Analyzing the State of Static Analysis
A Large-Scale Evaluation in Open Source Software

Beller, Moritz; Bholanath, Radjino; McIntosh, Shane; Zaidman, Andy

DOI
10.1109/SANER.2016.105

Publication date
2016

Document Version
Accepted author manuscript

Published in
Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering

Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software

Moritz Beller, Radjino Bholanath
Delft University of Technology
The Netherlands
m.m.beller@tudelft.nl
radjino.bholanath@gmail.com

Shane McIntosh
McGill University
Canada
shane.mcintosh@mcgill.ca

Andy Zaidman
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

Abstract—**The use of automatic static analysis has been a software engineering best practice for decades. However, we still do not know a lot about its use in real-world software projects: How prevalent is the use of Automated Static Analysis Tools (ASATs) such as FindBugs and JSHint? How do developers use these tools, and how does their use evolve over time? We research these questions in two studies on nine different ASATs for Java, JavaScript, Ruby, and Python with a population of 122 and 168,214 open-source projects. To compare warnings across the ASATs, we introduce the General Defect Classification (GDC) and provide a grounded-theory-derived mapping of 1,825 ASAT-specific warnings to 16 top-level GDC classes. Our results show that ASAT use is widespread, but not ubiquitous, and that projects typically do not enforce a strict policy on ASAT use. Most ASAT configurations deviate slightly from the default, but hardly any introduce new custom analyses. Only a very small set of default ASAT analyses is widely changed. Finally, most ASAT configurations, once introduced, never change. If they do, the changes are small and have a tendency to occur within one day of the configuration's initial introduction.**

I. Introduction

Automated Static Analysis Tools (ASATs) scan the source or binary code of a software system for a set of pre-defined problems. ASATs can be configured to detect: (1) *functional* problems, such as resource leakage or incorrect logic; and (2) *maintainability* problems, such as non-compliance with best practices or violations of style conventions.

Next to testing and manual code review, ASATs have become an important pillar of modern Software Quality Assurance approaches. By heeding the warnings that are reported from ASATs, development teams can address problems before they escape into released versions of their software. Coding standards like NASA's JPL C [1] and Java [2] standards mandate the use of ASATs during the development process; stronger still, they require that crucial software components be free of any ASAT warning.

However, aside from anecdotal evidence, little is currently known about whether and how rigorously developers use ASATs in the ecosystem of Open-Source Software (OSS). A deeper understanding of the real world application of ASATs can guide researchers' future work, help ASAT developers adapt their offerings to better fit their user base, and ultimately improve the user experience of ASATs.

In this paper, we set out to understand the prevalence of ASATs, their configuration in real software systems, and how those configurations evolve. To study the prevalence of ASATs, we quantitatively and qualitatively analyze 122 popular OSS projects from the GitHub, OpenHub, SourceForge, and Gitorious software forges in search of the use of 9 popular ASATs. In a second study on the configuration and evolution of ASATs, we: (1) produce a *General Defect Classification* (GDC) in order to map the specific problems that are detected by the 9 studied ASATs to a common format; and (2) analyze how 168,214 OSS projects configure the studied ASATs with respect to the GDC. We address the following broad research questions:

RQ 1 *How common is the use of ASATs in practice?*
Half of the state-of-the-art OSS projects already employ automated static analysis, although they typically use only one ASAT in an ad-hoc fashion, where the ASAT is not integrated with the flow of development.

RQ 2 *How are ASATs configured?*
The ASAT configurations in the studied OSS projects barely deviate from the default ASAT configuration and rarely introduce custom checks.

RQ 3 *How does the use of ASATs evolve?*
Most ASAT configurations, once committed, never change. The ASAT configurations that do change are typically only very slightly modified within the first week of their appearance in the studied repositories.

The remainder of this paper is structured as follows. Section II situates this study with respect to the literature on ASATs. Section III provides the rationale for our research questions. Section IV presents the results of our prevalence study (RQ 1). Section V provides an overview of our GDC, while Section VI leverages this classification to analyze ASAT configuration (RQ 2) and evolution (RQ 3). Section VII discusses the broader implications of our results and, finally, Section VIII draws conclusions.

II. Related Work

We first review existing research on ASATs and discuss how it is related to our study of the prevalence and the use of ASATs. Finally, we give an overview of the classifications that the GDC builds upon.

A. Automatic Static Analysis Tools

ASATs traditionally use techniques such as data-flow analysis and control-flow analysis to find defects in source code [3]–[5]. However, because these techniques do not scale at large, abstractions have to be introduced [4]. These abstractions, plus the fact that checking common properties of programs is an undecidable problem [6], lead to *false positives*, warnings about defects that do not exist, and *false negatives*, when warnings about actual defects are missing.

While false negatives impact the efficiency of ASATs because defects are missed, false positives cause developers to waste time investigating incorrect warnings in the code. Deciding whether a warning is a real defect or a false positive takes three to eight minutes on average [7]–[9]. As there can be as many as 50 false positives for every accurate warning [10], analyzing warnings is a time-consuming activity. In general, there are roughly 40 warnings for every thousand lines of code [11]. This overload of warnings is a prime reason for developers to avoid using ASATs [12]. While researchers have studied the reasons why developers do or do not use ASATs, there is little data on the prevalence of ASATs in practice. In this study, we therefore quantitatively investigate the state of adoption of ASATs in OSS projects.

Many ASATs differ in the type of defects that they detect. However, even when tools focus on uncovering the same defect type, the variance in defects found is still very large [6], [13]–[15]. These results indicate that using several ASATs has benefits over using just a single ASAT. However, this increases the number of warnings that developers need to investigate. Thus, deciding to use multiple ASATs is striking a balance between an improved defect detection rate and the additional investigation effort of an increased number of warnings. We aim to determine how common the use of multiple ASATs is.

To better deal with a large number of warnings, studies have investigated ways to prioritize them [9], [11], [16]–[18]. This has the advantage that a developer can decide how many warnings to analyze based on the importance of the warnings. In lieu of those ranking algorithms, developers can use configuration files to indicate which rules they consider important. This can reduce the number of warnings generated and suppress rules that are prone to emitting false positives. Another reason to study developer preferences is to observe if the use of ASATs reflects their potential. Wedyan et al. and others observed that 15% of all detected defects were functional and the rest maintainability-related [15], [19], [20]. Many studies observed that ASATs rarely find any functional defects [15], [20]–[23]. In this paper, we analyze a large number of ASAT configuration files to see how these previous observations are reflected in them.

B. Defect Classifications

In 1993, the IEEE published a standard for classifying defects [24]. It served as the basis for IBM's Orthogonal Defect Classification (ODC) scheme [25]. This scheme uses the defect type as one of the aspects from which to classify the defect. While the ODC scheme has been used in research [26], [27], several studies conclude that it was too abstract and required adaptations to fit any particular use in practice [14], [28], [29]. In this paper, we propose the General Defect Classification (GDC), a remote ODC-descendant that is a generalization of the scheme refined by Beller et al. [28] and Mäntylä and Lassenius [29]. Its ancestry can be traced back to the work of El Emam et al. [30]. Central to this genealogy of classifications is their high inter-rater reliability. The GDC, in contrast to its predecessors, is specifically tailored to reason across multiple ASATs.

III. Research Questions

The goal of this paper is to *increase our understanding of how static analysis tools are used in the real-world.* To that end, we study a large collection of OSS projects from both statically (Java) and dynamically typed languages (JavaScript, Ruby, and Python) in professional and non-professional popular OSS settings.

In pursuing our goal, we must first establish a baseline of how widely-used static analysis is in these projects. Hence, in our first research question, we ask:

RQ 1 How common is the use of ASATs in practice?

We refine the research question into three sub-research questions that we manually investigate:

RQ 1.1 What is the prevalence of ASATs?

RQ 1.2 How common is the simultaneous use of multiple ASATs?

RQ 1.3 Do projects enforce ASAT use?

Having gained insight into how widespread the use of ASATs is through manual analysis, we set out to study how a large corpus of projects use ASATs by automatically investigating the ASAT configuration files in their repositories:

RQ 2 How are ASATs configured?

The answer to this this research question can be important for the creators of ASATs and their users: Coming up with sensible defaults for software is a hard problem [31]. A large-scale study of their user base could help ASAT developers uncover if their defaults generally fit the tool's use in practice so that users spend less time configuring their ASATs. To this end, we want to gain insight specifically into the following sub-research questions:

RQ 2.1 What type of warnings are explicitly enabled?

RQ 2.2 What type of warnings are explicitly disabled?

RQ 2.3 How well do default configurations reflect real-world configurations?

RQ 2.4 How prevalent are custom rules in the OSS configurations?

Finally, in order to understand which role ASATs take in the development process, and if and how their configurations files change over a project's lifetime, we ask:

RQ 3 How does the use of ASATs evolve?

Specifically, we answer the following sub-RQs:

RQ 3.1 How often do ASAT configurations change?

RQ 3.2 How much do ASAT configurations change?

RQ 3.3 When do ASAT configurations change?

IV. Prevalence Analysis (RQ 1)

In this section, we address the question of how wide-spread the use of static analysis is in popular OSS Projects.

A. Methodology

To answer this question, we followed the study design depicted in Figure 1. We started by examining the four popular OSS project hosting platforms GitHub, OpenHub, SourceForge, and Gitorious (Step 1) in December 2014. We also considered other sources, primarily other code hosting services, but found them unsuitable: Some lacked representative projects (for example, on GitLab [32], the most popular repositories belonged to the GitLab company itself), others provided no means of ranking projects by their popularity (for example, the now-defunct Google Code [33]).

Proportional to the number of projects that are hosted on each platform [34], we selected the 100 most popular repositories on GitHub (ranked by number of stars), 20 on Gitorious (ranked by development activity), and 10 from both SourceForge (ranked by number of downloads) and OpenHub (ranked by number of users). In contrast to the other three, OpenHub is not a forge, but a "public directory of free and open source software," which includes links to the project's actual repository. OpenHub's overall popularity ranking was only available for the 10 most popular projects. After eliminating duplicates and non-software repositories, we ended up with 122 unique projects to analyze.

Using a mixed methods approach, we investigated their use of ASATs in two distinct ways (Step 2). On the one hand, in a manual analysis of the projects' websites, contribution guidelines and ASAT configuration files in repositories, we investigated whether and how projects documented their ASAT use (Step 2a). On the other hand, we sent out a short survey to contributors of the same 122 projects, asking which static analysis tools they are using, when they are using them, if it is a necessary precondition to check code before it can enter the main project repository, and whether ASATs are an integral part of their workflow (Step 2b). In order to maximize the number of responses, we sent this question to the projects' mailing lists, newsgroups, or fora, and contacted the two top-committers. We also explicitly lowered the barrier to entry of the survey by embracing a discussion-style answer of developers directly to our informal email [35]. Finally, in Step 3, we compared the results that we had collected using Steps 2a and 2b.

B. Results

In this section, we introduce the results from the manual repository and website analysis, then describe the corresponding results from the surveys, and finally compare them.

Repository Analysis. Table I presents an overview of the results from analyzing the information in project repositories and websites regarding ASATs for RQs 1.1-1.3. Overall, our results stem from analyzing 122 projects (see Bholanath's thesis [35, Appendix A] for the complete list). Most of them (36%+23%=59%) either mention the use of ASATs in their official project documentation, or their repository contains ASAT configuration files or their build processes specify explicit dependencies on ASATs. Of those projects, 28 use multiple ASATs. Examining the project sources separately, 53 out of 83 GitHub projects use ASATs and 25 of those use multiple ASATs. Only one OpenHub project does not use ASATs, and 2 of the other 8 projects use multiple ASATs. ASATs are not popular among our SourceForge projects, with only three of them adopting ASATs, all of which use a single ASAT. Finally, 7 out of 20 Gitorious projects use ASATs, but only one of them uses multiple ASATs.

TABLE I
Prevalence of ASATs according to our **Repository Analysis**.

Source	Projects	Use 1 ASAT	Use > 1 ASATs
GitHub	83	34%	30%
OpenHub	9	67%	22%
SourceForge	10	30%	0%
Gitorious	20	30%	5%
Total	122	36%	23%

TABLE II
Prevalence of ASATs according to our **survey**.

Source	Projects	Use 1 ASAT	Use > 1 ASATs	Enforce Use
GitHub	19	36%	32%	42%
OpenHub	1	0%	0%	0%
SourceForge	3	34%	66%	0%
Gitorious	10	30%	40%	30%
Other⋆	3	100%	66%	33%
Total	36	41%	36%	36%

⋆ Replied for their project B as a reaction to our survey in a mailing list of another project A (unrelated to B).

Survey. We received responses from 36 projects, achieving a relatively high response rate for surveys of 30%. Table II shows the corresponding results from the survey. The last column displays the percentage of projects that use the results of these tools as one of the factors to decide whether a code contribution should be integrated into the project repository. This displays information that we could not always obtain from a repository analysis. Overall, we observe that ASATs are used by 41% (Table I) + 36% (Table II) = 77% of projects who answered the survey. Most respondents state that ASATs are only sporadically used by developers when they believe that a code change warrants ASAT use.

Concerning RQ 1.3, we observe that a slight majority of the projects that use ASATs, 15 out of 28, rely on a single tool. Five projects use more than two ASATs, with no project using more than three comparable ASATs. Abilian [36] is the only project that uses more than three ASATs, but for three languages (JavaScript, CSS, and Python). All other projects only use a single ASAT or multiple ASATs that check for defects in the same language. Slightly less than half of the projects that use ASATs (13 out of 28) place a strict ASAT-regression policy on new code. This means that code that is submitted for review or integrated into the project repository must not introduce new ASAT warnings.

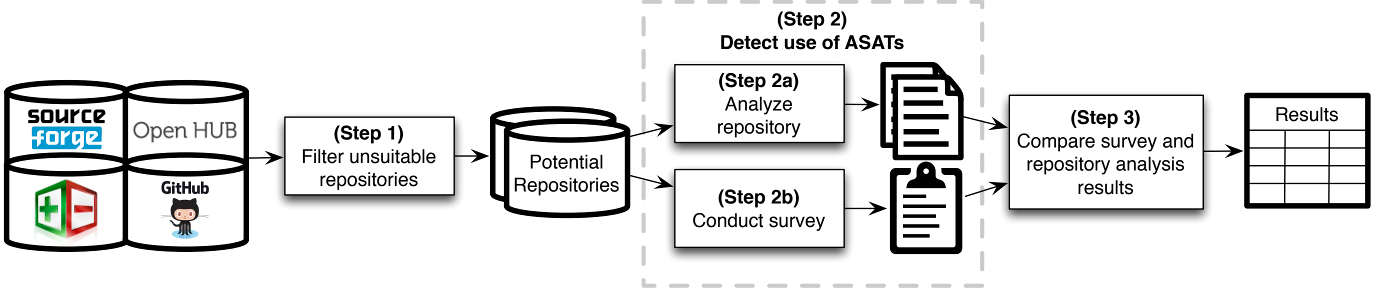

Fig. 1. An overview of the study design for the prevalence analysis.

Comparison. Using a mixed methods approach to evaluate the validity of our results [37], we compared the detailed results that are summarized in Tables I and II for all 36 projects for which we had both sources available. For close to 20% of the projects that responded to our questionnaire, the repository analysis results deviate in some way from the survey responses, which we consider to be the ground-truth. In three cases, the repository analysis shows that ASATs are used, while in reality, the project does not use any ASATs. A reason for this might be that the information that we gathered from the repository or website might be outdated. For example, the Bash project [38] mentioned that they previously used COVERITY [39], for which traces can still be found in existing sources. For two other projects, two tools are found in the project information, while respondents only note that one of them is in use. Moreover, two projects use a different ASAT than reported. Furthermore, there are seven projects that used more ASATs than the repository analysis indicated and a single project that used ASATs even though the repository analysis showed otherwise.

V. General Defect Classification (GDC)

When we reason about multiple ASATs to answer RQs 2 and 3, we are confronted with the problem that each tool provides a plethora of different individual rules (checks), often without an explicit ordering scheme or a topology. If we want to derive meaningful results from a comparison of multiple ASATs, we therefore need to design a common, more general classification scheme to allow us to abstract over the tool-specifics. To that end, we propose the GDC, which is based on the *code review defect classification* [28], which lent itself to an adoption of ASAT warnings because it categorizes "human static analysis" (i.e., code reviews), similar to "automatic static analysis" warnings. One useful property of a defect classification is that it can be used to categorize not only defects, but also warnings (or, review suggestions) and actual code changes. As such, we use the terms "defect" and "warning" interchangeably.

Figure 2 depicts the two different high-level categories of the GDC, maintainability and functional defects. It also shows the 16 second-level defect categories (7 functional, 9 maintainability). Similar to our prior work [28], there are two top-levels categories, functional and maintainability. Each of them is refined into a set of sub-categories that further characterize the warning. As one example of a new category, "Metric" pertains to warnings that "measure a certain attribute of the code," like the "NestedIfDepth" warning in CHECKSTYLE.

In a grounded-theory-driven approach, the first two authors separately browsed through all available FINDBUGS checks and tried to classify them into fitting groups, using the code review classification scheme as a blueprint [28]. Wherever we could not find a suitable existing category, we introduced a new one and sorted it into the topology. Upon completion of this task, the authors met and compared their adopted classifications, distilling a common first draft of the GDC. After this, we mapped the traditionally more maintainability-oriented CHECKSTYLE warnings into this preliminary ASAT warning topology. In this round, we soon reached saturation and only introduced two new categories under the maintainability sub-level.

To stimulate future research, we distribute our GDC classification along with detailed explanations on the error types and our manual mapping of all 2,385 checks of the nine ASATs to their corresponding GDC categories freely.[1]

VI. Configuration & Evolution (RQ 2, RQ 3)

In this section, we describe the design and results of our studies that address RQs 2 and 3.

A. Study Design

Figure 3 depicts our high-level study design for RQs 2 and 3. To facilitate the technical part of our analysis, we first decided to only consider projects that are hosted on GITHUB

[1] http://dx.doi.org/10.6084/m9.figshare.1603419

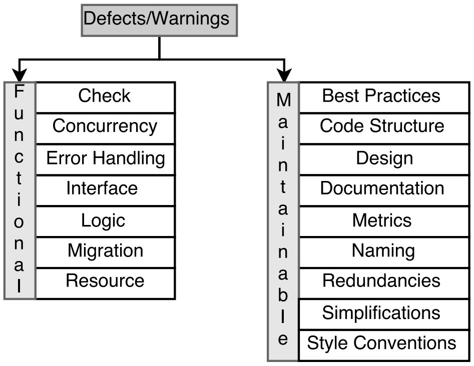

Fig. 2. Top- and second-level categories of the GDC.

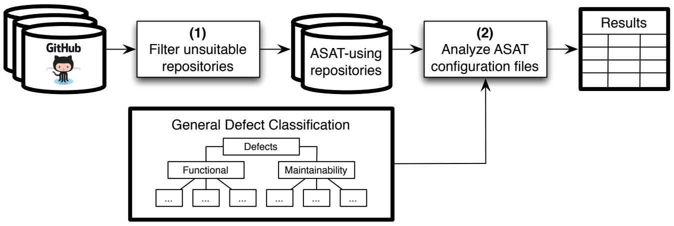
GitHub
(1)
Filter unsuitable repositories
ASAT-using repositories
(2)
Analyze ASAT configuration files
Results
General Defect Classification
Defects
Functional
Maintainability
...

Fig. 3. An overview of the study design for the ASAT configuration analysis.

(1). Having chosen a selection of nine ASATs, we then developed two tools: ASAT-CONFIGURATION-ANALYZER[2] for RQ 2 and the ASAT-HISTORY-ANALYZER[3] for RQ 3. Before we could use them, however, we had to crawl GITHUB for the occurrence of any of the supported ASAT configuration files and store a URL at which we can retrieve the content of the file (1). The tools then receive a list of URLs to configuration files, which it downloads and parses, applying the mapping of the individual tool checks to the GDC from Section V (2). The results are classified distributions of warnings that capture how developers configure their ASATs on a more abstract level than the individual tools would allow. We answer our research questions on the basis of these distributions.

B. Methods

In this section, we describe our study methodology.

Selection of ASATs (1). We placed some restrictions on the ASATs that we could use. First, an ASAT has to be configurable. If an ASAT is not configurable, then no study regarding its use is necessary. We can simply conclude that all developers use the ASAT in the same manner. Furthermore, if an ASAT is configurable, it needs to store its configuration in a separate file (and not, for example, via command line arguments). Finally, the configuration file needs to be parsable. In practice, this means that the configuration needs to be in a machine-readable format such as XML, JSON, or even a custom key-value pairing.

We used the ASATs that we encountered for RQ 1 (see Section IV) as a starting point. We expanded our search with search engines and programming support sites such as Stack Overflow [40]. Table III lists the nine tools which fit our criteria. Most tools use standard formats to store their configuration. Two tools, JSL and PYLINT, use key-value pairs in plain text format. FINDBUGS is a peculiar case. The tool uses XML files to either exclude or include rules in a specific class, file, or package. However, whether an element is an inclusion or an exclusion of a rule is specified via command line arguments. Thus, we could not determine this in a consistent way. Instead, we used the configuration files of the FINDBUGS Eclipse plugin. This plugin also stores its configuration in plain text key-value pairs.

One factor that could influence how developers configure their tools is what type of defects a tool focuses on. For the Java tools, CHECKSTYLE focuses primarily on coding style, FINDBUGS on functional defects, and PMD tries to find both types. For the JavaScript tools, JSCS focuses on coding style rules, while both JSHINT and ESLINT try to find all types of defects. JSHINT will refocus in an upcoming major release to functional defects and has marked many rules as deprecated in preparation for the removal of these coding style rules [50]. This might already have affected the configurations of developers, if they stopped using the deprecated rules in preparation for the change. JSL, PYLINT, and RUBOCOP do not state a particular focus on a specific subset of defects. However, RUBOCOP seems to favor checking for coding style issues, as made evident by the fact that most of their rules are classified by the tool itself as belonging to the *Style* category.

TABLE III
DESCRIPTION OF THE ASATS FOR RQ 2 AND 3.

Tool	Language	Format	Extendable	Released	# of Rules
CHECKSTYLE [41]	Java	XML	Yes	2001	179
FINDBUGS [42]	Java	Text	Yes	2003	160
PMD [43]	Java	XML	Yes	2002	330
ESLINT [44]	JavaScript	JSON	Yes	2013	157
JSCS [45]	JavaScript	JSON	Yes	2013	116
JSHINT [46]	JavaScript	JSON	No	2011	253
JSL [47]	JavaScript	Text	No	2005	63
PYLINT [48]	Python	Text	Yes	2006	390
RUBOCOP [49]	Ruby	YAML	Yes	2012	221

Analyzing Configuration Files (2). Developers can configure most ASATs to fit their specific needs through a configuration file. For RQs 2 and 3, we study how developers use ASATs through them as a proxy. In an ASAT configuration, developers can enable the rules that check for defects that they consider important, and disable rules they do not deem important (e.g., perhaps because of a high false positives rate). Without a configuration file, developers rely on the default, which might not align with their specific needs. The contents of an ASAT configuration file are hence an important indicator of how developers use ASATs to check for defects in their code and how well the tool's default settings reflect its use. A version-controlled ASAT configuration is crucial for collaboration because it enforces consistent static checks across developers.

For RQ 2.1 and RQ 2.2, we were interested in the distribution of the rules that are explicitly activated by developers and likewise those that are disabled by developers. This indicates which types of warnings are considered important by developers, and conversely, which types of warnings they avoid. To mitigate the influence of the set of possible ASAT rules, we normalized these distributions according to the number of rules in a category. To see why this is important, consider a hypothetical tool with just two defect categories, A and B, where A has one rule and B has two. If the developers enable the rule in category A once and each of the rules in category B, then a uniform distribution of defects would show B as being twice as actively-enabled as A. However, we can see that each individual rule in A and B was enabled once. The results in this normalized form allow us to study the relations between a category with a large number of rules and another with just one or two rules.

For RQ 2.3, we were interested to see if and how the

[2] https://github.com/rbholanath/ASAT-Configuration-Analyzer
[3] https://github.com/rbholanath/ASAT-History-Analyzer

configurations of developers deviated from the defaults, as these are indications of whether the default configuration accurately reflects the wishes of ASAT users. A developer can deviate from the default configuration in three ways: 1) Disable a rule that was enabled by default, 2) Enable a rule that was disabled by default, or 3) Reconfigure a rule.

Not all rules can be reconfigured. An example of a configurable rule is the "NestedIfDepth," for which the depth threshold can be customized via a simple integer value. Reconfiguring a rule indicates that developers want to check for this convention, but do not agree with the default convention as specified by the creators of the tool. We assume that a rule is reconfigured when a developer includes an enabled default rule in his own configuration.

To see if developers deviate from the default, we simply computed what percentage of configuration files included one or more deviations for a default rule. To examine how developers deviate from the default configurations, we computed, for each rule, how many configuration files included a particular type of deviation for that rule.

For RQ 2.4, we determined the prevalence of customized warning rules in comparison to the built-in rules of a tool. We consider a rule to be custom-made if it was not included as a built-in rule in a recent version of the ASAT. For each tool, this can indicate whether the developers consider the tool to be incomplete, which might result in developers writing custom rules to find these defects. Generally, this can be an indication of whether current ASATs can adequately cover the defects that developers wish to find.

Analyzing Configuration Evolution (2). For each identified configuration file from (1), we then performed an analysis of its evolution over time. The first metric, for RQ 3.1, is simply how often a file was changed. This tells us if developers have a need to adapt their ASAT configuration, either because the ASAT was updated or because of changing needs among developers. For RQ 3.2, we calculated the total number of line changes in a file. We defined this as the difference between the number of lines added and deleted in a single change. If this number is zero, it likely means that there are only lines modified, which count as both an addition and a deletion in the information of a change. We did not compute more fine-grained measures, such as an edit distance, because of the excessive computing load for all changes of the more than 160,000 configuration files and its relatively small expected information gain.

C. Study Objects

After selecting the ASATs to study, we needed to retrieve configuration files for every tool. We expected to find enough configuration files on GitHub. However, to further augment the study, we also collected data from Krugle [51] and OpenHub. To eliminate possible duplicates, we excluded OpenHub results which hosted their code on GitHub.

TABLE IV
Configuration files for each ASAT, grouped by source.

Tool	GitHub	OpenHub	Krugle	Total
Checkstyle	16,271	2,492	22	18,785
FindBugs	1,575	514	1	2,090
PMD	5,562	1,888	8	7,458
ESLint	4,427	5	3	4,435
JSCS	11,656	20	1	11,677
JSHint	105,619	3,086	65	108,770
JSL	862	0	0	862
Pylint	3,941	123	7	4,071
RuboCop	10,063	0	3	10,066
Total	159,976	8,128	110	168,214

Table IV details the number of configuration files split per ASAT and hosting site. As projects typically have one configuration, the numbers are a good estimator for the number of different projects. We identify JSHint as the most wideley-used ASAT among our selection. The number of added files from Krugle is minimal for all tools. Moreover, for some tools, there were more configuration files hosted on GitHub than we could access due to limitations in GitHub's search (see Section VII-B).

D. Results

In this section, we detail the results to RQs 2 and 3.

Results of RQ 2. RQ 2.1 and 2.2 are concerned with the warning rules that developers enable and disable respectively in their configurations. As explained in Section VI-B, we normalized the distribution of the enabled and disabled warning rules according to the number of rules in a category. Figure 4 details these normalized results for every tool. Every one of the 9 bars displays the percentage of normalized rules that belong to a specific category in our classification. Due to this, the differences in Figure 4 from a uniform distribution are due to developers over- or under-proportionally enabling or disabling rules in this category. As an example, almost 10% of the normalized rules that are enabled in FindBugs configurations belong to the Check category. The figures also allow us to identify categories that are outliers for a specific tool. For instance, the Metric and Migration categories contain a large percentage of the enabled rules for RuboCop. In both the enabled and disabled distributions, some tools show categories with no enabled or disabled rules.

For RQ 2.3, we calculated how many configurations deviated from the default. The second column of Table V shows how many configuration files changed one or more default rules, i.e., disabled a rule that was enabled in the default configuration or vice versa. The third column shows how many files did not change a default rule, but reconfigured a default rule. Blank spaces indicate that the tool does not allow individual rules to be reconfigured. The fourth column lists the percentage of configuration files that do not contain a deviation for any default rule.

With Table V as the basis, we also assessed in three ways how developers deviate from a default configuration. First, for all rules that are enabled in default configurations, we calculated how many developers disabled them. Subsequently, for every rule that was turned off by default, we calculated

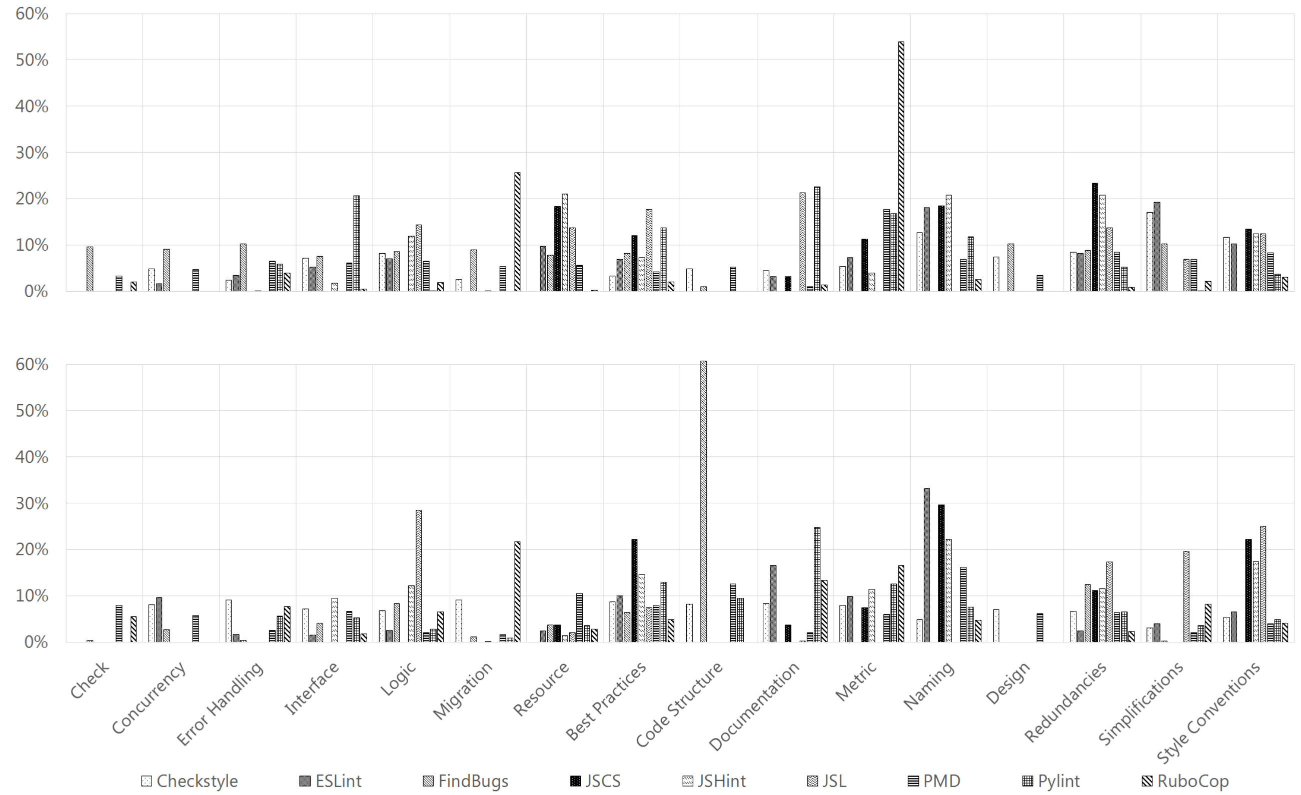

Fig. 4. Normalized average means of enabled (top) and disabled (bottom) checks per ASAT for all 16 second-level GDC categories.

in how many configurations that rule was enabled. Finally, for every rule that was enabled in the default configuration and which could be configured, we calculated how many configurations possibly reconfigured them.

Our results show that there is wide agreement with the default rules that ASAT providers ship: Only for the tools ESLint (2% of default rules affected) and JSHint (10%) did more than 50% of configuration files deviate from the default by reconfiguring a subset of rule defaults. For FindBugs and JSHint more than 50% of developers enabled 2 and 5 default-disabled rules, respectively.

For RQ 2.4, we calculated the percentage of custom rules in the configuration of developers. Table VI shows the results. Custom rules never account for more than 5% of all of the enabled rules of a tool. For 3 out of 8 ASATs, this percentage is even lower than 1%. We omit JSL from these results because JSL does not permit custom rules.

Results of RQ 3. Figure 5 shows the results for RQ 3.1 regarding how often and how profoundly configuration files change. A little over 80% of all configuration files are never changed after their creation. The range in the chart represents 99.5% of the total data. Less than 10% of all files are changed just once and less than 5% twice. The maximum number of times that a configuration file was changed is 248, for a Checkstyle configuration.

For the 19% of configuration files that were changed after

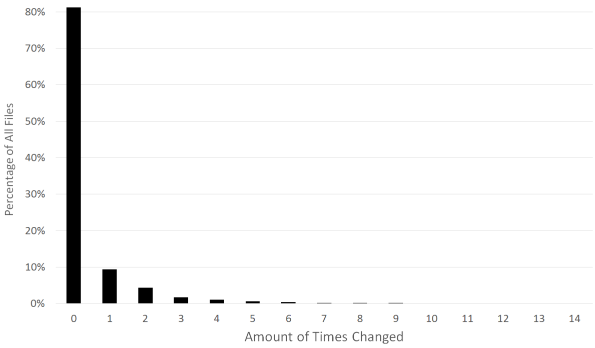

Fig. 5. Number of changes to an ASAT configuration (median 0, mean 0.5).

TABLE V
Summary of rule changes from default configurations.

Tool	Changed	Reconfigured	No Deviations	Total
ESLint	80.5%	5.7%	13.8%	4,274
FindBugs	93.0%	—	7.0%	2,057
JSHint	89.6%	0.7%	9.7%	104,914
JSL	94.6%	—	5.4%	848
Pylint	53.3%	—	46.7%	3,951
RuboCop	79.1%	3.2%	17.7%	9,579

TABLE VI
Average mean of custom rules in ASAT configurations.

Tool	Percentage of Custom Rules
Checkstyle	0.2%
ESLint	4.1%
FindBugs	1.3%
JSCS	4.7%
JSHint	0.1%
PMD	2.9%
Pylint	1.1%
RuboCop	0.9%

their initial creation, we analyzed each change of every file to determine the size of the change. The left distribution in Figure 6 shows the results to RQ 3.2. The total number of changes is zero for more than 25% of all files. This means that either lines were only modified “in-place,” or that there were as many completely new lines added as deleted. Furthermore, there is a greater chance that a change has more additions than deletions. The range in the chart captures more than 90% of the data. The rest of the data is spread out from -1,126 to 2,055 total changes.

The right distribution in Figure 6 shows the results to RQ 3.3. We see that 18% of the changes are made on the same day that the file is created and 33.5% of changes are made within the first week. The tail of the data is quite extensive, as the range shown in the chart covers just over 65% of the data. However, no date more than 15 days after the creation of the file individually represents more than 1% of all changes. The maximum is 11.5 years for a CHECKSTYLE configuration.

VII. DISCUSSION

In this section, we discuss our results and possible threats to the validity of our conclusions.

A. Results

For RQ 1.1, we found that the percentage of projects using ASATs according to the survey is higher than that of our repository analysis. Excluding projects that are present in both sets, 77% of the respondents of our survey note that ASATs are used compared to 52% of project resources. It seems highly likely that the respondents to our survey were more inclined to use ASATs, possibly explaining the 25%-point difference.

> The results of a large mining repository analysis give a useful approximation of the real ASAT use of OSS projects. It might be inaccurate on a single project basis.

However, the results from the survey also showed that the majority of projects that use ASATs (15 out of 28) only run these tools sporadically and without enforcing them. Researchers should avoid using data that is solely collected from project repositories and documentation to draw conclusions about ASAT use. This highlights the need to analyze multiple data sources in empirical software engineering [37], [52].

Our manual repository investigation showed that less than 59% of projects use ASATs in various levels of strictness for RQ 1.2. These results seem to contradict prior results [12], [53], [54], which claim that ASATs have not yet achieved significant use among software projects.

> ASATs are common, but not ubiquitous in popular OSS.

Table IV shows that our ASAT selection contained six times more JavaScript projects that used an ASAT than such Java projects. This is against our expectation since there are only three times more JavaScript than Java projects on GITHUB. Before drawing further conclusions, we need to evaluate this initial finding on a larger set of dynamic and static languages.

> Projects in a dynamically-typed language like JavaScript might require or benefit more from ASAT use than projects in a statically-typed language like Java.

The questionnaire also showed that the way in which ASATs are used varies. 64% of projects use ASATs sporadically and without attaching any consequences to the warning results.

> Few projects have ASATs tightly integrated into their workflows and even fewer projects mandate that the codebase should be ASAT-warning free.

Past research has suggested that an important factor of improving the adoption of ASATs was to make this integration as easy and seamless as possible [12]. For instance, COVERITY provides both GITHUB and TRAVIS CI [55] integration [56], [57], making it easy to integrate ASATs into an already existing code review workflow.

> In order to fully benefit from ASATs, projects should include them into their standard workflow, for example as part of their continuous integration processes.

Concerning RQ 1.3, we observed that most projects use one ASAT. This is in spite of the fact that the use of multiple ASATs can provide a large increase in defect detection capabilities [6], [13]–[15]. Developers might be unaware of these benefits or an overload of warning messages generated by multiple ASATs might cause developers to avoid them [12].

For RQ 2.1, we observed from Figure 4 (top graph) that 65% of all enabled rules belong to the GDC maintainability defect category. The other 35% of rules belong to a functional defect category. A reason for this might be that, ASATs perform poorly at finding functional defects [15], [20]–[22]. Ayewah [19] and Wagner [20] argue that the reason could be that ASATs do not know what code is intended to do, which is crucial if one wants to find functional defects. If developers notice the poor performance of these functional defect rules they might place less importance on them and subsequently leave them out of their configurations.

Concerning RQ 2.2, from Figure 4 (bottom graph) we observe that 75% of all the disabled rules are maintainability defects. However, the ratio of maintainability defects to functional defects is not significantly larger for the rules that developers disable than it is for those rules that developers enable. Even though the ability of ASATs to find functional defects is limited [15], [20]–[22], developers do not widely disable these rules. A potential reason for this might be that these rules do not emit a lot of false positives. On the contrary,

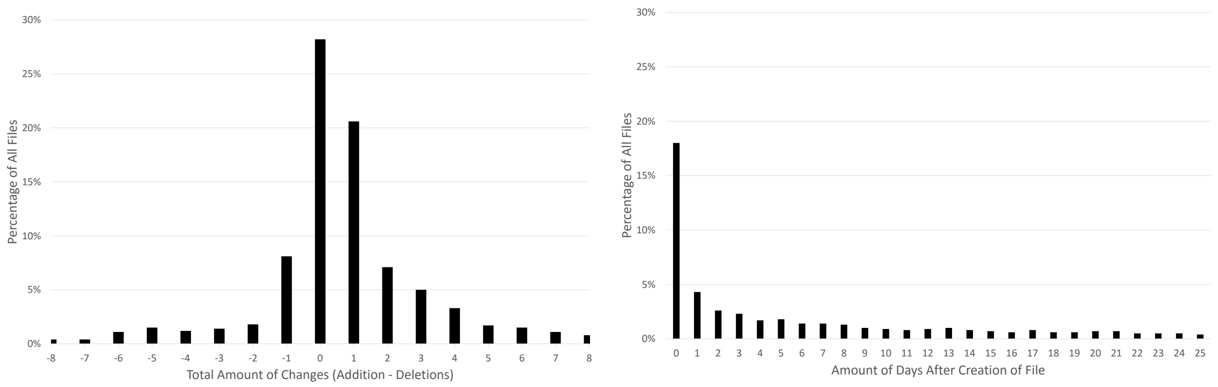

Fig. 6. Distribution of change size (left; median: 1, mean: 1.64) and changes per days after initial configuration (right; median: 32 days, mean: 151).

a rule that never emitted a warning might not be worth disabling, as it might still find a defect in the future.

Both the majority of actively enabled and disabled rules are maintainability-related.

As we have only compared the choices that developers make in their configuration files explicitly, this high-level observation is not a contradiction. To investigate it in more depth, we need a study that also takes into account the implicit defaults of configurations.

On a lower level, Figure 4 shows some outliers for individual tools. For instance, regarding enabled rules, the *Metric* category for RuboCop and the *Logic* category for Pylint stand out. For disabled rules, the FindBugs *Code Structure* category and the RuboCop *Migration* category are noticable outliers. These outliers indicate that, for a single tool or programming language, developers sometimes consider a specific category less or more important than developers using other ASATs or languages.

For the results regarding RQ 2.3, Table V shows that, for all tools, less than half of all configurations do not change or reconfigure any rule from the default configuration. For 5 out of 6 tools, this percentage is even lower than 20% and for 3 out of 6 tools it is less than 10%.

Most configurations change or reconfigure rules from the default configuration, but typically only one rule.

The results described in Section VI-D, and Table V and Table VI in particular, indicate that there are few rules that a noticeable percentage of all developers change or reconfigure. Figure 4 could suggest improvement opportunities in the default configurations of ASATs. For the enabled rules, 5 out of 7 tools have zero default rules that are disabled by developers in more than 25% of all configuration files. Moreover, less than 5% of the rules for the other two tools are disabled more than 25% of the time. For the rules that are disabled by default, 3 out of 6 tools do not have any rules that are turned back on by more than 25% of all developers. The other three tools have a higher number of such rules. Most striking are the results for FindBugs. Even though there are just eight rules that are disabled by default, the results show that the default configuration should probably enable rather than disable some of those rules. Regarding the reconfigurable rules, the percentage of rules that are potentially reconfigured by developers are low among all three tools. However, both ESLint and JSHint still have rules that pass the 50% mark. The creators of these tools should therefore consider changing the default settings.

Developers only widely disagree with few rules in default configurations.

Finally, regarding RQ 2.4, our results show that custom rules do not comprise a sizable segment of all rules, amounting to less than 5% for all tools. This can indicate that developers do consider the ASATs to be complete, in the sense that they need not create custom rules to check for defects that are not included in the built-in rule set. Nevertheless, this could also be an unwillingness to create custom rules, with developers manually checking for those rules they consider to be missing in the ASATs that they use.

Custom rules comprise less than 5% of all rules that are used by developers.

Regarding RQ 3.1, the results show that the use of ASATs is relatively stagnant (i.e., does not evolve). Over 80% of all the configuration files that we analyzed are created and then used as-is for the remainder of the project's lifetime to date. Moreover, only 5% of all configuration files are changed more than twice and less than 2% are altered more than five times.

Most configuration files never change.

Looking at only the files that are changed, the results for RQ 3.2 show that, for most files, the total number of changed lines lies within a reduction of five lines to an increase of five lines. Furthermore, more than 28% of all files have an equal number of added and deleted lines, indicating that there were likely only modified lines.

> Most changes to configuration files are small in size.

The results for RQ 3.3 show that a configuration files is most likely to be changed on the same day that it was created. Looking ahead one week, we see that slightly over a third of all changes were made in this time span. Going even further, almost half of all changes are made within a month after a file's creation. Thus, we observe that developers that make changes to their configuration files do not only do so in the period where they are still getting used to the ASAT. Assuming that this period lasts a week, or surely no longer than a month, at least 50% of all changes are made after the ASAT was used for a lengthy amount of time.

> A third of all changes happen in the first week after the creation of an ASAT configuration.

B. Threats to Validity

In this section, we explain which threats affect the validity of our study and show how we mitigated them.

Internal Validity. Since most study points stem from GITHUB (see Table IV), this might bias our results. To minimize the bias, we looked for all code hosting services and search engines that allowed us to find ASAT configuration files. Hence, our bias towards GITHUB might simply be reflective of its current popularity among OSS projects.

There might be errors in our measurements due to the use of our analysis tools. We verified that the tools worked as expected on small, manually curated samples and through automated tests. Moreover, we programmed our tool defensively, that is, the tool skips those configuration files that do not conform to a strict specification. To mitigate this risk further, we open-sourced our tools.

GitHub's search only retrieves 1,000 hits. We worked around it by strategically boxing and modifying its file size parameter in one-byte increments. However, this was too coarse-grained for some searches. As a result, we could not retrieve a few hundred configuration files for most tools, and about 220,000 for JSHINT. Our sample size of over one third of the total JSHINT population is still significant.

For RQ 2.3, we assumed that the current default settings still applied when the ASATs were initially adopted by the studied repositories. If the default configuration changed significantly over time, our results might be inaccurate. However, manual inspection of a few projects showed that the default typically evolves gracefully, adding new options, but not changing existing ones.

External Validity. This study only considers the configurations of ASATs from OSS projects. As such, its generalizability towards closed-source projects might be limited.

Our study targets nine ASATs and four programming languages, representing a diverse set of tools (see Table IV). Therefore, we expect those results that abstracted over all the tools and presented a general view of the studied ASATs to further generalize over ASATs outside of this study as well. However, replication studies are needed to confirm this.

VIII. FUTURE WORK & CONCLUSIONS

In this paper, we have performed an investigation into how a large set of OSS projects use static analysis. Our findings show that, 60% of the most popular and (therefore arguably) most advanced projects make use of ASATs. Projects which use ASATs typically do not embed them in their workflow and use them only sporadically. Our results seem to suggest that dynamically-typed languages benefit from or require more ASAT support than static languages. Future research could broaden the group of languages for this analysis to assert and further investigate this finding.

Our analysis into the usage of ASATs through their configuration files has shown that the default configurations of most tools are a good fit to the majority of projects. Only two tools contained default checks that developers regularly disagreed with. In line with the picture of a light use of ASATs are our results on the evolution of their configuration files: There typically is no evolution. Most ASAT configurations, after an initial period of change of one week, remain unchanged in project repositories.

Our findings seem to suggest that OSS developers need to be made aware of the benefits of using ASATs, and how easy an integration into their fixed workflow or even continuous integration process can be. On the other hand, developers might be skeptical of the practical usefulness of ASATs due to a possible overload with irrelevant warnings.

Apart from a purely empirical analysis, this paper also contributes the GDC and practical guidelines for users and creators of ASATs. Possible benefiters are:

Researchers, who can replicate our study and use the classification for further studies on ASATs. The GDC might be especially useful for studies on the intersection of ASATs with code review [58].

Practitioners, who could assess the strengths and weaknesses of ASATs by inspecting the distribution profile of the number of supported checks in each category. For example, FINDBUGS emphasizes functional checks.

Tool Creators of FINDBUGS and RUBOCOP, who may want to re-assess the defaults for their rules in two GDC categories. Developers seem to accept the remainder of the defaults.

Dashboard Creators of tools, such as TEAMSCALE [59] and SONARQUBE [60], who could rank, compare, filter, prioritize, and possibly remove duplicates when they assemble warnings from multiple ASATs in one location.

References

[1] NASA, "JPL C Standard," 2015, accessed on: November 14th, 2015. [Online]. Available: http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf

[2] NASA, "JPL Java Standard," 2015, accessed on: November 14th, 2015. [Online]. Available: http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_Java.pdf

[3] S. C. Johnson, "Lint, a c program checker," in *Computer Science Technical Report 65*. Bell Laboratories, 1977.

[4] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[5] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[6] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, no. 0, pp. 5–21, 2008.

[7] Coverity Inc., "Effective management of static analysis vulnerabilities and defects," Coverity Inc., White Paper, 2009.

[8] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.

[9] J. Ruthruff, J. Penix, D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, Conference Proceedings, pp. 341–350.

[10] J. Kamperman, "Automated software inspection: A new approach to increased software quality and productivity," Reasoning Inc., White paper, 2002.

[11] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, Conference Paper, pp. 41–50.

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

[13] N. Rutar, C. Almazan, and J. Foster, "A comparison of bug finding tools for java," in *15th International Symposium on Software Reliability Engineering*, 2004, Conference Proceedings, pp. 245–256.

[14] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, *Comparing Bug Finding Tools with Reviews and Tests*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3502, book section 4, pp. 40–55.

[15] F. Wedyan, D. Alrmuny, and J. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *International Conference on Software Testing Verification and Validation*. IEEE, 2009, Conference Proceedings, pp. 141–150.

[16] S. Heckman, "Adaptively ranking alerts generated from automated static analysis," *Crossroads*, vol. 14, no. 1, pp. 1–11, 2007.

[17] S. Kim and M. Ernst, "Which warnings should i fix first?" in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, Conference Paper, pp. 45–54.

[18] T. Kremenek and D. Engler, *Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2694, book section 16, pp. 295–315.

[19] N. Ayewah, W. Pugh, D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, Conference Paper, pp. 1–8.

[20] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An evaluation of two bug pattern tools for java," in *1st International Conference on Software Testing, Verification, and Validation*, 2008, Conference Proceedings, pp. 248–257.

[21] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, Conference Paper, pp. 241–252.

[22] C. Couto, J. Montandon, C. Silva, and M. T. Valente, "Static correspondence and correlation between field defects and warnings reported by a bug finding tool," *Software Quality Journal*, vol. 21, no. 2, pp. 241–257, 2013.

[23] A. K. Tripathi and A. Gupta, "A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for java programs," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2014, Conference Proceedings, p. 23.

[24] IEEE, "Ieee standard classification for software anomalies," *IEEE Std 1044-1993*, pp. 1–32, 1994.

[25] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[26] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk, "Preliminary results on using static analysis tools for software inspection," in *15th International Symposium on Software Reliability Engineering*, 2004, Conference Proceedings, pp. 429–439.

[27] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[28] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: which problems do they fix?" in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, Conference Paper, pp. 202–211.

[29] M. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430–448, 2009.

[30] K. El Emam and I. Wieczorek, "The repeatability of code defect classifications," in *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, 1998, Conference Proceedings, pp. 322–333.

[31] J. P. Kesan and R. C. Shah, "Setting software defaults: Perspectives from law, computer science and behavioral economics," *Notre Dame L. Rev.*, vol. 82, p. 583, 2006.

[32] GitLab Inc., "Code, test, and deploy together," 2015, accessed on: November 14th, 2015. [Online]. Available: https://about.gitlab.com/about/

[33] Google Inc., "Bidding farewell to google code," 2015, accessed on: November 14th, 2015. [Online]. Available: http://google-opensource.blogspot.nl/2015/03/farewell-to-google-code.html

[34] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, Conference Proceedings, pp. 384–387.

[35] R. Bholanath, "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software," Master's thesis, Delft University of Technology, 2015, http://repository.tudelft.nl/view/ir/uuid:3d834130-8dd7-420a-9af9-6e77761cdad6/.

[36] The Abilian Team, "Abilian github repository," 2015, accessed on: November 14th, 2015. [Online]. Available: https://github.com/abilian/

[37] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 179–190.

[38] GNU, "The bash shell," 2015, accessed on: November 14th, 2015. [Online]. Available: https://gnu.org/software/bash/bash.html

[39] Coverity Inc., "Software testing and static analysis tools — coverity," 2014, accessed on: October 3rd, 2014. [Online]. Available: http://www.coverity.com/

[40] Stack Exchange Inc., "Stack Overflow," 2015, accessed on: November 14th, 2015. [Online]. Available: http://stackoverflow.com/

[41] O. Burn, "checkstyle - checkstyle 5.9-snapshot," 2014, accessed on: October 14, 2014. [Online]. Available: http://checkstyle.sourceforge.net/

[42] FindBugs, "Findbugs - find bugs in java programs," 2014, accessed on: October 2nd, 2015. [Online]. Available: http://findbugs.sourceforge.net/

[43] PMD, "Pmd," 2014, accessed on: October 2nd, 2014. [Online]. Available: http://pmd.sourceforge.net/

[44] ESLint, "Eslint - pluggable javascript linter," 2015, accessed on: November 7th, 2015. [Online]. Available: http://eslint.org/

[45] JSCS, "Jscs - about," 2015, accessed on: May 7th, 2015. [Online]. Available: http://jscs.info/

[46] A. Kovalyov, "Jshint, a javascript code quality tool," 2015, accessed on: November 7th, 2015. [Online]. Available: http://jshint.com/

[47] M. Miller, "Javascript lint," 2015, accessed on: November 7th, 2015. [Online]. Available: http://www.javascriptlint.com/

[48] Logilab, "Pylint - code analysis for python — www.pylint.org," 2015, accessed on: May 7th, 2015. [Online]. Available: http://pylint.org/

[49] B. Batsov, "Rubocop — a ruby static code analyzer," 2015, accessed on: November 7th, 2015. [Online]. Available: http://batsov.com/rubocop/

[50] A. Kovalyov, "Jshint option reference," 2015, accessed on: May 8th, 2015. [Online]. Available: http://jshint.com/docs/options/

[51] Aragon Consulting Group, Inc., "Krugle – #1 for enterprise code search," 2015, accessed on: November 14th, 2015. [Online]. Available: http://www.krugle.com/

[52] S. Negara, M. Vakilian, N. Chen, R. Johnson, and D. Dig, *Is it dangerous to use version control histories to study source code evolution?*, ser. ECOOP 2012–Object-Oriented Programming. Springer, 2012, pp. 79–103.

[53] R. Kumar and A. Nori, "The economics of static analysis tools," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, Conference Proceedings, pp. 707–710.

[54] G. Gousios, A. Zaidman, M. D. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 358–368.

[55] Travis CI GmbH, "Travis continuous integration," 2015, accessed on: November 14th, 2015. [Online]. Available: https://travis-ci.com

[56] Coverity Inc., "Coverity scan - github integration," 2015, accessed on: October 21st, 2015. [Online]. Available: https://scan.coverity.com/github

[57] ——, "Coverity scan - travis ci integration," 2015, accessed on: November 13th, 2015. [Online]. Available: https://scan.coverity.com/travis_ci

[58] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2015, Conference Proceedings, pp. 161–170.

[59] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software quality control in real-time," in *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE'14)*, 2014.

[60] G. Campbell and P. P. Papapetrou, *SonarQube in Action*. Manning Publications Co., 2013.