,r_2)} \neq rule\,agreement_{(r_2,r_1)}$. Therefore, the perfect overlap is obtained if the agreement for both rules is equal to one. This means that all the issues generated by s and r are always detected in the same classes or position, and no issues are detected separately in a different class. Such measure was used as the granularity of the rules differs greatly between the tools. For example, BCH has only few wide rules like "write clean code" where as SonarQube has several smaller rules falling under that rule. The lower granularity rules might always exist with the wider rule as they both are meant to detect the smaller issues but the wider rule exists also without the smaller rule as it catches other issues as well.

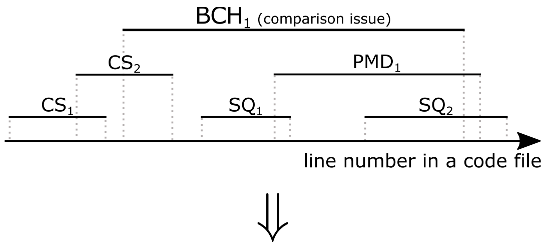
Naturally, most of these rule pairs are not even meant to be similar, they might never occur together or they could be defined for a different level of granularity. However, we decided to inspect all possible rule pairs to make the comparison as objective as possible. This would have not been true, had we manually selected the inspected rule pairs based on their description. We believe similar rules should be found also by comparing all rules as similar rules should consistently highlight the same positions of code.

Granularity of the comparisons. In both *rule agreement* and *tool agreement* we have used the term "in the same position" in their definition. In this paper, the comparisons were performed on two granularity levels: class level and line level. On *class level* the requirement was for the issues to be found in the same class, regardless where in the class the issues were located. Practically, we checked for each issue from one tool whether the other tool had a issue in the same class. However, on *line level*, in addition the issues being in the same class, the lines affected by the issues had to overlap. A third granularity option would have been method level but, as several of the tools did not report it, it was not used in the paper.

As the granularity of the rules varies between the tools, we checked what fraction of the affected lines are overlapping between the issues, instead of requiring sufficient overlap between both issues. To quantify the degree of overlapping, we used the percentage of the lines inside the comparison range. The results were grouped based on four percentage thresholds: 100%, 90%, 80%, and 70%.

The concept is illustrated in Figure 1. The lines represent issues in a code file, indicating the start and end of the affected code lines. issue BCH_1 is the comparison issue to which other issues are compared to. Depending on the used threshold, different issues are selected based on the overlapping percentage, as shown in the table associated with the figure. For example, SQ_1 would be always considered as a similar issue as it is completely "inside" BCH_1. 90% of PMD_1 is within the comparison issue BCH_1 and, therefore, if the selected threshold is 90% or less it is listed as a similar issue, otherwise it is discarded. However, as less than 70% of CS_1 and CS_2 are overlapping with the comparison issue, they will not be listed as a similar issue regardless of the threshold.

As a final step, we manually inspected all the rules with agreement 100 % to verify if they were related to the same type of problem. For this purpose, two authors having high expertise with SAT tools manually checked all the 100 % matching rules applying open coding. In the case of disagreements, a third author helped to solve the inconsistencies. The whole process lasted around three hours.



Threshold	100 %	90 %	80 %	70 %
Comparison issue	BCH_1			
Issues overlapping with the comparison issue	SQ_1	SQ_1 PMD_1	SQ_1 PMD_1	SQ_1 PMD_1 SQ_2

Figure 1: Determining issues in the same position as BCH_1 at "line-level" for thresholds 100%, 90 %, 80%, and 70%. (RQ$_2$)

Precision of the tools (RQ$_3$). In our last research question, we aimed at assessing the precision of the considered tools. Since previous work already assessed the accuracy of static analysis tools [10], with this analysis we aimed at corroborating or contrasting the findings previously achieved. In any case, it is worth remarking immediately that, in the context of our study, we did not consider the conventions used by the considered projects: as such, we could not know if certain issues output by the tools were actually relevant for developers or if they were considered meaningless. Since the individual conventions used by the projects are not always explicitly established or easy to mine, we can only estimate the false positive rate by looking at the precision of the tools, being aware of the potential bias of this analysis. Nonetheless, we still believe that an analysis of this type might be useful to researchers and tool vendors to understand the extent to which the issues output by tools can be considered reliable.st

From a theoretical point of view, precision is defined as the ratio between the true positive issues identified by a tool and the total number of issues it detects, i.e., true positives plus false positive items (TPs + FPs). Formally, for each tool we computed precision as follows:

$$precision = \frac{TPs}{TPs + FPs} \tag{3}$$

It is worth remarking that our focus on precision is driven by recent findings in the field that showed that the presence of false positives is among the most critical barriers to the adoption of static analysis tools in practice [10, 3]. Hence, our analysis provides research community, practitioners, and tool vendors with indications on the actual precision of the currently available tools—and aims at possibly highlighting limitations that can be addressed by further studies. It is also important to remark that we do not assess recall, i.e., the number of true positive items identified over the total number of issues present in a software project, because of the lack of a comprehensive ground truth. We plan to create a similar dataset and perform such an additional evaluation as part of our future research agenda.

When assessing precision, a crucial detail is related to the computation of the set of true positive issues identified by each tool. In the context of our work, we conducted a manual analysis of the rules highlighted by the six considered tools, thus marking each of them as true or false positive based on our analysis of (1) the issue identified and (2) the source code of the system where the issue was detected. Given the expensive amount of work required for a manual inspection, we could not consider all the rules output by each tool, but rather focused on statistically significant samples. Specifically, we took into account a 95% statistically significant stratified sample with a 5% confidence interval [50] of the 65,133, 8,828, 62,293, 402,409, 33,704, and 467,583 items given by Better Code Hub, Coverity Scan, SonarQube, Checkstyle, FindBugs, and PMD respectively: This step led to the selection of a set of 375 items from Better Code Hub, 367 from Coverity Scan, 384 from SonarQube, 384 from Checkstyle, 379 from FindBugs, and 380 from PMD.

Using stratified sampling, the components of the target samples are separated into distinct groups or strata, and within each stratum, they are similar to one another and respect key relevant criteria of the initial sample (including weights, in our case) [51]. Therefore, the selection of a stratified sample already resolves the problem of having different weights. Indeed, this strategy intrinsically ensures that the weights in the selected sample are consistent with the ones in the original superset.

To increase the reliability of this manual analysis, two of the authors of this paper having a high expertise with

Table 2: The selected samples to compute the Precision of the six static analysis tools (RQ_3)

SATs	# samples
Better Code Hub	375
Checkstyle	384
Coverity Scan	367
FindBugs	379
PMD	380
SonarQube	384

SAT tools (henceforth called the inspectors) first independently analyzed the rule samples. They were provided with a spreadsheet containing six columns: (1) the name of the static analysis tool the row refers to, i.e., Better Code Hub, Coverity Scan, Sonarqube, Checkstyle, FindBugs, and PMD; (2) the full path of the rule identified by the tool that the inspectors had to verify manually; and (3) the rule type and specification, e.g., the code smell. The inspectors' task was to go over each of the rules and add a seventh column in the spreadsheet that indicated whether the rule was a true or a false positive. During this analysis, the inspectors had to first understand the context of the issue, namely the class or the project of interest. As such, before analyzing a issue, they inspected the project with the aim of having a general understanding of the functionalities implemented. Afterwards, they analyzed the specific class affected by a potential problem raised by the issue and analyzed the content of the class and, if needed, the content of the other classes called by the class. This last step, namely the analysis of the class dependencies, was required in just five cases; in the others, the inspectors were able to assess the validity of a issue just by looking at the source code of the affected class. Both the inspectors were able to complete the entire process in one week.

After this analysis, the two inspectors had a four-hour meeting where they discussed their work and resolved any disagreements: All the items marked as true or false positive by both inspectors were considered as actual true or false positives; in the case of a disagreement, the inspectors re-analyzed the rule in order to provide a common assessment. Overall, after the first phase of inspection, the inspectors reached an agreement of 0.84—which we computed using Krippendorff's alpha Kr_α [52] and which is higher than 0.80, which has been reported to be the standard reference score for Kr_α [53]. Overall, the inspectors individually spent, approximately, 160 person/hours.

In Section 5.3, we report the precision values obtained for each of the considered tools and discuss some qualitative examples that emerged from the manual analysis of the sample dataset.

4.5. Replicability

In order to allow the replication of our study, we have published the raw data in a replication package[36].

[36] https://figshare.com/s/5df8c271baa0368cd695

Table 3: Support for the top 15 programming languages according to the PYPL index. Results for all the programming languages supported by the tools are reported in Appendix A.

Tool	Python	Java	Javascript	C#	C/C++	PHP	R	TypeScript	Objective C	Swift	Go	Matlab	Kotlin	Rust	VBA
Better Code Hub	x	x	x	x	x			x	x	x	x		x		
Checkstyle		x													
Coverity	x	x	x	x	x				x	x	x		x		
FindBugs		x													
PMD		x	x												
SonarQube	x	x	x	x	x	x		x	x	x	x		x		

Table 4: Typologies of issues spotted by the tools.

Tool	Syntax	Bugs	Security	Design	Bad practices
Better Code Hub				x	x
Checkstyle	x			x	
Coverity	x		x	x	x
FindBugs		x		x	x
PMD	x	x	x	x	x
SonarQube		x	x	x	x

5. Analysis of the Results

In this section, we report and discuss the results obtained when addressing our research questions (RQ$_s$).

5.1. Static Analysis Tools detected issues (RQ$_1$)

Table 3 reports information about the tools' support for the top 15 programming languages, according to PYPL classification[37]. The complete results, including all the programming languages supported by the six tools are reported in our Appendix A. As we can observe, the most popular programming language is JAVA, which is the only one supported by all the considered tools—this is one of the reasons for the dataset selection in Section 4.2.

Other languages having high support by the selected tools are JAVASCRIPT, C/C#/C++, OBJECTIVE C, SWIFT, GO, and KOTLIN, that are supported by Better Code Hub, Coverity, and SonarQube (and also PMD for Javascript).

Turning to the tools perspective, we can observe that SonarQube is the one providing the best support for the top 15 programming languages, covering 11 of them. Almost the same set of languages are also supported by Better Code Hub and Coverity which support respectively 10 and 9 out of the top 15 programming languages.

The remaining three tools only focus on a very narrow set of programming languages. Specifically, Checkstyle and Findbugs only support JAVA source code while PMD provides support only for JAVA and JAVASCRIPT.

Table 4 shows the categories of issues covered by the subject tools. As a result of the coding process described in Section 4.4, we came out with 5 categories, namely SYNTAX, BUGS, SECURITY, DESIGN, BAD PRACTICES.

Results of our classification show that the selected tools mainly deal with design concerns. They all capture design issues and bad practices (excluding Checkstyle for the latter). Some of the tools also check for other characteristics such as syntactic violations, and indicators of bugs/security flaws.

As an overall observation, we can conclude that the selected tools are mainly focused on the same characteristics, hence justifying the empirical comparison subject of our next two research questions.

We obtained results only for 47 projects out of the 112 contained in the Qualitas Corpus dataset, applying the rules defined by Better Code Hub, Checkstyle, Coverity Scan, FindBugs, PMD, and SonarQube. Unfortunately, the used versions of Better Code Hub and Coverity Scan were not able to analyze all the dataset. So, we considered only the projects analyzed by all the six tools.

Table 5: Issues detected by the six SATs in the 47 projects (RQ$_1$)

Tool	Detected rule			
	# rule	# occurrences	# type	# severity
Better Code Hub	8	27,888	3	3
Checkstyle	88	9,686,813	12	2
Coverity	130	7,431	26	3
FindBugs	255	33,704	9	3
PMD	275	3,380,493	7	5
SonarQube	180	418,433	3	5

In total, the projects were infected by **936** rules violated **13,554,762 times** (number of total issues). 8 (out of 10) rules were detected by Better Code Hub 27,888 times, 88 (out of 173) rules were detected by Checkstyle 9,686,813 times, 130 rules were detected by Coverity Scan 7,431 times, 255 (out of 424) rules were detected by FindBugs 33,704 time, 275 (out of 305) rules were detected by PMD 3,380,493 times, and 180 (out of 413) rules were detected by SonarQube 418.433 times (Table 6 and Table 5). It is important to note that in Table 5, the detection capability is empty for Coverity. As mentioned earlier, the full detection capability is only provided to clients and not on the public API. We also computed how often rules were violated by grouping them based on type and severity. The full results of this additional analysis are reported in our replication package[36].

Given the scope of rules that were detected, our projects were affected by all rules that are detectable by Better Code Hub and by some rules that are detectable by Coverity Scan and SonarQube (Table 7). For the sake of readability, we report only the Top-10 rules detected in our projects by the six tools. The complete list is available in the replication package[36].

[37]`https://pypl.github.io/PYPL.html`

Table 6: Rule diffusion across the 47 projects (RQ_1)

Project Name	#Classes	#Methods	SQ	BCH	Coverity	Checkstyle	PMD	FindBugs	Total
AOI	865	2568	10865	924	123	250201	108458	1979	372550
Collections	646	2019	4545	584	25	85501	39893	185	130733
Colt	627	1482	7452	560	62	172034	47843	4	227955
Columba	1288	2941	7030	662	70	166062	49068	1345	224237
DisplayTag	337	683	853	452	22	32033	10137	32	43529
Drawswf	1031	1079	3493	559	65	368052	22264	69	394502
Emma	509	962	4451	648	55	68838	16524	172	90688
FindBugs	1396	4691	12496	600	134	320087	90309	1068	424694
Freecol	1569	4857	5963	607	337	127363	79588	704	214562
Freemind	1773	3460	5698	662	112	128590	50873	1536	187471
Ganttproject	1093	2404	12349	642	64	71872	36689	898	122514
Hadoop	3880	10701	24125	682	665	284315	228966	1547	540300
HSQLDB	1284	5459	14139	620	178	192010	109625	182	316754
Htmlunit	3767	9061	5176	924	141	92998	59807	467	159513
Informa	260	644	992	594	56	11276	9364	217	22499
Jag	1234	1926	6091	301	56	24643	19818	408	51317
James	4138	2197	6091	656	82	336107	29253	25	372214
Jasperreports	2380	4699	17575	702	226	643076	96000	1420	758999
Javacc	269	689	3693	504	29	24936	17784	39	46985
JBoss	7650	18239	42190	415	51	1084739	377357	1158	1505910
JEdit	2410	4918	15464	630	134	434183	93605	74	544090
JExt	2798	2804	7185	585	339	276503	42693	125	327430
JFreechart	1152	3534	6708	660	88	154064	89284	849	251653
JGraph	314	1350	2577	666	128	98119	22516	41	124047
JGgraphPad	433	916	2550	599	10	62230	18777	75	84241
JGgraphT	330	696	922	562	35	23808	11147	15	36489
JGroups	1370	4029	14497	602	391	265886	89601	1560	372537
JMoney	183	455	575	426	52	15377	5639	118	22187
Jpf	143	443	522	558	21	14736	7054	20	22911
JRefactory	4210	5132	18165	580	129	207911	116452	2633	345870
Log4J	674	1028	2042	625	125	40206	15463	162	58623
Lucene	4454	10332	11332	707	85	627683	233379	585	873771
Marauroa	266	777	1228	547	33	53616	10681	148	66253
Maven	1730	4455	3110	642	121	225017	46620	1242	276752
Megamek	3225	8754	14974	600	321	346070	174680	3430	540075
Myfaces_core	1922	5097	22247	312	121	619072	174790	790	817332
Nekohtml	82	269	623	460	13	12987	3979	56	18118
PMD	1263	3116	8818	616	50	109519	47664	543	167210
POI	2276	8648	19463	903	771	476488	162045	792	660462
Proguard	1043	1815	3203	646	23	115466	37221	6	156565
Quilt	394	638	1075	386	13	16840	7488	170	25972
Sablecc	251	886	4385	520	25	30840	19756	101	55627
Struts	2598	6719	8878	616	57	231912	106513	253	348229
Sunflow	227	670	1549	478	46	32251	20937	63	55324
Trove	421	477	454	216	78	15507	5430	416	22101
Weka	2147	6286	32258	604	1437	365535	195774	4118	599726
Xalan	2174	4758	18362	844	232	330254	121685	1864	473241
Total	**74,486**	**169,763**	**418,433**	**27,888**	**7,431**	**9,686,813**	**3,380,493**	**33,704**	**13,554,762**

***Finding 1**. The selected tools share the same main features. The amount of rules detected by the six static analysis tools is significant (**936** rules detected **13,554,762 times**); hence, we can proceed with the analysis of the remaining RQ_s.*

5.2. Static Analysis Tools Agreement (RQ_2)

Our second research question focused on the analysis of the agreement between the SATs.

Agreement based on the overlapping at "class-level". In order to include Coverity Scan in this analysis, we first evaluated the detection agreement at "class-level", considering each class where the rule detected by the other five tools overlapped at 100% and where at least one rule of Coverity Scan was violated in the same class.

To calculate *Tool similarity* (column "%", Table 8), we checked the occurrences of the rules of both tools in our projects, then we considered only the minimum value. For example, in Table 8, calculating the percentage between Checkstyle - PMD rule pairs, we have 9,686,813 rules Checkstyle detected and 3,380,493 PMD ones detected. The combination of these rule should be maximum 3,380,493 (the minimum value between the two). We calculated the percentage considering the column "# occurrences" and the column "# possible occurrences".

The *Tool similarity* on the "class-level" is always low, as reported in Table 8. This means that a piece of code violated by a rule detected by one tool is almost never violated by another rule detected by another tool. In the best case (Table 8), only 9.378% of the possible rule (FindBugs-PMD). Moreover, we did not investigate *Tool similarity* at "class-level" considering more that two tools (e.g. Checkstyle-FindBugs-PMD).

Table 7: The top-10 issues detected by Better Code Hub, Checkstyle, Coverity Scan, FindBugs, PMD, and SonarQube (RQ_1)

Id	**Better Code Hub Detected Rule**	#
	WRITE_CLEAN_CODE	16,055
	WRITE_CODE_ONCE	14,692
	WRITE_SHORT_UNITS	6,510
	AUTOMATE_TESTS	6,475
	WRITE_SIMPLE_UNITS	6,362
	SMALL_UNIT_INTERFACES	6,352
	SEPARATE_CONCERNS_IN_MODULES	5,880
	COUPLE_ARCHITECTURE_COMPONENTS_LOOSELY	2,807
Id	**Checkstyle Detected Rule**	#
	IndentationCheck	3,997,581
	FileTabCharacterCheck	2,406,876
	WhitespaceAroundCheck	865,339
	LeftCurlyCheck	757,512
	LineLengthCheck	703,429
	RegexpSinglelineCheck	590,020
	FinalParametersCheck	406,331
	ParenPadCheck	333,007
	NeedBracesCheck	245,110
	MagicNumberCheck	223,398
Id	**Coverty Scan Detected Rule**	#
	Dereference null return value	1,360
	Dm: Dubious method used	689
	Unguarded read	556
	Explicit null dereferenced	514
	Resource leak on an exceptional path	494
	Dereference after null check	334
	Resource leak	301
	DLS: Dead local store	293
	Missing call to superclass	242
	Se: Incorrect definition of serializable class	224
Id	**FindBugs Detected Rule**	#
	BC_UNCONFIRMED_CAST	2,840
	DM_NUMBER_CTOR	2,557
	BC_UNCONFIRMED_CAST_OF_RETURN_VALUE	2,424
	DM_DEFAULT_ENCODING	1,946
	RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	1,544
	DLS_DEAD_LOCAL_STORE	1,281
	DM_FP_NUMBER_CTOR	959
	SE_NO_SERIALVERSIONID	944
	REC_CATCH_EXCEPTION	887
	SE_BAD_FIELD	878
Id	**PMD Detected Rule**	#
	LawOfDemeter	505,947
	MethodArgumentCouldBeFinal	374,159
	CommentRequired	368,177
	LocalVariableCouldBeFinal	341,240
	CommentSize	153,464
	DataflowAnomalyAnalysis	152,681
	ShortVariable	136,162
	UselessParentheses	128,682
	BeanMembersShouldSerialize	111,400
	ControlStatementBraces	110,241
Id	**SonarQube Detected Rule**	#
S1213	The members of an interface declaration or class should appear in a pre-defined order	30,888
S125	Sections of code should not be ”commented out”	30,336
S00122	Statements should be on separate lines	26,072
S00116	Field names should comply with a naming convention	25,449
S00117	Local variable and method parameter names should comply with a naming convention	23,497
S1166	Exception handlers should preserve the original exceptions	21,150
S106	Standard outputs should not be used directly to log anything	19,713
S1192	String literals should not be duplicated	19,508
S134	Control flow statements ”if”,”for”,”while”,”switch” and ”try” should not be nested too deeply	17,654
S1132	Strings literals should be placed on the left side when checking for equality	13,576

Table 8: Issue pairs that overlap at the "class-level" (RQ$_2$)

issue pairs	# occurrences	# possible occurrences	%
Checkstyle - PMD	4,872	3,380,493	0.144
SonarQube - PMD	4,126	418,433	0.98
FindBugs - PMD	3,161	33,704	9.378
SonarQube - Checkstyle	1,495	418,433	0.357
FindBugs - Checkstyle	1,265	33,704	3.753
BCH-PMD	1,017	27,888	3.646
SonarQube - FindBugs	849	33,704	2.518
BCH-SonarQube	517	27,888	1.853
BCH-Checkstyle	440	27,888	1.577
BCH-FindBugs	235	27,888	0.842
Coverity - BCH	117	7,431	1.574
Coverity - Checkstyle	128	7,431	1.723
Coverity - FindBugs	120	7,431	1.615
Coverity - PMD	128	7,431	1.723
Coverity - SonarQube	128	7,431	1.723
Total	**18,598**	**4,457,178**	**0.417**

Table 9: Issue pairs that overlap at the 100% threshold considering the "line-level" (RQ$_2$)

issue pairs	# occurrences	# possible occurrences	%
Checkstyle - PMD	4,872	3,380,493	0.144
SonarQube - PMD	4,126	418,433	0.98
FindBugs - PMD	3,161	33,704	9.378
SonarQube - Checkstyle	1,495	418,433	0.357
FindBugs - Checkstyle	1,265	33,704	3.753
BCH-PMD	1,017	27,888	3.646
SonarQube - FindBugs	849	33,704	2.518
BCH-SonarQube	517	27,888	1.853
BCH-CheckSyle	440	27,888	1.577
BCH-FindBugs	235	27,888	0.842
Total	**17,977**	**4,430,023**	**0.4%**

For each rule pair we computed the detection agreement at class level. For the sake of readability, we report these results in Appendix A. Specifically, the three tables (Table .13, Table .14, and Table .15) overview the detection agreement of each rule pair, according to the procedure described in Section 4.4. As further explained in the appendix, for reasoning of space we only showed the 10 most recurrent pairs, putting the full set of results in our replication package [36]. In these tables, the third and fourth columns (eg. "#BCH pairs" and "# CHS pairs", Table .13) report how many times a rule instance from a tool exists with another one. The remaining two columns report the agreement of each tool considered in the rule pairs (eg. "#BCH Agr." and "# CHS Agr.", Table .13). Results showed for all the rule pairs that the agreement at "class-level" is very low, as none of the most recurrent rule pairs agree well. The results also highlighted the difference in the granularity of the rules.

Agreement based on the overlapping at the "line-level". Since we cannot compare at "line-level" the rules detected by Coverity Scan, we could only consider the remaining five SATs. Several rule pairs were found according to the 100%, 90%, 80%, and 70% thresholds (Figure 1). Using the threshold of **100%** which indicates that a rule completely resides within the comparison rule, we found **17,977 rule pairs**, as reported in Table 9. Using the thresholds of 90%, 80%, and 70% the following rule pairs were found respectively: 17,985, 18,004, and 18,025 (Table 10). These rules resided partially within the reference rule.

Similarly to what happened with the agreement at "class-level", it is important to note that the overlap at the "line-level" is always low. Results show that, also in this case, only 9.378% of the possible rule occurrences are detected in the same line by the same two tools (FindBugs and PMD). In addition, also in this case we did find no pair rules at "line-level" considering more that two tools (e.g. Checkstyle-FindBugs-PMD).

When considering the agreement for each rule pair at "line-level', we could not obtain any result because of computational reasons. Indeed, the analysis at line-level of 936 rule types that have been violated 13,554,762 times would have required a prohibitively expensive amount of time/space—according to our estimations, it would have been taken up to 1.5 years—and, therefore, we preferred excluding it.

Manual validation of the agreement based on the overlapping at "class-level". The manual inspection of the 66 rules with 100% agreement resulted into six couples of rules from Checkstyle and PMD. It is interesting to note that for all the other tools, rules with 100% agreement were referred to totally different concepts. As an example, none of the rules agreeing with 100% between BCH and Coverity Scan were considered to be related to the same quality issue. An example of a rule pair is BCH "automate_tests" that matches with Coverity Scan rule "dead default in swich". Table 11 reports the rules that resulted to be related to the same quality issue.

Finding 2. *The detection agreement among the different tools is very low (less than 0.4%). The rule pairs Checkstyle - PMD as the lowest overlap (0.144%) and FindBugs - PMD the highest one (9.378%). Consequently also the detection agreement is very low.*

Finding 3. *Among the 66 rules with 100% detection agreement, only six are related to the same quality issues.*

5.3. Static Analysis Tools precision (RQ$_3$)

In the context of our last research question, we focused on the precision of the SATs when employed for potential quality rules detection. Table 12 reports the results of our manual analyses. As shown, the precision of most tools is quite low, e.g., SonarQube has a precision of 18%, with the only exception of CheckStyle whose precision is equal to 86%.

Table 10: Issue pairs that overlap at the different thresholds, i.e., 90/80/70%, considering the "line-level" (RQ$_2$)

issue pairs	# occurrences			
	90%	**80%**	**70%**	**possible**
	#(%)	#(%)	#(%)	
Checkstyle - PMD	4,872 (0.144)	4,874 (0.144)	4,876 (0.144)	3,380,493
SonarQube - PMD	4,126 (0.986)	4,130 (0.987)	4,139 (0.989)	418,433
FindBugs - PMD	3,167 (9.39)	3,173 (9.41)	3,173 (9.41)	33,704
SonarQube - Checkstyle	1,495 (0.357)	1,496 (0.357)	1,496 (0.357)	418,433
FindBugs - Checkstyle	1,265 (3.753)	1,265 (3.753)	1,265 (3.753)	33,704
BCH-PMD	1,017 (3.646)	1,019 (3.646)	1,024 (3.647)	27,888
SonarQube - FindBugs	849 (2.519)	849 (2.519)	849 (2.519)	33,704
BCH-SonarQube	517 (1.853)	521 (1.868)	522 (1.868)	27,888
BCH-CheckSyle	440 (1.577)	440 (1.577)	441 (1.578)	27,888
BCH-FindBugs	237 (0.849)	237 (0.849)	240 (0.860)	27,888
Total	**18,004 (0.4)**	**18,004 (0.4)**	**18,025 (0.4)**	**4,430,023**

Table 11: Rules with 100% agreement related to similar quality issue (RQ$_2$)

Checkstyle	**PMD**	**Manually validation**
JavadocVariableCheck	CommentRequired	PMD Rule is more generic. Checkstyle refers to variables while PMD to all elements.
MissingJavadocMethodCheck	CommentRequired	PMD Rule is more generic. Checkstyle refers to methods while PMD to all elements.
StaticVariableNameCheck	VariableNamingConventions	PMD Rule is more generic. Checkstyle only refers to static variables. PMD to all variables.
NeedBracesCheck	ControlStatementBraces	Checkstyle rule is more generic. PMD only refers to control statements. Checkstyle to all blocks.
ParameterNameCheck	FormalParameterNamingConventions	PMD defines one rule for both local variables and formal parameters. Checkstyle and PMD define two separate rules.
EmptyStatementCheck	EmptyIfStmt / EmptyWhileStmt	PMD and Checkstyle define one rule for both if and while statements. PMD defines two separate rules.

Table 12: Precision of the considered SATs over the manually validated sample set of rules (RQ$_3$)

SAT	**# rules**	**# True Positives**	**Precision**
Better Code Hub	375	109	29%
Checkstyle	384	330	86%
Coverity Scan	367	136	37%
FindBugs	379	217	57%
PMD	384	199	52%
SonarQube	384	69	18%

In general, based on our findings, we can first corroborate previous findings in the field [7, 10, 8] and the observations reported by Johnson et al. [10], who found through semi-structured interviews with developers that the presence of false positives represents one of the main issues that developers face when using SATs in practice. With respect to the qualitative insights obtained by interviewing developers [10], our work concretely quantifies the capabilities of the considered SATs.

Looking deeper into the results, we could delineate some interesting discussion points. First, we found that for Better Code Hub and Coverity Scan almost two thirds of the recommendations represented false alarms, while the lowest performance was achieved by SonarQube. The poor precision of the tools is likely due to the high sensitivity of the rules adopted to search for potential issues in the source code, e.g., threshold values that are too low lead to the identification of false positive TD items. This is especially true in the case of SonarQube: In our dataset, it outputs an average of 47.4 violations per source code class, often detecting potential TD in the code too hastily.

A slightly different discussion is the one related to the other three SATs, namely PMD, FindBugs, and Checkstyle.

As for the former, we noticed that it typically fails when raising rules related to naming conventions. For instance, this is the case of the *'AbstractName'* rule: it suggests the developer that an abstract class should contain the term `Abstract` in the name. In our validation, we discovered that in several cases the recommendation was wrong because the contribution guidelines established by developers explicitly indicated alternative naming conventions. A similar problem was found when considering FindBugs. The precision of the tool is 57% and, hence, almost half of the rules were labeled as false positives. In this case, one of the most problematic cases was related to the *'BC_UNCONFIRMED_CAST'* rules: these are raised when a cast is unchecked and not all instances of the type casted from can be cast to the type it is being cast to. In most cases, these rules have been labeled as false positives because, despite casts were formally unchecked, they were still correct by design, i.e., the casts could not fail anyway because developers have implicitly ensured that all of them were correct.

Finally, Checkstyle was the SAT having the highest precision, i.e., 86%. When validating the instances output by the tool, we realized that the rules raised are related to pretty simple checks in source code that cannot be considered false positives, yet do not influence too much the functioning of the source code. To make the reasoning clearer, let consider the case of the *'IndentationCheck'* rule: as the name suggests, it is raised when the indentation of the code does not respect the standards of the project. In our sample, these rules were all true positives, hence contributing to the increase of the precision value. However, the implementation of these recommendations would improve the documentation of the source code but not dealing with possible defects or vulnerabilities. As such, we claim that the adoption of Checkstyle would be ideal when used in combination with additional SATs.

To broaden the scope of the discussion, the poor performance achieved by the considered tools reinforces the preliminary research efforts to devise approaches for the automatic/adaptive configuration of SATs [54, 55] as well as for the automatic derivation of proper thresholds to use when locating the presence of design issues in source code [56, 57]. It might indeed be possible that the integration of those approaches into the inner workings of the currently available SATs could lead to a reduction of the number of false positive. In addition, our findings also suggest that the current SATs should not limit themselves to the analysis of source code but, for instance, complementing it with additional resources like naming conventions actually in place in the target software system.

Finding 4. *Most of the considered SATs suffer from a high number of false positive rules, and their precision ranges between 18% and 57%. The only expection is Checkstyle (precision=86%), even though most of the rules it raises are related to documentation issues rather than functional problems and, as such, its adoption should be complemented with other SATs.*

6. Discussion and Implications

The results of our study provide a number of insights that can be used by researchers and tool vendors to improve SATs. Specifically, these are:

There is no silver bullet. According to the results obtained in our study, and specifically for $\mathbf{RQ}_2$, different SAT rules are able to cover different issues, and can therefore find different forms of source code quality problems: Hence, we can claim that *there is no silver bullet that is able to guarantee source code quality assessment on its own.* On the one hand, this finding highlights that practitioners interested in detecting quality issues in their source code might want to combine multiple SATs to find a larger variety of problems. On the other hand, and perhaps more importantly, our results suggest that the research community should have an interest in and be willing to devise more advanced algorithms and techniques that can improve the detection capabilities of currently available tools. As an example, we can envision a wider adoption of more complex static analysis methods, e.g., taint tracking and typestate, other than ensemble methods or meta-models [58, 59, 60, 61], that can (1) combine the results from different SATs and (2) account for possible overlaps among the rules of different SATs. This would allow the presentation of more complete reports about the code quality status of the software systems to their developers.

Learning to deal with false positives. One of the main findings of our study concerns with the low performance achieved by all SATs in terms of precision of the recommendations provided to developers ($\mathbf{RQ}_3$). Our findings represent the first attempt to concretely quantify the capabilities of the considered SATs in the field. Moreover, our study provides two practical implications: (1) It corroborates and triangulates the qualitative observations provided by Johnson et al. [10], hence confirming that the real accuracy of SATs is threatened by the presence of false positives; (2) it supports the need for more research on how to deal with false positives, and particularly on how to filter likely false alarms [62] and how to select/prioritize the rules to be presented to developers [**?** 37, 28]. While some preliminary research efforts on the matter have been made, we believe that more research should be devoted to these aspects. Finally, our findings may potentially suggest the need for further investigation into the effects of false positives in practice: For example, it may be worthwhile for researchers to study what the maximum number of false positive instances is that developers can deal with, e.g., they should devise a critical mass theory for false positive ASAT rules [63] in order to augment the design of existing tools and the way they present rules to developers.

Complementing static analysis tools. The findings from our $\mathbf{RQ}_2$ highlight that most of the issues reported by the static analysis tools are related to rather simple problems, like the writing of shorter units or the automation of software tests. These specific problems could possibly be avoided if current SATs tools would be complemented with effective tools targeting (1) automated refactoring and (2) automatic test case generation. In other words, our findings support and strongly reinforce the need for a joint research effort among the communities of source code quality improvement and testing, which are called to study possible synergies between them as well as to devise novel approaches and tools that could help practitioners complement the outcome provided by SATs with that of other refactoring and testing tools. For instance, with effective refactoring tools, the number of violations output by SATs would be notably reduced, possibly enabling practitioners to focus on the most serious issues. At the same time, the opposite is true as well. Running a static analyzer on large projects might lead to have too many potential issues to address and this would make the resolution of problems infeasible in practice. Yet, most projects still employ static analyzers in a continuous integration context [2] to adhere to the rules they care about. In this sense, static analyzers might be used to identify possible refactoring opportunities and guide developers toward the improvement of source code quality [64]. This point further reinforces the need for a joint research effort toward a better integration of static analysis tools and other software engineering methods.

7. Threats to Validity

A number of factors might have influenced the results reported in our study. This section discusses the main threats to validity and how we mitigated them.

Construct Validity. Threats in this category concern the relationship between theory and observation. A first aspect is related to the dataset used. In our work, we selected 112 projects from the Qualitas Corpus [43], which is one of the most reliable data sources in software engineering research [65]. We are aware of the fact that this dataset contains data collected in 2013, hence we could have missed some newly introduced constructs (e.g., lambda expressions). This is a limitation of the study, which we could not address because of time and computation constraints. Replications of our study on newer systems might address this limitation and highlight new insights on how static analysis tools work in practice.

Another possible threat relates to the configuration of the SATs employed. None of the considered projects had all the SATs configured and so we had to manually introduce them; in doing so, we adopted the default configuration of the tools. However, we are aware that different configurations given directly by the developers of the projects could affect the results.

Nevertheless, it is important to point out that this choice did not influence our analyses: indeed, we were interested in comparing the capabilities of existing tools independently from their practical usage in the considered systems. The problem of configuring the tools therefore does not change the answers to our research questions.

Internal Validity. As for potential confounding factors that may have influenced our findings, it is worth mentioning that some issues detected by SonarQube were duplicated: in particular, in some cases the tool reported the same issue violated in the same class multiple times. To mitigate this issue, we manually excluded those cases to avoid interpretation bias; we also went over the rules output by the other SATs employed to check for the presence of duplicates, but we did not find any. Another relevant threat concerns with the analysis of the precision of the static analysis tools ($\mathbf{RQ}_3$). By design, we did not analyze whether the considered projects established conventions or internal policies to ignore some of the violated rules raised by static analysis tools. This may have influenced our findings because the reported precision might not be consistent with the developer's perception of the accuracy of the tools. Nonetheless, mining conventions is not always possible, since some of them are not explicitly reported by developer. As such, the results of $\mathbf{RQ}_3$ might be an underestimation of the accuracy of the static analysis tools. We are aware of this limitation, yet we still believe that the reported results might be interesting for both developers and tool vendors, since they represent a lower-bound validity of the tools in practice.

External Validity. Threats in this category are concerned with the generalization of the results. While we cannot claim that our results fully represent every Java project, we considered a large set of projects with different characteristics, domains, sizes, and architectures. This makes us confident of the validity of our results in the field, yet replications conducted in other contexts would be desirable to corroborate the reported findings.

Another discussion point is related to our decision to focus only on open-source projects. In our case, this was a requirement: we needed to access the code base of the projects in order to configure the SATs. Nevertheless, open-source projects are comparable—in terms of source code quality—to closed-source or industrial applications [66]; hence, we are confident that we might have obtained similar results by analyzing different projects. Nevertheless, additional replications would provide further complementary insights and are, therefore, still desirable.

Finally, we limited ourselves to the analysis of Java projects, hence we cannot generalize our results to projects in different programming languages. Therefore, further replications would be useful to corroborate our results.

Conclusion Validity. With respect to the correctness of the conclusions reached in this study, this has mainly to do with the data analysis processes used. In the context of $\mathbf{RQ}_3$, we conducted iterative manual analyses in order to study the precision of the tools, respectively. While we cannot exclude possible imprecision, we mitigated this threat by involving more than one inspector in each phase, who first conducted independent evaluations that were later merged and discussed. Perhaps more importantly, we made all data used in the study publicly available with the aim of encouraging replicability, other than a further assessment of our results.

In $\mathbf{RQ}_2$ we proceeded with an automatic mechanism to study the agreement among the tools. As explained in Section 4.3, different SATs might possibly output the same rules in slightly different positions of the source code, e.g., highlighting the violation of a rule at two subsequent lines of code. To account for this aspect, we defined thresholds with which we could manage those cases where the same rules were presented in different locations. In this case, too, we cannot exclude possible imprecision; however, we extensively tested our automated data analysis script. More specifically, we manually validated a subset of rules for which the script indicated an overlap between two tools with the aim of assessing whether it was correct or not. This manual validation was conducted by one of the authors of this paper, who took into account a random sample of 300 candidate overlapping rules. In this sample, the author could not find any false positives, meaning that our script correctly identified the agreement among tools. This further analysis makes us confident of the validity of the findings reported for $\mathbf{RQ}_2$.

8. Conclusion

We performed a large-scale comparison of six popular static analysis tools (Better Code Hub, CheckStyle, Coverity Scan, FindBugs, PMD, and SonarQube) with respect to the detection of static analysis rules. We analyzed 47 Java projects from the Qualitas Corpus dataset, and derived similar rules that can be detected by the tools. We also compared their detection agreement at "line-level" and "class-level", and manually analyzed their precision.

Our future work includes an extension of this study with the evaluation of the recall, and the in-vivo assessment of the tools. Furthermore, we plan to conduct additional investigations into how different types of static analyses, e.g., taint tracking or typestate, can impact the detection capabilities observed in our study.

Acknowledgments

The authors would like to thank the Associate Editor and anonymous reviewers for the feedback provided during the review process. Fabio is partially supported by the Swiss National Science Foundation - SNF Project No. PZ00P2_186090. Furthermore, this work has been partially supported by the EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract No. 2020W3A5FY).

References

[1] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, I. Gorton, Measure it? Manage it? Ignore it? Software practitioners and technical debt, Symposium on the Foundations of Software Engineering (2015) 50–60.

[2] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, M. Di Penta, How open source projects use static code analysis tools in continuous integration pipelines, in: Int. Conf. on Mining Software Repositories (2017), pp. 334–344.

[3] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, A. Zaidman, How developers engage with static analysis tools in different contexts, Empirical Software Engineering (2019) 1–39.

[4] T. W. Thomas, H. Lipford, B. Chu, J. Smith, E. Murphy-Hill, What questions remain? an examination of how developers understand an interactive static analysis tool, in: Symposium on Usable Privacy and Security (2016).

[5] M. Mantere, I. Uusitalo, J. Roning, Comparison of static code analysis tools, in: Int. Conf. on Emerging Security Information, Systems and Technologies (2009), pp. 15–22.

[6] J. Wilander, M. Kamkar, A comparison of publicly available tools for static intrusion prevention, in: Workshop on Secure IT Systems (2002).

[7] N. Antunes, M. Vieira, Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services, in: International Symposium on Dependable Computing (2009).

[8] R. K. McLean, Comparing static security analysis tools using open source software, in: Int. Conf. on Software Security and Reliability (2012), pp. 68–74.

[9] M. A. Al Mamun, A. Khanam, H. Grahn, R. Feldt, Comparing four static analysis tools for java concurrency bugs, in: Third Swedish Workshop on Multi-Core Computing (2010).

[10] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in: 2013 35th International Conference on Software Engineering (ICSE 2013), IEEE, pp. 672–681.

[11] V. Lenarduzzi, A. Sillitti, D. Taibi, A survey on code analysis tools for software maintenance prediction, in: Software Engineering for Defence Applications - SEDA 2018, volume 925 of *Advances in Intelligent Systems and Computing (AISC)*, Springer-Verlag, 2019.

[12] S. Wagner, J. Jürjens, C. Koller, P. Trischberger, Comparing bug finding tools with reviews and tests, in: International Conference on Testing of Communicating Systems (2005), p. 40–55.

[13] N. Nagappan, T. Ball, Static analysis tools as early indicators of pre-release defect density, in: 27th International Conference on Software Engineering (ICSE 2005), pp. 580–586.

[14] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, M. A. Vouk, On the value of static analysis for fault detection in software, IEEE Transactions on Software Engineering 32 (2006) 240–253.

[15] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, P. Balachandran, Making defect-finding tools work for you, in: 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE 2010), p. 99–108.

[16] N. Saarimäki, V. Lenarduzzi, D. Taibi, On the diffuseness of code technical debt in open source projects, in: International Conference on Technical Debt (TechDebt 2019).

[17] V. Lenarduzzi, A. Martini, D. Taibi, D. A. Tamburri, Towards surgically-precise technical debt estimation: Early results and research roadmap, in: International Workshop on Machine Learning Techniques for Software Quality Evaluation (2019), MaLTeSQuE 2019, pp. 37–42.

[18] V. Lenarduzzi, N. Saarimäki, D. Taibi, Some sonarqube issues have a significant but smalleffect on faults and changes. a large-scale empirical study, Journal of Systems and Software 170 (2020).

[19] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended static checking for java, in: Conference on Programming Language Design and Implementation (2002), p. 234–245.

[20] S. Heckman, L. Williams, A systematic literature review of actionable alert identification techniques for automated static code analysis, Information and Software Technology 53 (2011) 363 – 387. Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.

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

[22] D. Kong, Q. Zheng, C. Chen, J. Shuai, M. Zhu, Isa: A source code static vulnerability detection system based on data fusion, in: International Conference on Scalable Information Systems, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[23] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, G. Pinto, Are static analysis violations really fixed? a closer look at realistic usage of sonarqube, in: 27th International Conference on Program Comprehension (ICPC 2019), p. 209–219.

[24] B. Lu, W. Dong, L. Yin, L. Zhang, Evaluating and integrating diverse bug finders for effective program analysis, in: L. Bu, Y. Xiong (Eds.), Software Analysis, Testing, and Evolution (2018), Cham, pp. 51–67.

[25] N. Rutar, C. B. Almazan, J. S. Foster, A comparison of bug finding tools for java, in: Symposium on Software Reliability Engineering (2004), pp. 245–256.

[26] P. Tomas, M. J. Escalona, M. Mejias, Open source tools for measuring the Internal Quality of Java software products. A survey, Computer Standards and Interfaces 36 (2013) 244–255.

[27] M. Schnappinger, M. H. Osman, A. Pretschner, A. Fietzke, Learning a classifier for prediction of maintainability based on static analysis tools, in: 27th International Conference on Program Comprehension (ICPC 2019), p. 243–248.

[28] V. Lenarduzzi, F. Lomio, H. Huttunen, D. Taibi, Are sonarqube rules inducing bugs?, International Conference on Software Analysis, Evolution and Reengineering (SANER) Preprint: arXiv:1907.00376 (2019).

[29] F. Rahman, S. Khatri, E. T. Barr, P. Devanbu, Comparing static bug finders and statistical prediction, in: 36th International Conference on Software Engineering (ICSE 2014), p. 424–434.

[30] P. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, N. Saarimäki, D. Sas, S. Soares de Toledo, A. Tsintzira, An overview and comparison of technical debt measurement tools, IEEE Software (2021).

[31] D. Taibi, A. Janes, V. Lenarduzzi, How developers perceive smells in source code: A replicated study, Information and Software Technology 92 (2017) 223 – 235.

[32] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, H. C. Gall, Context is king: The developer perspective on the usage of static analysis tools, in: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER 2018), pp. 38–49.

[33] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, C. Jaspan, Lessons from building static analysis tools at google, Commun. ACM 61 (2018) 58–66.

[34] T. Muske, R. Talluri, A. Serebrenik, Repositioning of static analysis alarms, in: 27th International Symposium on Software Testing and Analysis (2018), p. 187–197.

[35] E. Bodden, Self-adaptive static analysis, in: 40th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE 2018), p. 45–48.

[36] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu, To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools, Automated Software Engg. 22 (2015) 561–602.

[37] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, H. Mei, Automatic construction of an effective training set for prioritizing static analysis warnings, in: Int. Conf. on Automated software engineering (ASE 2010), pp. 93–102.

[38] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, G. Rothermel, Predicting accurate and actionable static analysis warnings: An experimental approach, in: 30th International Conference on Software Engineering (ICSE 2008), p. 341–350.

[39] N. Ayewah, W. Pugh, The google findbugs fixit, in: 19th International Symposium on Software Testing and Analysis (2010), p. 241–252.

[40] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh, Using static analysis to find bugs, IEEE Software 25 (2008) 22–29.

[41] M. Fowler, K. Beck, Refactoring: Improving the design of existing code, Addison-Wesley Longman Publishing Co., Inc. (1999).

[42] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, A. W. B. Regnell, Experimentation in Software Engineering: An Introduction, 2000.

[43] R. M. Terra, L. F. Miranda, M. T. Valente, R. da Silva Bigonha, Qualitas.class corpus: a compiled version of the qualitas corpus, ACM SIGSOFT Software Engineering Notes 38 (2013) 1–4.

[44] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, There and back again: Can you compile that snapshot?, Journal of Software: Evolution and Process 29 (2017) e1838.

[45] T. Lewowski, L. Madeyski, How far are we from reproducible research on code smell detection? a systematic literature review, Information and Software Technology 144 (2022) 106783.

[46] A. Singh, R. Bhatia, A. Singhrova, Taxonomy of machine learning algorithms in software fault prediction using object oriented metrics, Procedia computer science 132 (2018) 993–1001.

[47] C. Tavares, M. Bigonha, E. Figueiredo, Analyzing the impact of refactoring on bad smells, in: Proceedings of the 34th Brazilian Symposium on Software Engineering (2020), pp. 97–101.

[48] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[49] J. Saldaña, The coding manual for qualitative researchers, sage, 2021.

[50] M. Sandelowski, Sample size in qualitative research, Research in nursing & health 18 (1995) 179–183.

[51] J. Neyman, On the two different aspects of the representative method: the method of stratified sampling and the method of purposive selection, in: Breakthroughs in statistics, Springer, 1992, pp. 123–150.

[52] K. Krippendorff, Content analysis: An introduction to its methodology, Sage publications, 2018.

[53] J.-Y. Antoine, J. Villaneau, A. Lefeuvre, Weighted krippendorff's alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation., in: 14th Conference of the European Chapter of the Association for Computational Linguistics (2014), pp. 550–559.

[54] S. Nadi, T. Berger, C. Kästner, K. Czarnecki, Mining configuration constraints: Static analyses and empirical results, in: International Conference on Software Engineering (ICSE 2014), ACM, pp. 140–151.

[55] D. Di Nucci, F. Palomba, R. Oliveto, A. De Lucia, Dynamic selection of classifiers in bug prediction: An adaptive method, Transactions on Emerging Topics in Computational Intelligence 1 (2017) 202–212.

[56] M. Aniche, C. Treude, A. Zaidman, A. Van Deursen, M. A. Gerosa, Satt: Tailoring code metric thresholds for different software architectures, in: International Working Conference on Source Code Analysis and Manipulation (SCAM 2016), pp. 41–50.

[57] F. Fontana Arcelli, V. Ferme, M. Zanoni, A. Yamashita, Automatic metric thresholds derivation for code smell detection, in: International workshop on emerging trends in software metrics (2015), pp. 44–53.

[58] G. Catolino, F. Ferrucci, An extensive evaluation of ensemble techniques for software change prediction, Journal of Software: Evolution and Process (2019) e2156.

[59] G. Catolino, F. Palomba, F. Fontana Arcelli, A. De Lucia, A. Zaidman, F. Ferrucci, Improving change prediction models with code smell-related information, arXiv preprint arXiv:1905.10889 (2019).

[60] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, A. Zaidman, Enhancing change prediction models using developer-related factors, Journal of Systems and Software 143 (2018) 14–28.

[61] F. Palomba, M. Zanoni, F. Fontana Arcelli, A. De Lucia, R. Oliveto, Toward a smell-aware bug prediction model, Transactions on Software Engineering 45 (2017) 194–218.

[62] F. Fontana Arcelli, V. Ferme, M. Zanoni, Filtering code smells detection results, in: International Conference on Software Engineering-Volume 2 (ICSE 2015), pp. 803–804.

[63] P. E. Oliver, G. Marwell, Whatever happened to critical mass theory? a retrospective and assessment, Sociological Theory 19 (2001) 292–311.

[64] N. Imtiaz, B. Murphy, L. Williams, How do developers act on static analysis alerts? an empirical study of coverity usage, in: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE 2019), IEEE, pp. 323–333.

[65] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble, The qualitas corpus: A curated collection of java code for empirical studies, Asia Pacific Software Engineering Conference (2010) 336–345.

[66] V. Lenarduzzi, D. Tosi, L. Lavazza, S. Morasca, Why do developers adopt open source software? past, present and future, in: Open Source Systems, Springer International Publishing, 2019, pp. 104–115.

Appendix A

In the following, we report more results achieved in the context of $\mathbf{RQ}_2$. For each rules pair we computed the detection agreement at class level as reported in Table .13, Table .14, and Table .15, according to the process described in Section 4.4. It is worth remarking that, for the sake of readability, we only show the 10 most recurrent pairs. The results for the remaining thresholds are reported in the replication package[36].

Table .13: The 10 most recurrent rule pairs detected in the same class by the considered SATs and their corresponding agreement values (RQ_2)

SonarQube	**FindBugs**	**# SQ pairs**	**# FB pairs**	**SQ Agr.**	**FB Agr.**
CommentedOutCodeLine	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	2	1	0.000	1.000
S1155	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	1	1	0.000	1.000
S1186	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	1	1	0.000	1.000
S135	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	3	1	0.001	1.000
S1312	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	1	1	0.000	1.000
complex_class	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	1	1	0.000	1.000
S1195	DM_DEFAULT_ENCODING	1	1	1.000	0.001
S1195	EI_EXPOSE_REP	1	1	1.000	0.001
S1301	LG_LOST_LOGGER_DUE_TO_WEAK_REFERENCE	1	1	0.001	1.000
S1148	FI_NULLIFY_SUPER	1	1	0.000	1.000
SonarQube	**PMD**	**# SQ pairs**	**# PMD pairs**	**SQ Agr.**	**PMD Agr.**
S1192	FinalizeOnlyCallsSuperFinalize	1	1	0.000	1.000
S2110	JUnit4SuitesShouldUseSuiteAnnotation	1	1	1.000	0.001
S2110	AvoidCatchingGenericException	1	5	1.000	0.000
S2110	AvoidCatchingNPE	1	4	1.000	0.011
S2110	AvoidPrintStackTrace	1	1	1.000	0.000
S2110	CloseResource	1	2	1.000	0.000
S2110	CommentSize	1	2	1.000	0.000
S2110	DataflowAnomalyAnalysis	1	5	1.000	0.000
S2110	JUnit4TestShouldUseBeforeAnnotation	1	1	1.000	0.001
S2110	ShortVariable	1	25	1.000	0.000
SonarQube	**CheckStyle**	**# SQ pairs**	**# CHS pairs**	**SQ Agr.**	**CHS Agr.**
S2252	CommentsIndentationCheck	1	1	1.000	0.000
S2200	NeedBracesCheck	2	12	1.000	0.000
S2123	RedundantModifierCheck	2	56	1.000	0.002
S2123	InvalidJavadocPositionCheck	2	4	1.000	0.000
S2123	RegexpSinglelineCheck	2	42	1.000	0.000
S2123	WhitespaceAroundCheck	2	22	1.000	0.000
S2200	EqualsHashCodeCheck	2	1	1.000	0.002
S2200	FileTabCharacterCheck	2	8	1.000	0.000
S2200	FinalParametersCheck	2	9	1.000	0.000
S2200	IndentationCheck	2	1	1.000	0.000
SonarQube	**CoverityScan**	**# SQ pairs**	**# CS pairs**	**SQ Agr.**	**CS Agr.**
S1244	Unexpected control flow	2	2	0.001	1.000
S00105	Use of hard-coded cryptographic key	1	1	0.000	1.000
S1125	Dead default in switch	4	1	0.001	1.000
S1126	Dead default in switch	2	1	0.001	1.000
S1132	Dead default in switch	1	1	0.000	1.000
S1149	Dead default in switch	1	1	0.000	1.000
S1151	Dead default in switch	8	1	0.001	1.000
S1172	Dead default in switch	4	1	0.002	1.000
S1213	Dead default in switch	26	1	0.001	1.000
S1226	Dead default in switch	7	1	0.001	1.000
CheckStyle	**FindBugs**	**# CHS pairs**	**# FB pairs**	**CHS Agr.**	**FB Agr.**
AvoidStarImportCheck	SA_FIELD_SELF_COMPUTATION	8	2	0.000	1.000
FinalParametersCheck	NP_NONNULL_FIELD_NOT_INITIALIZED_IN_CONSTRUC-TOR	3	1	0.000	1.000
RedundantModifierCheck	IC_SUPERCLASS_USES_SUBCLASS_DURING_INITIALIZA-TION	1	1	0.000	1.000
DesignForExtensionCheck	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	8	1	0.000	1.000
NeedBracesCheck	TQ_EXPLICIT_UNKNOWN_SOURCE_VALUE_REACHES_-NEVER...	28	1	0.000	1.000
JavadocVariableCheck	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	8	1	0.000	1.000
JavadocVariableCheck	IC_SUPERCLASS_USES_SUBCLASS_DURING_INITIALIZA-TION	1	1	0.000	1.000
WhitespaceAroundCheck	BIT_IOR	10	1	0.000	1.000
JavadocPackageCheck	IC_SUPERCLASS_USES_SUBCLASS_DURING_INITIALIZA-TION	1	1	0.000	1.000
MissingJavadocMethodCheck	FI_MISSING_SUPER_CALL	149	1	0.001	1.000

Agr. means Agreement

Table .14: The 10 most recurrent rule pairs detected in the same class by the considered SATs and their corresponding agreement values (RQ$_2$)

BetterCodeHub	**CheckStyle**	**# BCH pairs**	**# CHS pairs**	**BCH Agr.**	**CHS Agr.**
WRITE_SIMPLE_UNITS	AvoidEscapedUnicodeCharactersCheck	1	29859	0.000	0.529
WRITE_SHORT_UNITS	AvoidEscapedUnicodeCharactersCheck	1	29859	0.000	0.529
WRITE_CODE_ONCE	AvoidEscapedUnicodeCharactersCheck	1	29859	0.000	0.529
WRITE_SIMPLE_UNITS	OperatorWrapCheck	1	60694	0.000	0.312
WRITE_SHORT_UNITS	OperatorWrapCheck	1	60694	0.000	0.312
WRITE_CODE_ONCE	OperatorWrapCheck	1	60694	0.000	0.312
WRITE_SHORT_UNITS	IllegalTokenTextCheck	4	418	0.001	0.306
WRITE_SIMPLE_UNITS	IllegalTokenTextCheck	4	418	0.001	0.306
AUTOMATE_TESTS	IllegalTokenTextCheck	2	418	0.000	0.306
WRITE_CODE_ONCE	AtclauseOrderCheck	30	210	0.002	0.306
BetterCodeHub	**CoverityScan**	**# BCH pairs**	**# CS pairs**	**BCH Agr.**	**CS Agr.**
AUTOMATE_TESTS	Exception leaked to user interface	2	1	0.000	1.000
SEPARATE_CONCERNS_IN_MODULES	Unsafe reflection	2	2	0.000	1.000
COUPLE_ARCHITECTURE_COMPONENTS_LOOSELY	AT: Possible atomicity violation	2	1	0.001	1.000
WRITE_CLEAN_CODE	Use of hard-coded cryptographic key	20	1	0.001	1.000
WRITE_SIMPLE_UNITS	Exception leaked to user interface	2	1	0.000	1.000
WRITE_SHORT_UNITS	Exception leaked to user interface	2	1	0.000	1.000
WRITE_CODE_ONCE	Exception leaked to user interface	2	1	0.000	1.000
WRITE_SHORT_UNITS	Unsafe reflection	2	2	0.000	1.000
WRITE_SIMPLE_UNITS	Unsafe reflection	2	2	0.000	1.000
AUTOMATE_TESTS	Dead default in switch	2	1	0.000	1.000
BetterCodeHub	**FindBugs**	**# BCH pairs**	**# FB pairs**	**BCH Agr.**	**FB Agr.**
WRITE_CODE_ONCE	ICAST_BAD_SHIFT_AMOUNT	12	8	0.001	1.000
WRITE_SIMPLE_UNITS	SA_LOCAL_SELF_ASSIGNMENT_INSTEAD_OF_FIELD	2	1	0.000	1.000
AUTOMATE_TESTS	FI_MISSING_SUPER_CALL	3	1	0.000	1.000
SMALL_UNIT_INTERFACES	ICAST_BAD_SHIFT_AMOUNT	12	8	0.002	1.000
WRITE_SHORT_UNITS	INT_VACUOUS_BIT_OPERATION	6	1	0.001	1.000
WRITE_SHORT_UNITS	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	4	1	0.001	1.000
SEPARATE_CONCERNS_IN_MODULES	EQ_COMPARING_CLASS_NAMES	2	1	0.000	1.000
AUTOMATE_TESTS	NP_ALWAYS_NULL_EXCEPTION	2	1	0.000	1.000
WRITE_CLEAN_CODE	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	4	1	0.000	1.000
AUTOMATE_TESTS	ICAST_BAD_SHIFT_AMOUNT	3	8	0.000	1.000
BetterCodeHub	**PMD**	**# BCH pairs**	**# PMD pairs**	**BCH Agr.**	**PMD Agr.**
SEPARATE_CONCERNS_IN_MODULES	FinalizeOnlyCallsSuperFinalize	2	1	0.000	1.000
WRITE_CODE_ONCE	FinalizeOnlyCallsSuperFinalize	4	1	0.000	1.000
SEPARATE_CONCERNS_IN_MODULES	AvoidMultipleUnaryOperators	3	2	0.001	1.000
COUPLE_ARCHITECTURE_COMPONENTS_LOOSELY	AvoidMultipleUnaryOperators	3	2	0.001	1.000
SMALL_UNIT_INTERFACES	FinalizeOnlyCallsSuperFinalize	4	1	0.001	1.000
WRITE_CLEAN_CODE	InvalidLogMessageFormat	2	1	0.000	1.000
AUTOMATE_TESTS	FinalizeOnlyCallsSuperFinalize	2	1	0.000	1.000
AUTOMATE_TESTS	EmptyStatementBlock	2	160	0.000	0.748
WRITE_SHORT_UNITS	EmptyStatementBlock	2	160	0.000	0.748
WRITE_SIMPLE_UNITS	EmptyStatementBlock	4	160	0.001	0.748
CheckStyle	**CoverityScan**	**# CHS pairs**	**# CS pairs**	**CHS Agr.**	**CS Agr.**
FinalParametersCheck	Unsafe reflection	313	2	0.001	1.000
InvalidJavadocPositionCheck	IP: Ignored parameter	4	2	0.000	1.000
NoWhitespaceAfterCheck	IP: Ignored parameter	1	2	0.000	1.000
MissingJavadocMethodCheck	IP: Ignored parameter	112	2	0.001	1.000
MagicNumberCheck	IP: Ignored parameter	50	2	0.000	1.000
LineLengthCheck	IP: Ignored parameter	6	2	0.000	1.000
JavadocVariableCheck	IP: Ignored parameter	36	2	0.000	1.000
JavadocStyleCheck	IP: Ignored parameter	84	2	0.001	1.000
JavadocMethodCheck	IP: Ignored parameter	66	2	0.001	1.000
IndentationCheck	IP: Ignored parameter	1152	2	0.000	1.000

Agr. means Agreement

Table .15: The 10 most recurrent rule pairs detected in the same class by the considered SATs and their corresponding agreement values (RQ$_2$)

CheckStyle	**PMD**	**# CHS pairs**	**# PMD pairs**	**CHS Agr.**	**PMD Agr.**
FinalParametersCheck	AvoidMultipleUnaryOperators	51	2	0.000	1.000
RegexpSinglelineCheck	AvoidMultipleUnaryOperators	75	2	0.000	1.000
NoFinalizerCheck	FinalizeOnlyCallsSuperFinalize	1	1	0.015	1.000
VisibilityModifierCheck	InvalidLogMessageFormat	15	1	0.000	1.000
AbbreviationAsWordInNameCheck	FinalizeOnlyCallsSuperFinalize	1	1	0.000	1.000
NoWhitespaceAfterCheck	AvoidMultipleUnaryOperators	7	2	0.000	1.000
VariableDeclarationUsageDistanceCheck	AvoidMultipleUnaryOperators	7	2	0.001	1.000
NeedBracesCheck	AvoidMultipleUnaryOperators	90	2	0.000	1.000
JavadocParagraphCheck	FinalizeOnlyCallsSuperFinalize	1	1	0.000	1.000
NonEmptyAtclauseDescriptionCheck	FinalizeOnlyCallsSuperFinalize	10	1	0.000	1.000
CoverityScan	**FindBugs**	**#CS pairs**	**#FB pairs**	**CS Agr.**	**FB Agr.**
UCF: Useless control flow	BC_UNCONFIRMED_CAST_OF_RETURN_VALUE	1	2	1.000	0.001
Dead default in switch	DLS_DEAD_LOCAL_STORE	1	5	1.000	0.004
DLS: Dead local store	NP_ALWAYS_NULL_EXCEPTION	3	1	0.010	1.000
UCF: Useless control flow	BC_UNCONFIRMED_CAST	1	1	1.000	0.000
Unsafe reflection	NP_LOAD_OF_KNOWN_NULL_VALUE	2	2	1.000	0.012
UCF: Useless control flow	UCF_USELESS_CONTROL_FLOW	1	1	1.000	0.011
OGNL injection	RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	1	1	1.000	0.001
Failure to call super.finalize()	FI_NULLIFY_SUPER	1	1	0.500	1.000
REC: RuntimeException capture	RV_RETURN_VALUE_OF_PUTIFABSENT_IGNORED	1	1	0.013	1.000
USELESS_STRING: Useless/non-informative string...	FI_MISSING_SUPER_CALL	5	1	0.250	1.000
CoverityScan	**PMD**	**# CS pairs**	**# PMD pairs**	**CS Agr.**	**PMD Agr.**
Dead default in switch	AvoidInstantiatingObjectsInLoops	1	1	1.000	0.000
TLW: Wait with two locks held	ShortVariable	1	10	1.000	0.000
TLW: Wait with two locks held	UseCorrectExceptionLogging	1	3	1.000	0.010
TLW: Wait with two locks held	UseConcurrentHashMap	1	5	1.000	0.002
TLW: Wait with two locks held	UnusedImports	1	21	1.000	0.000
TLW: Wait with two locks held	UnnecessaryFullyQualifiedName	1	2	1.000	0.000
TLW: Wait with two locks held	TooManyMethods	1	1	1.000	0.000
TLW: Wait with two locks held	TooManyFields	1	1	1.000	0.001
TLW: Wait with two locks held	StdCyclomaticComplexity	1	2	1.000	0.000
TLW: Wait with two locks held	ProperLogger	1	2	1.000	0.001
FindBugs	**PMD**	**# FB pairs**	**# PMD pairs**	**FB Agr.**	**PMD Agr.**
DMI_EMPTY_DB_PASSWORD	LawOfDemeter	1	9	1.000	0.000
TESTING	OnlyOneReturn	1	169	1.000	0.002
ICAST_BAD_SHIFT_AMOUNT	DataflowAnomalyAnalysis	8	100	1.000	0.001
BIT_IOR	SignatureDeclareThrowsException	1	14	1.000	0.000
BC_IMPOSSIBLE_DOWNCAST_OF_TOARRAY	ForLoopCanBeForeach	1	6	1.000	0.000
LG_LOST_LOGGER_DUE_TO_WEAK_REFERENCE	MethodArgumentCouldBeFinal	1	62	1.000	0.000
QF_QUESTIONABLE_FOR_LOOP	MethodArgumentCouldBeFinal	1	6	1.000	0.000
HRS_REQUEST_PARAMETER_TO_HTTP_HEADER	MethodArgumentCouldBeFinal	1	6	1.000	0.000
ICAST_BAD_SHIFT_AMOUNT	ConfusingTernary	8	4	1.000	0.000
BIT_IOR	ConfusingTernary	1	3	1.000	0.000
SonarQube	**BetterCodeHub**	**# SQ pairs**	**# BCH pairs**	**SQ Agr.**	**BCH Agr.**
S2275	COUPLE_ARCHITECTURE_COMPONENTS_LOOSELY	1	2	1.000	0.001
S2275	AUTOMATE_TESTS	1	2	1.000	0.000
S888	COUPLE_ARCHITECTURE_COMPONENTS_LOOSELY	2	4	0.667	0.001
S888	WRITE_CLEAN_CODE	2	10	0.667	0.001
S888	WRITE_CODE_ONCE	2	26	0.667	0.002
S888	AUTOMATE_TESTS	2	2	0.667	0.000
S888	SEPARATE_CONCERNS_IN_MODULES	2	2	0.667	0.000
ObjectFinalizeOverridenCalls SuperFinalizeCheck	AUTOMATE_TESTS	1	3	0.500	0.000
S2232	SEPARATE_CONCERNS_IN_MODULES	1	1	0.500	0.000
S2232	AUTOMATE_TESTS	1	1	0.500	0.000

Agr. means Agreement

Table .16: Programming languages supported by the tools.

Tool	C	C	C++	Go	Groovy	Java	Javascript	Objective C	Perl	Python	Ruby	Scala
Better Code Hub	x	x	x	x	x	x	x	x	x	x	x	x
Checkstyle						x						
Coverity	x	x	x	x		x	x	x		x	x	
FindBugs						x						
PMD						x	x					x
SonarQube	x	x	x	x		x	x	x		x	x	x
Tool	**Shell Script**	**Solidity**	**Swift**	**TypeScript**	**Kotlin**	**VB.NET**	**Salesforce.com**	**Visualforce**	**Modelica**	**PL**	**PL**	**Apache Velocity**
Better Code Hub	x	x	x	x	x							
Checkstyle												
Coverity			x		x	x						
FindBugs												
PMD							x	x	x	x		x
SonarQube			x	x	x	x				x	x	
Tool	**XML**	**XSL**	**Terraform**	**CloudFormation**	**ABAP**	**COBOL**	**CSS**	**Flex**	**HTML5**	**RPG**	**T-SQL**	**VB6**
Better Code Hub												
Checkstyle												
Coverity												
FindBugs												
PMD	x	x										
SonarQube	x		x	x	x	x	x	x	x	x	x	x