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

Valentina Lenarduzzi · Savanna Lujan · Nyyti Saarimäki · Fabio Palomba

Received: date / Accepted: date

Abstract Background. Developers use Automated Static Analysis Tools (ASATs) 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 ASATs to favor the understanding of their actual capabilities.

Aims. Despite the work done so far, there is still a lack of knowledge regarding (1) which source quality problems can actually be detected by static analysis tool warnings, (2) what is their agreement, and (3) what is the precision of their recommendations. We aim at bridging this gap by proposing a large-scale comparison of six popular static analysis tools for Java projects: Better Code Hub, CheckStyle, Coverity Scan, Findbugs, PMD, and SonarQube.

Method. We analyze 47 Java projects and derive a taxonomy of warnings raised by 6 state-of-the-practice ASATs. To assess their agreement, we compared them by manually analyzing - at line-level - whether they identify the same issues. Finally, we manually evaluate the precision of the tools.

Results. The key results report a comprehensive taxonomy of ASATs warnings, show little to no agreement among the tools and a low degree of precision.

Conclusions. We provide a taxonomy that can be useful to researchers, practitioners, and tool vendors to map the current capabilities of the tools. Further-
more, our study provides the first overview on the agreement among different tools as well as an extensive analysis of their precision.

**Keywords** Static analysis tools · Software Quality · Empirical Study.

1 Introduction

Automated Static Analysis Tools (ASATs) are instruments that analyze characteristics of the source code without executing it, so that they can discover potential source code quality issues [14]. These tools are getting more popular as they are becoming easier to use—especially in continuous integration pipelines [55]—and there is a wide range to choose from [52]. 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 [48].

To help practitioners with this selection process, researchers have been conducting empirical studies to compare the capabilities of existing ASATs [29, 54]. Most of these investigations have focused on (1) the features provided by the tools, e.g., which maintainability dimensions can be tracked by current ASATs, (2) comparing specific aspects considered by the tools, such as security [4, 31] or concurrency defects [1], and (3) assessing the number of false positives given by the available static analysis tools [20].

Recognizing the effort spent by the research community, which led to notable advances in the way tool vendors develop ASATs, 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 them and improve the level of support provided to developers.

In this paper, we aim to address this gap of knowledge by designing and conducting a large-scale empirical investigation into the detection capabilities of six popular state-of-the-practice ASATs, namely SonarQube, Better Code Hub, Coverity Scan, Findbugs, PMD, and CheckStyle. Specifically, we run the considered tools against a corpus of 47 projects from the Qualitas Corpus dataset and (1) depict which source quality problems can actually be detected by the tools, (2) compute the agreement among the recommendations given by them at line-level, and (3) manually compute the precision of the tools.

---

1ASATs verify code compliance with a specific set of warnings 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.
The key results of the study report a taxonomy of static analysis tool warnings, but also show that, among the considered tools, SonarQube is the one able to detect most of the quality issues that can be detected by the other ASATs. 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 usefulness of some tools is seriously threatened by the presence of false positives—this result corroborates and enlarges previous findings [20] on a larger scale and considering a broader set of tools.

Based on our findings, our paper finally discusses and distills a number of lessons learned, limitations, and open challenges that should be considered and/or addressed by both the research community and tool vendors.

Structure of the paper. The study setting is described in Section 2. The results are presented in Section 3 and discussed in Section 4. Section 5 identifies the threats to the validity of our study, while Section 6 presents related works on static analysis tools. Finally, in Section 7 we draw conclusions and provide an outlook on our future research agenda.

2 Empirical Study Design

We designed our empirical study as a case study based on the guidelines defined by Runeson and Höst [39]. The following sections describe the goals and specific research questions driving our empirical study as well as the data collection and analysis procedures.

2.1 Goal and Research Questions

The goal of our empirical study is to compare state-of-the-practice ASATs 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 available on the identification of source code quality issues with ASATs 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 challenges, 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.

More specifically, our goal can be structured around three main research questions (RQs). As a first step, we aimed at understanding what kind of issues different tool warnings detect when run on source code. An improved analysis of this aspect may extend our knowledge on whether and how various types of quality issues are identified by the existing tools. Hence, we asked:
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 ASATs 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.

Finally, we focused on investigating the potential usefulness of the tools in practice. While they could output numerous warnings that alert developers of the presence of potential quality problems, it is still possible that some of these warnings 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 [20]; our study aims at corroborating and extending the available findings, as further remarked in Section 2.4.

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.

2.2 Context of the Study

The context of our study consisted of software systems and tools. In the following, we describe our selection.

Project Selection

We selected projects from the Qualitas Corpus collection of software systems (Release 20130901), using the compiled version of the Qualitas Corpus [47]. The dataset contains 112 Java systems with 754 versions, more than 18 million LOCs, 16,000 packages, and 200,000 classes analyzed. Moreover, the
dataset includes projects from different contexts such as IDEs, databases, and programming language compilers. More information is available in [47]. In our study, we considered the “r” release of each of the 112 available systems. Since two of the automated static analysis tools 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\(^2\).

**ASATs Selection**

We selected the six tools described below. The choice of focusing on those specific tools was driven by the familiarity of the authors with them: this 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.

**SonarQube**\(^3\) 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 warnings” defined for most common development languages. If the analyzed source code violates a coding warning, the tool reports an “issue”. The time needed to remove these issues is called remediation effort.

SonarQube includes reliability, maintainability, and security warnings. Reliability warnings, 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 the term “code smells” adopted in SonarQube does not refer to the commonly known term code smells defined by Fowler et al. [18], but to a different set of warnings.

**Coverity Scan**\(^4\) 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\(^5\). 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

---

\(^2\)The GitHub projects, with the related configuration adopted for executing the tools, will be made public in the case of acceptance of this paper.

\(^3\)http://www.sonarsource.org/

\(^4\)https://scan.coverity.com/

\(^5\)https://scan.coverity.com/faq\#what-is-coverity-scan
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.6

Better Code Hub7 is also 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.8

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.

Checkstyle9 is an open-source developer 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 Style10 and Sun Java Style11. In addition to standard checks provided by Checkstyle, customized configuration files are also possible according to user preference.12

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 warning, with the first reporting actual problems and the second possible issues to be verified.

FindBugs14 is a static analysis tool for evaluating Java code, more precisely Java bytecode. 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

---

6https://devguide.python.org/coverity
7https://bettercodehub.com/
8https://pybit.es/bettercodehub.html
9https://checkstyle.org
10https://checkstyle.sourceforge.io/google_style.html
11https://checkstyle.sourceforge.io/sun_style.html
12https://checkstyle.sourceforge.io/index.html
13https://checkstyle.sourceforge.io/checks.html
14http://findbugs.sourceforge.net
following main reasons: difficult language features, misunderstood API features, misunderstood invariants when code is modified during maintenance, and garden variety mistakes.\textsuperscript{15, 16} 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\textsuperscript{17}.

PMD\textsuperscript{18} 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 binary distributions. PMD uses a set of warnings 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 warning set configurations\textsuperscript{19} for Java projects. The warning sets can also be customized according to the user preference\textsuperscript{20}. These warnings are classified under 8 different categories: best practices, code style, design, documentation, error prone, multi threading, performance, and security. Moreover, the violations of warnings are measured on a priority scale from 1-5, with 1 being the most severe and 5 being the least\textsuperscript{21}.

2.3 Study Setup and Data Collection

This section describes the study setup used to collect the data from each tool and the data collection process. We analyzed a single 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.

We executed SonarQube on each project using SDK 8 (Oracle) and the sonar-scanner package version 4.2. Each project was analyzed using the original sources and the binaries provided in the dataset. Moreover, we configured the analysis (in the file sonar-project.properties) reporting information regarding the project key, project name, project version, source directories, test directories, binary directories, and library directories. It is important to

\textsuperscript{15}http://findbugs.sourceforge.net/findbugs2.html
\textsuperscript{16}http://findbugs.sourceforge.net/factSheet.html
\textsuperscript{17}http://findbugs.sourceforge.net/bugDescriptions.html
\textsuperscript{18}https://pmd.github.io/latest/
\textsuperscript{19}https://github.com/pmd/pmd/tree/master/pmd-java/src/main/resources/rulesets/java
\textsuperscript{20}https://pmd.github.io/latest/index.html
\textsuperscript{21}https://pmd.github.io/latest/pmd$_$rules$_$java.html
note that the analysis performed using the original binaries reduced the issues of compilation errors and missing libraries. Moreover, it also helped to reduce issues related to false positives. Once the configuration file was set up, the analysis began by running sonar scanner in the root directory of the project:

```bash
$ sonar-scanner
```

After all the projects had been analyzed, we extracted the data related to the projects using the "SonarQube Exporter" tool. This makes it possible to extract SonarQube project measures and issues in CSV format. Under the target directory, the extraction directory was made. The CSV files were then extracted from the server by running the following command:

```bash
$ java-jar csv-from-sonar-0.4.1-jar-with-dependencies.jar
```

All projects starting with “QC:” were extracted from the server by clicking “Save issues” and “Save measures”. This exported CSV files into the extraction directory for each project.

**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. For our projects, the following commands were used in the directory of the project where the build.xml and pom.xml files reside:

- `ant` (build.xml) for building and deploying Java projects. The projects had various Java versions, so the appropriate Java version was installed according to the documentation (if available) for building.
  ```bash
  − $ cov-build {dir cov-int ant jar
  − $ cov-build {dir cov-int ant
  − maven (pom.xml) for building projects. The appropriate maven version was used according to the documentation.
  − $ cov-build {dir cov-int mvn install
  − $ cov-int {dir cov-int mvn clean install
  ``

After building the project, if over 85% of tests were successfully executed, a tar archive of the cov-int build folder was created by running:

```bash
$ tar czvf myproject.tgz cov-int
```

Once the tar archive was created, the file was submitted through the project dashboard at https://scan.coverity.com/projects/myproject to “Submit Build”. The analysis was then performed, and the results were displayed on the dashboard. The data regarding the analysis was extracted from

---

22[https://docs.sonarqube.org/latest/analysis/languages/java/](https://docs.sonarqube.org/latest/analysis/languages/java/)

23[https://github.com/pervcomp/SonarQube-Exporter/releases](https://github.com/pervcomp/SonarQube-Exporter/releases)
the dashboard from “View Defects”, where the detailed report regarding the analysis results was located. Under Outstanding Issues -> Export CSV, the CSV file containing all issues of the project was downloaded for each project.

**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. The analysis was conducted between January 2019 and February 2019, so the Better Code Hub version hosted during that time was used.

Once the configuration file had been created, it was saved in the root directory of the project. The local changes to the project were added, committed, and pushed to GitHub. Afterwards, the project analysis started from Better Code Hub's dashboard, which is connected to the GitHub account https://bettercodehub.com/repositories. The analysis started by clicking the “Analyze” button on the webpage. The results were then shown on the webpage. Once all the projects had been analyzed, the data was sent in one CSV file. All projects containing more than 100,000 lines of code were not analyzed, due to Better Code Hub’s limit.

**Checkstyle.** The JAR file for the Checkstyle analysis was downloaded directly from Checkstyle’s website\(^\text{24}\) 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, Checkstyle offers two different types of warning sets for the analysis\(^\text{24}\).

For each of the warning sets, the configuration file was downloaded directly from Checkstyle’s website\(^\text{25}\). In order to start the analysis, the files checkstyle=8.30-all.jar and the configuration file in question were saved in the directory where all the cloned repositories from Java Qualitas Corpus resided. To make the analysis more swift, a bash script was made to execute the analysis for each project in one go. This can be seen in Listing 1.

```
#!/bin/bash
while read in; do java -jar checkstyle-8.30-all.jar -c /RULESET -f xml "$in"/ > "$in"_CS_RULESET.xml; done < projectList.txt
```

where RULESET represents the warning set used for the analysis, "$in" represents the project name which is imported from projectList.txt, and "$in"_CS_RULESET.xml represents the export file name of the analysis results in XML format. The text file projectList.txt consists of all the project names, in order to execute the analysis for all projects in one go. An example of how the

\(^\text{24}\)https://checkstyle.org/#Download
\(^\text{25}\)https://github.com/checkstyle/checkstyle/tree/master/src/main/resources
projects were analyzed with Checkstyle according to the warning set Google Java Style\textsuperscript{26} is show in Listing 2.

\begin{flushleft}
\textbf{Listing 2} Example of Checkstyle bash script for Google Java Style configuration.
\end{flushleft}

\begin{verbatim}
#!/bin/bash
while read in; do java -jar checkstyle-8.30-all.jar
    -c /google_checks.xml -f xml "$in"/ > "$in".CS_Google_Checks.xml;
done < projectList.txt
\end{verbatim}

\textbf{Findbugs.} FindBugs 3.0.1 was installed by running \texttt{brew install findbugs} in the command line. Once installed, the GUI was then engaged by writing \texttt{spotbugs}. From the GUI, the analysis was executed through \texttt{File} $\rightarrow$ \texttt{New Project}. The classpath for the analysis was identified to be the location of the project directory.

Moreover, the source directories were identified to be the project JAR executables. Once the classpath and source directories were identified, the analysis was engaged by clicking \texttt{Analyze} in the GUI. Once the analysis finished, the results were saved through \texttt{File} $\rightarrow$ \texttt{Save as} using the XML file format.

\textbf{PMD.} PMD 6.23.0 was downloaded from GitHub\textsuperscript{27} as a zip file. After unzipping, the analysis was engaged by identifying several parameters: project directory, export file format, warning set, and export file name. In addition to downloading the zip file, PMD offers 32 different types of warning sets for Java written projects\textsuperscript{28}. We developed a bash script, shown in Listing 3, to engage the analysis for each project in one go.

\begin{flushleft}
\textbf{Listing 3} PMD bash script tailored towards each warning set.
\end{flushleft}

\begin{verbatim}
#!/bin/bash
while read in; do
    $HOME/pmd-bin-6.23.0/bin/run.sh pmd
dir "$in"/
    -f xml -R rulesets/java/RULESET
    -reportfile "$in".PMD_RULESET.xml;
done < projectList.txt
\end{verbatim}

The parameter \texttt{HOME} represents the full path where the binary resides, "$in" represents the project name which is imported from \texttt{projectList.txt}, \texttt{RULESET} represents the warning set used for the analysis, and "$in".PMD_RULESET.xml represents the export file name of the analysis results in XML format. Just like in Listing 1, \texttt{projectList.txt} consists of all the

\textsuperscript{26}https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/google_checks.xml

\textsuperscript{27}https://github.com/pmd/pmd/releases/download/pmd$_$releases\%2F6.23.0/pmd-bin-6.23.0.zip

\textsuperscript{28}https://github.com/pmd/pmd/tree/master/pmd-java/src/main/resources/rulesets/java
project names. An example of how the projects were analyzed for the warning set Clone Implementation\textsuperscript{29} is show in Listing 4.

Listing 4 Example of PMD bash script for Clone Implementation configuration.

```bash
#!/bin/bash
while read in
do
  $HOME/pmd-bin-6.23.0/bin/run.sh pmd -dir "$in"/
  -f xml -R rulesets/java/clone.xml
  -reportfile "$in"_PMD_Clone.xml;
done < projectList.txt
```

2.4 Data Analysis

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

Source code quality issues identified by the tools (RQ$_1$). In order to determine the tool detection capabilities and warnings overlaps, we first identified warnings that can be detected by each tool, also considering type (if available), category, and severity. Then, we calculated how many warnings are violated in our projects.

Agreement among the tools (RQ$_2$). We expected that similar warnings should be violated in the same class, and in particular in the same position of the code. For the tools that provided information on the exact position (the lines of code where the warning is detected), we analyzed the agreement using the bottom-up approach. Therefore, we examined the overlapping positioning (start and end lines) of the warnings and identifying whether the warnings highlight the same kind of issue. Essentially pairing the warnings based on position and checking whether they have similar definitions.

In order to check whether the warning pairs identified in RQ$_1$ appear in the same position in the code, we scanned and checked in which classes each warning was violated. Then we counted how many times the warning pairs were violated in the same class. Only warning pairs identified across the tools were considered. warnings detected only by one tool were not considered.

For example, there was a warning pair identified between SonarQube and Better Code Hub: “SQ$_1$ and BCH$_1$”. Considering the warning pair detected by SonarQube (SQ) and Better Code Hub (BCH), we expected that these two warnings would be violated at least in the same class, since they identify the same kind of issue. So if for instance SQ$_1$ is violated 100 times and BCH$_1$ 130 times, we would expect to find the warning pair violated at least 100 times in the same class. Hence, we can calculate the agreement between the warnings.

\textsuperscript{29}https://github.com/pmd/pmd/blob/master/pmd-java/src/main/resources/rulesets/java/clone.xml
in the warning pair as follows:

\[ \text{Agreement}_r = \frac{\#SR}{\#\text{issues}_r} \]  

(1)

where \#SR is the number of instances in which a warning pair is detected together in the same class, and \#issues\_r is the overall number of issues of warning r detected in the warning pair.

Agreement is calculated separately for both warnings in a warning pair. Perfect overlap is obtained if the agreement for both warnings is equal to one. This means that all the issues generated by those warnings are always detected in the same classes, and no issues are detected separately in a different class.

After analyzing using the top-down approach from the definition level to the class level, we continued the analysis using the bottom-up approach from the line level to the definition level. For each class, we considered the start and end lines of the issue and compared the degree to which the warnings overlap according to the lines affected.

As the granularity of the warnings varies between the tools, we checked what fraction of the affected lines are overlapping between the warnings, instead of requiring sufficient overlap between both warnings. This was done by selecting one issue at a time (reference) and comparing all other issues (comparison) to that. If the lines affected by the comparison warning were residing within the lines affected by the reference warning, we defined the warnings as overlapped. To quantify the degree of overlapping, we used the percentage of the lines affected by the comparison issue that overlapped with the reference issue. The results were grouped based on four percentage thresholds: 100%, 90%, 80% and 70%.

The concept is visualized in Figure 1. The lines represent the issues in a code file, indicating the start and end of the affected code lines. The percentages represent the ratio of the lines affected by the warning that lie within the lines affected by the reference warning. Depending on the threshold used, different warnings are selected based on the overlapping percentage.

Unfortunately, only SonarQube, Better Code Hub, Checkstyle, PMD, and FindBugs provide information about the actual "position" of the detected warnings. They report information about the "start line" and "end line" for each class. Regarding Coverity Scan, this information is only available in the
web interface, but not in the APIs. Moreover, Coverity Scan licence does not allow to crawl the web interface. Since we have detected 8,828 Coverity Scan warnings violated in our projects, it would not have been visible to report this information manually.

**Precision of the tools (RQ3).** In our last research question, we aimed at assessing the precision of the considered tools. From a theoretical point of view, precision is defined as the ratio between the true positive source code quality 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:

\[
\text{precision} = \frac{\text{TPs}}{\text{TPs} + \text{FPs}}
\]  

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 [20, 52]. 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 quality 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 quality issues identified by each tool. In the context of our work, we conducted a manual analysis of the warnings highlighted by the six considered tools, thus marking each of them as true or false positive based on our analysis of (1) the quality 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 warnings 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 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.

To increase the reliability of this manual analysis, two of the authors of this paper (henceforth called the inspectors) first independently analyzed the warning 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 warning identified by the tool that the inspectors had to verify manually; and (3) the warning type and specification, e.g., the code smell. The inspectors’ task was to go over each of the warnings and add a seventh
column in the spreadsheet that indicated whether the warning was a true or a false positive. 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 warning 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 $\alpha$ [22] and which is higher than 0.80, which has been reported to be the standard reference score for $\alpha$ [3].

In Section 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.

2.5 Replicability

In order to allow the replication of our study, we have published the raw data in a replication package.$^{30}$

3 Analysis of the Results

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

RQ1. What quality issues can be detected by Static Analysis Tools?

Here we analyzed how many warnings are actually output by the considered static analysis tools as well as the types of issues they are able to discover.

Static Analysis Tools Detection Capability. We report the detection capability of each tool in terms of how many warnings can be detected, and the classification of internal warnings (e.g., type and severity). Moreover, we report the diffusion of the warning in the selected projects.

Better Code Hub detects a total of 10 warnings, of which 8 are grouped based on type and severity. Better Code Hub categorizes the 8 warnings under 3 types: RefactoringFileCandidateWithLocationList, RefactoringFileCandidate, and RefactoringFileCandidateWithCategory. Of these 8 warnings, 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 warnings: Medium, High, and Very High. Of these eight warnings, four were classified as Medium severity, four as High severity, and eight as Very High severity. Some of the warnings have more than one severity possibly assigned to them.

Checkstyle detects a total of 173 warnings which are grouped based on type and severity. Checkstyle categorizes the 173 warnings under 14 types:

$^{30}$ https://figshare.com/s/3c40dce067507b3e6c63
Annotations, Block Checks, Class Design, Coding, Headers, Imports, Javac Comments, Metrics, Miscellaneous, Modifiers, Naming Conventions, Regexp, Size Violations, and Whitespace. Of these 173 warnings, 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 Warning. The distribution of the checks with respect to the severity levels is not provided in the documentation.

Coverity Scan’s total scope of detectable warnings 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 warnings. These warnings were classified under three severity levels: Low, Medium, and High. Of these 130 warnings, 48 were classified as Low severity, 87 as Medium severity, and 12 as High severity. Like Better Code Hub, some of Coverity Scan’s warnings have more than one severity type assigned to them.

Findbugs detects a total of 424 warnings grouped based on type and severity. It categorizes the 424 warnings under 9 types: Bad practice, Correctness, Experimental, Internationalization, Malicious code vulnerability, Multithreaded correctness, Performance, Security, and Dodgy code. Of these 424 warnings, 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.

PMD detects a total of 305 warnings which are grouped based on type and severity. PMD categorizes the 305 warnings under 8 types: Best Practices, Code Style, Design, Documentation, Error Prone, Multithreading, Performance, and Security. Of these 305 warnings, 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 warnings according to five priority levels (from P1 “Change absolutely required” to P5 “Change highly optional”). Rule priority guidelines for default and custom-made warnings can be found in the PMD project documentation.31

SonarQube LTS 6.7.7 detects a total of 413 warnings which are grouped based on type and severity. SonarQube categorizes the 413 warnings under 3 types: Bugs, Code Smells, and Vulnerabilities. Of these 413 warnings, 107 warnings are classified as Bugs, 272 as Code Smells, and 34 as Vulnerabilities. In addition to the types, SonarQube groups the warnings under 5 severity types:

31https://pmd.github.io/latest/
Static Analysis Tools Warnings Detected in our projects. We obtained results only for 47 projects out of the 112 contained in the Qualitas Corpus dataset, applying the warnings 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.

In total, the projects were infected by 936 warnings violated 13,554,762 times. 8 (out of 10) warnings were detected by Better Code Hub 27,888 times,

---

### Table 1 RQ1. Rule diffusion across the 47 projects.

| Tool          | #occurrences | #types | #severity |
|---------------|--------------|--------|-----------|
| Better Code Hub | 13,554,762   | 3      | 3         |
| Checkstyle     | 530,068      | 3      | 3         |
| Coverity Scan  | 4,043,455    | 3      | 3         |
| Findbugs       | 7,431,846    | 7      | 3         |
| PMD            | 3,380,493    | 5      | 3         |
| SonarQube      | 5,099,602    | 5      | 3         |

**Static Analysis Tools Warnings Detected in our projects.** We obtained results only for 47 projects out of the 112 contained in the Qualitas Corpus dataset, applying the warnings 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 2 RQ1. Detection Capability and Detected Warnings in the 47 projects.

| Tool            | Detection Capability | Detected warning |
|-----------------|----------------------|------------------|
| Better Code Hub |                      |                  |
| Checkstyle      |                      |                  |
| Coverity Scan   |                      |                  |
| Findbugs        |                      |                  |
| PMD             |                      |                  |
| SonarQube       |                      |                  |

In total, the projects were infected by 936 warnings violated 13,554,762 times. 8 (out of 10) warnings were detected by Better Code Hub 27,888 times,
### RQ1. The top-10 warnings detected by Better Code Hub, Checkstyle, Coverity Scan, FindBugs, PMD, and SonarQube.

| Id | Better Code Hub Detected Rule | # |
|----|--------------------------------|---|
| 1  | WRITE _ CLEAN _ CODE            | 16,055 |
| 2  | WRITE _ CODE _ ONCE             | 14,692 |
| 3  | WRITE _ SHORT _ UNITS           | 6,510  |
| 4  | AUTOMATE _ TESTS                | 6,475  |
| 5  | WRITE _ SIMPLE _ UNITS          | 6,302  |
| 6  | SMALL _ UNIT _ INTERFACES       | 6,152  |
| 7  | SEPARATE _ CONCERNS _ IN _ MODULES | 5,880  |
| 8  | COUPLE _ ARCHITECTURE _ COMPONENTS _ LOOSELY | 2,807  |

| Id | Checkstyle Detected Rule | # |
|----|--------------------------|---|
| 1  | IndentationCheck         | 3,997,581 |
| 2  | FileTabCharacterCheck    | 2,496,876 |
| 3  | WhitespaceAroundCheck    | 865,339 |
| 4  | LeftCurlyCheck           | 737,512 |
| 5  | LineLengthCheck          | 763,429 |
| 6  | RegexpSinglelineCheck    | 590,029 |
| 7  | FinalParameterCheck      | 406,331 |
| 8  | ParenPadCheck            | 333,007 |
| 9  | NeedBraceCheck           | 245,110 |
| 10 | MagicNumberCheck         | 223,598 |

| Id | Coverity Scan Detected Rule | # |
|----|-----------------------------|---|
| 1  | Dereference null return value | 1,360 |
| 2  | Explicit null dereferenced  | 514  |
| 3  | Dereference after null check| 334  |
| 4  | Resource leak on an exceptional path | 494 |
| 5  | Resource leak               | 301  |
| 6  | Missing call to superclass  | 242  |
| 7  | Incorrect definition of serializable class | 224 |

| Id | FindBugs Detected Rule | # |
|----|------------------------|---|
| 1  | BC_UNCONFIRMED_CAST    | 2,840 |
| 2  | DM_DEFAULT_CTOR        | 2,557 |
| 3  | BC_UNCONFIRMED_CAST_OF_RETURN_VALUE | 2,424 |
| 4  | DM_DEFAULT_ENCODING    | 1,946 |
| 5  | RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE | 1,544 |
| 6  | DM_DEAD_LOCALSTORE     | 1,281 |
| 7  | DM_PP_NUMBER_CTOR      | 959  |
| 8  | SE_NO_SERIALVERSIONID  | 944  |
| 9  | REC_CATCH_EXCEPTION    | 887  |
| 10 | SE_BAD_FIELD           | 879  |

| Id | PMD Detected Rule | # |
|----|-------------------|---|
| 1  | LawOfDemeter      | 505,947 |
| 2  | MethodArgumentCouldBeFinal | 374,159 |
| 3  | CommentRequired   | 368,177 |
| 4  | LocalVariableCouldBeFinal | 341,240 |
| 5  | DataflowAnomalyAnalysis | 153,464 |
| 6  | ShortVariable     | 152,681 |
| 7  | UselessParentheses | 136,162 |
| 8  | BeanMembersShouldSerialize | 128,682 |
| 9  | ControlStatementBraces | 111,400 |

| Id | SonarQube Detected Rule | # |
|----|-------------------------|---|
| 1  | S1214                   | 30,888 |
| 2  | S125                    | 30,336 |
| 3  | S00122                  | 26,072 |
| 4  | S00116                  | 25,449 |
| 5  | S00117                  | 23,497 |
| 6  | S1166                   | 21,150 |
| 7  | S106                    | 19,713 |
| 8  | S134                    | 19,508 |
| 9  | S133                    | 17,654 |
| 10 | S1132                   | 13,576 |

88 (out of 173) warnings were detected by Checkstyle 9,686,813 times, 130 warnings were detected by Coverity Scan 7,431 times, 255 (out of 424) warnings were detected by Findbugs 33,704 times, 275 (out of 305) warnings were detected by PMD 3,380,493 times, and 180 (out of 413) warnings were detected by SonarQube 418,433 times (Table 1 and Table 2). It is important to note that in Table 2, 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 warnings were violated by grouping
them based on type and severity. The full results of this additional analysis are reported in our replication package\textsuperscript{30}.

Given the scope of warnings that were detected, our projects were affected by all warnings that are detectable by Better Code Hub and by some warnings that are detectable by Coverity Scan and SonarQube (Table 3). For the sake of readability, we report only the Top-10 warnings detected in our projects by the six tools. The complete list is available in the replication package\textsuperscript{30}.

**Finding 1.** Our analysis provided a mapping of the warnings that currently available tools can identify in software projects. Overall, the amount of warnings detected by the six automated static analysis tools is significant (936 warnings detected 13,554,762 times); hence, we can proceed with the analysis of the remaining RQs.

Table 4 RQ\textsubscript{2}. Rule pairs that overlap at the “class-level”.

| Rule pairs                  | # occurrences | # max occurrences | %   |
|-----------------------------|---------------|-------------------|-----|
| Checkstyle - PMD            | 4,872         | 3,380,493         | 0.144 |
| SonarQube - PMD             | 4,126         | 418,433           | 0.98 |
| Findbugs - PMD              | 3,161         | 33704             | 9.378 |
| SonarQube - Checkstyle      | 1,405         | 418,433           | 0.357 |
| Findbugs - Checkstyle       | 1,265         | 33704             | 3.753 |
| BCH-PMD                     | 1,017         | 27,888            | 3.646 |
| SonarQube - Findbugs        | 849           | 33704             | 2.518 |
| BCH-SonarQube               | 517           | 27,888            | 1.853 |
| BCH-Checkstyle              | 440           | 27,888            | 1.577 |
| BCH-Findbugs                | 205           | 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** |

RQ\textsubscript{2}. What is the agreement among different Static Analysis Tools?

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

**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 warnings detected by the other five tools overlapped at 100% and where at least one warning of Coverity Scan was violated in the same class.

To calculate the percentage of warnings pairs (columns “%”, Table 4) that appear together, we checked the occurrences of both tools in our projects, then we considered only the minimum value. For example, in Table 4, calculating the percentage between Checkstyle - PMD warning pairs, we have 9,686,813 warnings Checkstyle detected and 3,380,493 PMD ones detected. The combi-
nation of these warnings 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”.

| Rule 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%** |

The warnings overlap at the “class-level” is always low, as reported in Table 4. This means that a piece of code violated by a warning detected by one tool is almost never violated by another warning detected by another tool. In the best case (Table 4), only 9.378% of the possible warning (Findbugs-PMD). Moreover, we did find no warnings pair at “class-level” considering more that two tools (e.g. Checkstyle-Findbugs-PMD).

For each warnings 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 9, Table 10, and Table 8) overview the detection agreement of each warning pair, according to the procedure described in Section 2.4. 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. In these tables, the third and fourth columns (e.g. “#BCH pairs” and “# CHS pairs”, Table 9) report how many times a warning instance from a tool exists with another one. The tools have separate values for the number of co-occurrences as the number of instances differs as well, for example, could be that a large rule contains several instances of the comparison rule. Then for other rule this counts as one co-occurrence while for the other rule each included rule grows the number. This makes sure the agreement is between 0 and 1 for both rules. The remaining two columns report the agreement of each tools considered in the warning pairs (e.g. “#BCH Agr.” and “# CHS Agr.”, Table 9). Results showed for all the warning pairs that the agreement at “class-level” is very low, as none of the most recurrent warning pairs agree well. The results also highlighted the difference in the granularity of the warnings.

Agreement based on the overlapping at the “line-level”. Since we cannot compare at “line-level” the warnings detected by Coverity Scan, we could only consider the remaining five static analysis tools. Using the bottom-up approach (Figure 1), several rule pairs were found according to the 100%,
90%, 80%, and 70% thresholds. Using the threshold of 100% which indicates that a rule completely resides within the reference rule, we found 17,977 rule pairs, as reported in Table 5. Using the thresholds of 90%, 80%, and 70% the following rule pairs were found respectively: 17,985, 18,004, and 18,025 (Table 6). These warnings resided partially within the reference rule.

Table 6 RQ2. Rule pairs that overlap at the different thresholds, i.e., 90/80/70%, considering the “line-level”.

| Rule pairs         | 90%          | 80%          | 70%          | possible  |
|--------------------|--------------|--------------|--------------|-----------|
|                    | #(% )        | #(% )        | #(% )        | # occurrences |
| 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-CheckStyle     | 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  |

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 warnings at “line-level” considering more than two tools (e.g. Checkstyle-Findbugs-PMD).

When considering the agreement for each warning pair at “line-level”, we could not obtain any result because of computational reasons. Indeed, the analysis at line-level of 936 warning 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.

**Finding 2.** The warnings overlapping among the different tools is very low (less than 0.4%). The warning 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.

**RQ3. What is the precision of the static analysis tools?**

In the context of our last research question, we focused on the precision of the static analysis tools when employed for TD detection. Table 7 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%.
In general, based on our findings, we can first corroborate previous findings in the field [4, 20, 31] and the observations reported by Johnson et al. [20], 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 static analysis tools in practice. With respect to the qualitative insights obtained by interviewing developers [20], our work concretely quantifies the capabilities of the considered static analysis tools.

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 warnings 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 static analysis tools, namely PMD, Findbugs, and Checkstyle.

As for the former, we noticed that it typically fails when raising warnings related to naming conventions. For instance, this is the case of the ‘Abstract-Name’ warning: 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 warnings were labeled as false positives. In this case, one of the most problematic cases was related to the 'BC_UNCONFIRMED_CAST' warnings: 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 warnings 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 static analysis tool having the highest precision, i.e., 86%. When validating the instances output by the tool, we realized that the warnings 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' warning: 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 warnings 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 static analysis tools.

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 static analysis tools [33, 13] as well as for the automatic derivation of proper thresholds to use when locating the presence of design issues in source code [2, 17]. It might indeed be possible that the integration of those approaches into the inner workings of the currently available static analysis tools could lead to a reduction of the number of false positive items. In addition, our findings also suggest that the current static analysis tools 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 3. Most of the considered SATs suffer from a high number of false positive warnings, and their precision ranges between 18% and 57%. The only exception is Checkstyle (precision=86%), even though most of the warnings it raises are related to documentation issues rather than functional problems and, as such, its adoption should be complemented with other static analysis tools.

4 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 RQ1, different SAT warnings 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, e.g., ensemble methods or meta-models [10, 12, 11, 37], that can (1) combine the results from different static analysis
tools 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 static analysis tools in terms of precision of the recommendations provided to developers (RQ3). 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. [20], hence confirming that the real usefulness of static analysis tools 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 [16] and how to select/prioritize the warnings to be presented to developers [21, 28, 23]. 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 warnings [36] in order to augment the design of existing tools and the way they present warnings to developers.

Complementing static analysis tools. The findings from our RQ1 and RQ2 highlight that most of the issues reported by the state-of-the-art 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 static analysis 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 static analysis tools 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.

5 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 [47], which
is one of the most reliable data sources in software engineering research [46]. Another possible threat relates to the configuration of the SATs employed. None of the considered projects had all the static analysis tools configured and so we had to manually introduce them; in doing so, we relied on the default configuration of the tools since we could not rely on different configurations given directly by the developers of the projects. 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 static analysis tools employed to check for the presence of duplicates, but we did not find any.

**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 static analysis tools. Nevertheless, open-source projects are comparable—in terms of source code quality—to closed-source or industrial applications [26]; 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 RQ1 and RQ3, we conducted iterative manual analyses in order to build the taxonomy and 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 RQ2 we proceeded with an automatic mechanism to study the agreement among the tools. As explained in Section 2.3, different static analysis
tools might possibly output the same warnings 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 warnings 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 RQ2.

6 Related Work

Automated Static Analysis Tools (ASATs) are getting more popular [52, 25] as they are becoming easier to use [55]. The use of static analysis tools has been studied by several researchers in the last years [53, 34, 56, 35]. In this section, we report the relevant work on static analysis tools focusing on their usage [42, 24, 27], warnings and the detected problems [15, 19, 8].

Developers can use ASATs, such as SonarQube3 and CheckStyle32, to evaluate software source code, finding anomalies of various kinds in the code [40, 50]. Moreover, ASATs are widely adopted in many research studies in order to evaluate the code quality [20, 44, 30] and identify issues in the code [42, 24, 27]. Some studies demonstrated that some rules detected by ASATs can be effective for identifying issues in the code [56, 23, 27]. However, evaluating the performance in defect prediction, results are discordant comparing different tools (e.g. FindBugs35 and PMD37) [38].

Rutar et al. [40] compared five bug-finding tools for Java (Bandera33, ESC/Java234, FindBugs35, JLint36, and PMD37), that use syntactic bug pattern detection, on five projects, including JBoss 3.2.338 and Apache Tomcat 5.0.1939. They focused on the different warnings (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 warnings provided by the different tools are not correlated with

---

32 https://checkstyle.sourceforge.io/
33 http://bandera.projects.cs.ksu.edu/
34 https://kindsoftware.com/products/opensource/ESCJava2/
35 http://findbugs.sourceforge.net/
36 http://jlint.sourceforge.net/
37 https://pmd.github.io/
38 http://www.jboss.org/
39 http://jakarta.apache.org/tomcat/
each other. Complementing the work by Rutar et al. [40], we calculated the agreement of ASATs on TD identification. In addition, we investigated the precision with which these tools output warnings. Finally, we also investigated the types of TD items that can actually be detected by existing ASATs.

Tomas et al. [50] performed a comparative analysis by means of a systematic literature review. In total, they compared 16 Java code static analysis tools, including JDepend\(^40\), Findbugs\(^35\), PMD\(^37\), and SonarQube\(^3\). 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 them automate the calculation of internal quality metrics, the most common ones being code smells, complexity, and code size [50]. However, they did not investigate agreement between the tools’ detection rules.

Avgeriou et al. [5] identified the available static analysis tools 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 usage of static analysis tools, ASATs can help to find bugs [20]. However, developers are not sure about the usefulness of the rules [45, 51, 43], they do pay attention to different rules categories and priorities and remove violations related to rules with high severity [51] in order to avoid the possible risk of faults [45]. Moreover, false positives and the way in which the warnings are presented, among other things, are barriers to their wider adoption [20]. Some studies highlighted the need to reduce the number of detectable rules [32, 9] or summarize them based on similarities [51]. ASATs 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 [49]. 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 [28, 41], many studies deal with the many false positives produced by different tools, such as FindBugs\(^35\) [49, 6, 7], JLint\(^36\), PMD\(^37\), CheckStyle\(^32\), and JCSC\(^41\) [49].

At the best of our knowledge, our work is the first that investigate in details which source quality problems can actually be detected by the available tools, trying to make a comparison based on the description, what is their agreement, and what is the precision of their recommendations.

7 Conclusion

In this paper, we performed a large-scale comparison of six popular Static Analysis Tools (Better Code Hub, CheckStyle, Coverity Scan, Findgugs, PMD,
and SonarQube) with respect to the detection of static analysis warnings. We analyzed 47 Java projects from the Qualitas Corpus dataset, and derived similar warnings that can be detected by the tools. We also compared their detection agreement at “line-level” and “class-level”, and manually analyzed their precision. To sum up, the contributions of this paper are:

1. A comparison of the warnings that can be detected by the tools (taxonomy), which may be useful for researchers and tool vendors to understand which warnings should be considered during refactoring;
2. An analysis of the agreement among the tools, which can inform tool vendors about the limitations of the current solutions available the market;
3. The first quantification of the precision of six static analysis tools (Better Code Hub, CheckStyle, Coverity Scan, Findbugs, PMD, and SonarQube).

Our future work includes an extension of this study with the evaluation of the recall, and the in-vivo assessment of the tools.

References

1. Al Mamun MA, Khanam A, Grahn H, Feldt R (2010) Comparing four static analysis tools for java concurrency bugs. In: Third Swedish Workshop on Multi-Core Computing (MCC-10)
2. Aniche M, Treude C, Zaidman A, Van Deursen A, Gerosa MA (2016) Satt: Tailoring code metric thresholds for different software architectures. In: International Working Conference on Source Code Analysis and Manipulation (SCAM), pp 41–50
3. Antoine JY, Villaneau J, Lefeuvre A (2014) 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”, pp 550–559
4. Antunes N, Vieira M (2009) 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
5. Avgeriou P, Taibi D, Ampatzoglou A, Arcelli Fontana F, Besker T, Chatzigeorgiou A, Lenarduzzi V, Martini A, Moschou N, Pigazzini I, Saarimäki N, Sas D, Soares de Toledo S, Tsintzira A (2021) An overview and comparison of technical debt measurement tools. IEEE Software
6. Ayewah N, Pugh W (2010) The google findbugs fixit. In: 19th International Symposium on Software Testing and Analysis, p 241–252
7. Ayewah N, Hovemeyer D, Morgenthaler JD, Penix J, Pugh W (2008) Using static analysis to find bugs. IEEE Softw 25:22–29
8. Beller M, Bholanath R, McIntosh S, Zaidman A (2016) 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), vol 1, pp 470–481
9. Bodden E (2018) Self-adaptive static analysis. In: 40th International Conference on Software Engineering: New Ideas and Emerging Results, p 45–48
10. Catolino G, Ferrucci F (2019) An extensive evaluation of ensemble techniques for software change prediction. Journal of Software: Evolution and Process p e2156
11. Catolino G, Palomba F, De Lucia A, Ferrucci F, Zaidman A (2018) Enhancing change prediction models using developer-related factors. Journal of Systems and Software 143:14–28
12. Catolino G, Palomba F, Fontana Arcelli F, De Lucia A, Zaidman A, Ferrucci F (2019) Improving change prediction models with code smell-related information. arXiv preprint arXiv:190510889
13. Di Nucci D, Palomba F, Oliveto R, De Lucia A (2017) Dynamic selection of classifiers in bug prediction: An adaptive method. Transactions on Emerging Topics in Computational Intelligence 1(3):202–212
14. Ernst NA, Bellomo S, Ozkaya I, Nord RL, Gorton I (2015) Measure it? Manage it? Ignore it? Software practitioners and technical debt. Symposium on the Foundations of Software Engineering pp 50–60
15. Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R (2002) Extended static checking for java. In: Conference on Programming Language Design and Implementation, p 234–245
16. Fontana Arcelli F, Ferme V, Zanoni M (2015) Filtering code smells detection results. In: International Conference on Software Engineering-Volume 2, pp 803–804
17. Fontana Arcelli F, Ferme V, Zanoni M, Yamashita A (2015) Automatic metric thresholds derivation for code smell detection. In: International workshop on emerging trends in software metrics, pp 44–53
18. Fowler M, Beck K (1999) Refactoring: Improving the design of existing code. Addison-Wesley Longman Publishing Co, Inc
19. Heckman S, Williams L (2011) A systematic literature review of actionable alert identification techniques for automated static code analysis. Information and Software Technology 53(4):363 – 387, special section: Software Engineering track of the 24th Annual Symposium on Applied Computing
20. Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don’t software developers use static analysis tools to find bugs? In: 2013 35th International Conference on Software Engineering (ICSE), IEEE, pp 672–681
21. Kim S, Ernst MD (2007) Which warnings should i fix first? In: Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ACM, pp 45–54
22. Krippendorff K (2018) Content analysis: An introduction to its methodology. Sage publications
23. Lenarduzzi V, Lomio F, Huttunen H, Taibi D (2019) Are sonarqube rules inducing bugs? International Conference on Software Analysis, Evolution and Reengineering (SANER) Preprint: arXiv:190700376
24. Lenarduzzi V, Martini A, Taibi D, Tamburri DA (2019) Towards surgically-precise technical debt estimation: Early results and research
roadmap. In: International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE 2019, pp 37–42
25. Lenarduzzi V, Sillitti A, Taibi D (2019) A survey on code analysis tools for software maintenance prediction. In: Software Engineering for Defence Applications - SEDA 2018, Springer-Verlag, Advances in Intelligent Systems and Computing (AISC), vol 925
26. Lenarduzzi V, Tosi D, Lavazza L, Morasca S (2019) Why do developers adopt open source software? past, present and future. In: Open Source Systems, Springer International Publishing, pp 104–115
27. Lenarduzzi V, Saarimäki N, Taibi D (2020) Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study. Journal of Systems and Software 170
28. Liang G, Wu L, Wu Q, Wang Q, Xie T, Mei H (2010) Automatic construction of an effective training set for prioritizing static analysis warnings. In: Int. Conf. on Automated software engineering, pp 93–102
29. Mantere M, Uusitalo I, Roning J (2009) Comparison of static code analysis tools. In: Int. Conf. on Emerging Security Information, Systems and Technologies, pp 15–22
30. Marcilio D, Bonifácio R, Monteiro E, Canedo E, Luz W, Pinto G (2019) Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In: 27th International Conference on Program Comprehension, p 209–219
31. McLean RK (2012) Comparing static security analysis tools using open source software. In: Int. Conf. on Software Security and Reliability, pp 68–74
32. Muske T, Talluri R, Serebrenik A (2018) Repositioning of static analysis alarms. In: 27th International Symposium on Software Testing and Analysis, p 187–197
33. Nadi S, Berger T, Kästner C, Czarnecki K (2014) Mining configuration constraints: Static analyses and empirical results. In: International Conference on Software Engineering, ACM, pp 140–151
34. Nagappan N, Ball T (2005) Static analysis tools as early indicators of pre-release defect density. In: 27th International Conference on Software Engineering (ICSE), pp 580–586
35. Nanda MG, Gupta M, Sinha S, Chandra S, Schmidt D, Balachandran P (2010) Making defect-finding tools work for you. In: 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, p 99–108
36. Oliver PE, Marwell G (2001) Whatever happened to critical mass theory? a retrospective and assessment. Sociological Theory 19(3):292–311
37. Palomba F, Zanoni M, Fontana Arcelli F, De Lucia A, Oliveto R (2017) Toward a smell-aware bug prediction model. Transactions on Software Engineering 45(2):194–218
38. Rahman F, Khatri S, Barr ET, Devanbu P (2014) Comparing static bug finders and statistical prediction. In: 36th International Conference on Software Engineering, p 424–434
39. Runeson P, Host M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical Softw Engg 14(2):131–164
40. Rutar N, Almazan CB, Foster JS (2004) A comparison of bug finding tools for java. In: Symposium on Software Reliability Engineering, pp 245–256
41. Ruthruff JR, Penix J, Morgenthaler JD, Elbaum S, Rothermel G (2008) Predicting accurate and actionable static analysis warnings: An experimental approach. In: 30th International Conference on Software Engineering, p 341–350
42. Saarimäki N, Lenarduzzi V, Taibi D (2019) On the diffuseness of code technical debt in open source projects. In: International Conference on Technical Debt (TechDebt 2019)
43. Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C (2018) Lessons from building static analysis tools at google. Commun ACM 61(4):58–66
44. Schnappinger M, Osman MH, Pretschner A, Fietzke A (2019) Learning a classifier for prediction of maintainability based on static analysis tools. In: 27th International Conference on Program Comprehension, p 243–248
45. Taibi D, Janes A, Lenarduzzi V (2017) How developers perceive smells in source code: A replicated study. Information and Software Technology 92:223–235
46. Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) The qualitas corpus: A curated collection of java code for empirical studies. Asia Pacific Software Engineering Conference pp 336–345
47. Terra RM, Miranda LF, Valente MT, da Silva Bigonha R (2013) Qualitas.class corpus: a compiled version of the qualitas corpus. ACM SIGSOFT Software Engineering Notes 38:1–4
48. Thomas TW, Lipford H, Chu B, Smith J, Murphy-Hill E (2016) What questions remain? an examination of how developers understand an interactive static analysis tool. In: Symposium on Usable Privacy and Security
49. Thung F, Lucia, Lo D, Jiang L, Rahman F, Devanbu PT (2015) To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools. Automated Software Engg 22(4):561–602
50. Tomas P, Escalona MJ, Mejias M (2013) Open source tools for measuring the Internal Quality of Java software products. A survey. Computer Standards and Interfaces 36(1):244–255
51. Vassallo C, Panichella S, Palomba F, Proksch S, Zaidman A, Gall HC (2018) Context is king: The developer perspective on the usage of static analysis tools. In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 38–49
52. Vassallo C, Panichella S, Palomba F, Proksch S, Gall HC, Zaidman A (2019) How developers engage with static analysis tools in different contexts. Empirical Software Engineering pp 1–39
53. Wagner S, Jürjens J, Koller C, Trischberger P (2005) Comparing bug finding tools with reviews and tests. In: International Conference on Testing of Communicating Systems, p 40–55

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

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

56. Zheng J, Williams L, Nagappan N, Snipes W, Hudepohl JP, Vouk MA (2006) On the value of static analysis for fault detection in software. IEEE Transactions on Software Engineering 32(4):240–253
Appendix A

In the following, we report more results achieved in the context of RQ\textsubscript{2}. For each warnings pair we computed the detection agreement at class level as reported in Table 9, Table 10, and Table 8, according to the process described in Section 2.4. 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\textsuperscript{30}.

Table 8 RQ\textsubscript{2}. The 10 most recurrent rule pairs detected in the same class by the considered SATs and their corresponding agreement values.

| SonarQube | FindBugs | \# SQ pairs | \# FB pairs | SQ Agr. | FB Agr. |
|-----------|----------|-------------|-------------|---------|---------|
| CommentedOutCodeLine | RV\_RETURN\_VALUE\_OF\_PUTIFABSENT\_JG-NORED | 2 | 1 | 0.000 | 1.000 |
| S1155 | RV\_RETURN\_VALUE\_OF\_PUTIFABSENT\_JG-NORED | 1 | 1 | 0.000 | 1.000 |
| S1186 | RV\_RETURN\_VALUE\_OF\_PUTIFABSENT\_JG-NORED | 1 | 1 | 0.000 | 1.000 |
| S135 | RV\_RETURN\_VALUE\_OF\_PUTIFABSENT\_JG-NORED | 3 | 1 | 0.001 | 1.000 |
| S1312 | RV\_RETURN\_VALUE\_OF\_PUTIFABSENT\_JG-NORED | 1 | 1 | 0.000 | 1.000 |
| complex\_class | RV\_RETURN\_VALUE\_OF\_PUTIFABSENT\_JG-NORED | 1 | 1 | 0.000 | 1.000 |
| S1195 | DM\_DEFAULT\_ENCODING | 1 | 1 | 1.000 | 0.001 |
| S1195 | EJ\_EXPOSE\_REG | 1 | 1 | 1.000 | 0.001 |
| S1301 | LG\_LOST\_LOGGER\_DUE\_TO\_WEAK\_REFERENCE | 1 | 1 | 0.001 | 1.000 |
| S1148 | FJ\_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 | AvoidCatchingGenericException | 1 | 1 | 1.000 | 0.001 |
| 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 |

\textit{BCH} means Better Code Hub, and \textit{SQ} means SonarQube, while \textit{Ag.} means Agreement.
### Table 9 RQ2. The 10 most recurrent rule pairs detected in the same class by the considered SATs and their corresponding agreement values.

| 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 | 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 | 0.506 |
| WRITE_CODE_ONCE | IllegalTokenTextCheck | 2 | 418 | 0.000 | 0.506 |
| WRITE_CODE_ONCE | AtclauseOrderCheck | 30 | 210 | 0.002 | 0.506 |

| BetterCodeHub | CoverageScan | # BCH pairs | # CHS pairs | BCH Agr. | CHS 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 | 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 | # PMD pairs | BCH Agr. | PMD Agr. |
|---------------|----------|-------------|-------------|----------|----------|
| WRITE_CODE_ONCE | ICAST_BAD_SHIFT_AMOUNT | 12 | 8 | 0.001 | 1.000 |
| WRITE_SIMPLE_UNITS | LOCAL_STATIC_ASSIGNMENT_IN_SAME_FIELD | 1 | 1 | 0.000 | 1.000 |
| AUTOMATE_TESTS | FJ_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 | VACUOUS_OPERATION | 4 | 1 | 0.001 | 1.000 |
| WRITE_SHORT_UNITS | RETURN_VALUE_OF_PUTIFABSENT_IGNORED | 4 | 1 | 0.001 | 1.000 |
| SEPARATE_CONCERNS_IN_MODULES | RETURN_VALUE_OF_PUTIFABSENT_IGNORED | 2 | 1 | 0.000 | 1.000 |
| AUTOMATE_TESTS | NP_ALWAYS_NULL_EXCEPTION | 2 | 1 | 0.000 | 1.000 |
| WRITE_CLEAN_CODE | 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 | 3 | 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 | 1 | 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 | 2 | 160 | 0.000 | 0.748 |

| CheckStyle | CoverageScan | # BCH pairs | # CHS pairs | BCH Agr. | CHS 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 |

| CheckStyle | FindBugs | # BCH pairs | # CHS pairs | BCH Agr. | CHS Agr. |
|------------|----------|-------------|-------------|----------|----------|
| AvoidStarImportCheck | SA_FIELD SELF_COMPUTATION | 8 | 2 | 0.000 | 1.000 |
| FinalParameterCheck | IP: Ignored parameter | 3 | 1 | 0.000 | 1.000 |
| RedundantModifierCheck | IC_SUPERCLASS_USES_SUBCLASS DURING_INITIALIZATION | 1 | 1 | 0.000 | 1.000 |
| DesignForExtensionCheck | RETURN_VALUE_OF_PUTIFABSENT_IGNORED | 8 | 1 | 0.000 | 1.000 |
| NeedBraceCheck | TV:EXPLICIT_UNKNOWN_SOURCE_VALUE_REACHES_NULL_VALUE | 28 | 1 | 0.000 | 1.000 |
| JavadocVariableCheck | IC_SUPERCLASS_USES_SUBCLASS DURING_INITIALIZATION | 1 | 1 | 0.000 | 1.000 |
| JavadocVariableCheck | IC_SUPERCLASS_USES_SUBCLASS DURING_INITIALIZATION | 1 | 1 | 0.000 | 1.000 |
| WhitespaceAroundCheck | B1:MISSING_SUPER_CALL | 149 | 1 | 0.001 | 1.000 |
Table 10 RQ2. The 10 most recent rule pairs detected in the same class by the considered SATs and their corresponding agreement values.

| CheckStyle | PMD | # CHS pairs | # PMD Agr. | # FB pairs | FB Agr. |
|------------|-----|--------------|------------|------------|---------|
| FinalParametersCheck | AvoidMultipleUnaryOperators | 51 | 2 | 1.000 | 1.000 |
| RegexpPairingCheck | AvoidMultipleUnaryOperators | 75 | 2 | 0.000 | 1.000 |
| NoFinalizerCheck | FinalizeOnlyCallsSuperFinalize | 1 | 1 | 0.015 | 1.000 |
| ViabilityModifierCheck | 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 | 0.000 | 1.000 |
| NeedBraceCheck | AvoidMultipleUnaryOperators | 90 | 2 | 0.000 | 1.000 |
| JavadocParagraphCheck | FinalizeOnlyCallsSuperFinalize | 1 | 1 | 0.000 | 1.000 |
| NonEmptyAnd/orDescriptionCheck | FinalizeOnlyCallsSuperFinalize | 10 | 1 | 0.000 | 1.000 |

| CoverityScan | FindBugs | # CS pairs | # PMD Agr. | # FB pairs | FB Agr. |
|--------------|----------|------------|------------|------------|---------|
| UCF: Useless control flow | BC_UNCONFIRMED| 1 | 2 | 1.000 | 0.000 |
| Dead default in switch | DLS| 1 | 5 | 1.000 | 0.000 |
| DLS: Dead local store | WP| 3 | 1 | 0.010 | 1.000 |
| UCF: Useless control flow | BC_UNCONFIRMED| 1 | 1 | 1.000 | 0.000 |
| Unused reflection | NPL: NonPublicLocal | 2 | 2 | 0.000 | 0.012 |
| UCF: Useless control flow | UCF_USELESS_CONTROLFLOW | 1 | 1 | 1.000 | 0.111 |
| OMGN injection | BC_REDUNDANT_NULLCHECK | 1 | 1 | 1.000 | 0.001 |
| Failure to call super.finalize() | NFC_NULLIFY | 1 | 1 | 0.500 | 1.000 |
| REC: RuntimeException capture | RV_RETURN_VALUE | 1 | 1 | 0.013 | 1.000 |
| USELESS_STRING | Useless/non-informative string | 5 | 1 | 0.250 | 1.000 |

| CoverityScan | PMD | # CS Agr. | # PMD Agr. | # FB Agr. | FB Agr. |
|--------------|-----|-----------|------------|----------|--------|
| 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 | UseIncorrectExceptionLogging | 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 | UnnecessaryFullyQualifiedClassName | 1 | 21 | 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 | 0.000 |
| 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 |

| SonarQube | BetterCodeHub | # SQ Agr. | # CHS Agr. | # BCH Agr. |
|-----------|---------------|----------|-----------|----------|
| 2225 | COUPLEARCHITECTURECOMPONENTS_LOOSELY | 1 | 2 | 1.000 | 0.001 |
| 2227 | AUTOMATE_TESTS | 1 | 2 | 1.000 | 0.000 |
| 8888 | COUPLEARCHITECTURECOMPONENTS_LOOSELY | 2 | 4 | 0.667 | 0.001 |
| 8888 | WRITECLEancode | 2 | 10 | 0.667 | 0.001 |
| 8888 | WRITECODEONCE | 2 | 26 | 0.667 | 0.002 |
| 8888 | AUTOMATE_TESTS | 2 | 2 | 0.667 | 0.000 |
| 8888 | SEPARATECONCERNSINMODULES | 3 | 3 | 0.500 | 0.000 |
| 8888 | AUTOMATE_TESTS | 3 | 3 | 0.500 | 0.000 |
| 8888 | AUTOMATE_TESTS | 1 | 1 | 0.500 | 0.000 |

| SonarQube | CheckStyle | # SQ Agr. | # CHS Agr. |
|-----------|-----------|----------|-----------|
| 2225 | CommentL duplicatedCheck | 1 | 1 | 1.000 | 0.000 |
| 2220 | JobStatusCheck | 2 | 12 | 1.000 | 0.000 |
| 2213 | RedundantModifierCheck | 2 | 56 | 1.000 | 0.002 |
| 2213 | InvaludHostNameCheck | 2 | 4 | 1.000 | 0.000 |
| 2213 | RegexpPairingCheck | 2 | 42 | 1.000 | 0.000 |
| 2213 | WorkspaceAroundCheck | 2 | 22 | 1.000 | 0.000 |
| 2200 | EqualHashCodeCheck | 2 | 1 | 1.000 | 0.000 |
| 2200 | FileTailCharacterCheck | 2 | 8 | 1.000 | 0.000 |
| 2200 | FinalParameterCheck | 2 | 1 | 1.000 | 0.000 |
| 2200 | IndentationCheck | 2 | 1 | 1.000 | 0.000 |

| SonarQube | CoverityScan | # CS Agr. | # CHS Agr. |
|-----------|-------------|----------|-----------|
| 1244 | Unexpected control flow | 2 | 2 | 0.001 | 1.000 |
| 11125 | Use of hardcoded cryptographic key | 1 | 1 | 0.005 | 1.000 |
| 11126 | Dead default in switch | 4 | 1 | 0.001 | 1.000 |
| 11132 | Dead default in switch | 2 | 1 | 0.001 | 1.000 |
| 11149 | Dead default in switch | 1 | 1 | 0.000 | 1.000 |
| 11151 | Dead default in switch | 8 | 1 | 0.001 | 1.000 |
| 11172 | Dead default in switch | 4 | 1 | 0.002 | 1.000 |
| 12123 | Dead default in switch | 26 | 1 | 0.001 | 1.000 |
| 12126 | Dead default in switch | 7 | 1 | 0.001 | 1.000 |