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 do they adapt their use 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. We compare warnings across the ASATs, introduce the General Defect Classification (GDC) and provide a grounded-theory-derived mapping of 1,825 ASAT-specific warnings to the 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 the occurrence of 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 open source projects from the GITHUB, OPENHUB, SOURCEFORGE, and GITORIOUS software forges in search of use of 9 popular ASAT tools. 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 ASAT tools 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 tool in a very ad-hoc fashion not integrated in the development work flow.

RQ 2 How are ASATs configured?  
The ASAT configurations in OSS typically barely deviate from the default and introduce very few custom checks.

RQ 3 How does the use of ASATs evolve?  
Most ASAT configurations, once committed, never change and if they do, this is typically a very small adaption within a week of their introduction.

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 general defect classification, 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.

Whereas false negatives impact the efficiency of ASATs because defects are missed, false positives cause developers to waste time arbitraging 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 might be around 40 warnings per thousand lines of code [11]. This overload of warnings is a prime reason for developers to refrain from 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 open source projects.

Many ASATs differ in the type of defects they focus on. 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 have to process. Thus, deciding to use multiple ASATs is striking a balance between an improved defect detection rate and the additional work in processing the 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 defect [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 composed 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 aspect from which to classify the defect. While the ODC scheme has seen use among researchers [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]. It originally goes back to 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 aim of this paper is to increase our understanding of how static analysis tools are used in the real-world. We instantiate this research by studying a large collection of open-source software projects from both statically (Java) and dynamically typed languages (JavaScript, Ruby, and Python) in professional and non-professional popular OSS settings.

To this end, we must first establish a baseline of how common static analysis is in these projects. Hence, in our first research question, we ask:

RQ 1 How common is the use of static analysis techniques 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 through an automated investigation of the ASAT configuration files in their repository.

RQ 2 How are ASATs configured?

The answer to 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 enabled?

RQ 2.2 What type of warnings are 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 research questions:

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 widespread the use of static analysis is in popular Open-Source Projects.

A. Methodology

To answer this question, we followed the study design depicted in Figure 1. We started by examining the four popular open-source 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 had no way to rank projects by 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 (most 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 homes. OpenHub’s overall popularity ranking was only available for the 10 most popular projects. After eliminating duplicates and non-software repositories, in total, we ended up with 122 unique projects to analyze.

In 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 asked the same 122 projects in a short survey which static analysis tools they are using, when they are using it, if it is a necessary precondition to check code before it can enter the main-line repository, and whether it is an integral part of their work flow (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 entry barrier to the survey by embracing a discussion-style answer of developers directly to our informal email [35]. Afterwards, we compared the analysis results which we had available from both Steps 2a and 2b in Step 3.

B. Results

In this section, we first 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 [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 traces of ASATs, be it as configuration files or as explicit dependencies. From 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 using automated static analysis, all with a single tool. Finally, 7 out of 20 Gitorious projects use ASATs, but only one of them uses multiple ASATs.

Survey. We received responses from 36 projects, achieving a comparatively high response rate for surveys of 30%. Table II shows the corresponding results from the survey. We know the first four columns from Table I. The last column displays which percentage of projects use the results of these tools as one of the factors to decide whether a code contribution makes it into the repository. This displays information that we could not always obtain from a repository analysis. Overall, we observe that ASATs are used by 41%+36%=77% of projects who answered the survey. Most respondents state that ASATs are only sporadically used by developers whenever they believe they need to validate the quality of their code. 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 either 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, also place strict requirements on

<table>
<thead>
<tr>
<th>Source</th>
<th>Projects</th>
<th>Use 1 ASAT</th>
<th>Use &gt; 1 ASAT</th>
<th>Enforce Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>GitHub</td>
<td>83</td>
<td>34%</td>
<td>30%</td>
<td>30%</td>
</tr>
<tr>
<td>OpenHub</td>
<td>9</td>
<td>67%</td>
<td>22%</td>
<td>22%</td>
</tr>
<tr>
<td>SourceForge</td>
<td>10</td>
<td>30%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Gitorious</td>
<td>20</td>
<td>30%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Total</td>
<td>122</td>
<td>36%</td>
<td>23%</td>
<td>23%</td>
</tr>
</tbody>
</table>

* Repaired for their project R as a reaction to our survey in a mailing list of another project A (unrelated to B).
new code. This means that code that is submitted for review or pushed to the repository of these projects must not cause an ASAT regression, i.e. introduce new warnings on top of the already existing ones.

Comparison. In a mixed-methods approach to triage the validity of our results [37], we juxtaposed the detailed results 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 actuality the project does not use any ASATs. A reason for this might be that the information gathered from the repository or website might be outdated. For example, the Bash project [38] mentions 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 used. 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 over 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 that allows us to abstract over the tool-specifics. To this end, we propose the GDC, the General Defect Classification. The GDC 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). As in [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 blue print [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 study design and objects which are shared between RQ 2 and RQ 3 and their respective results.

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

Fig. 2. Top- and second-level categories of the GDC.
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 hosted on GitHub (1). Having chosen a selection of nine ASATs, we then developed two tools: ASAT-CONFIGURATION-ANALYZER² for RQ 2 and the ASAT-HISTORY-ANALYZER³ 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 detail 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 with search engines and programming support sites such as Stack Overflow [40]. Table III lists the nine tools which fit.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Language</th>
<th>Format</th>
<th>Extendable</th>
<th>Released</th>
<th># of Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECKSTYLE</td>
<td>Java</td>
<td>XML</td>
<td>Yes</td>
<td>2001</td>
<td>179</td>
</tr>
<tr>
<td>FINDBUGS</td>
<td>Java</td>
<td>Text</td>
<td>Yes</td>
<td>2003</td>
<td>160</td>
</tr>
<tr>
<td>PMD</td>
<td>Java</td>
<td>XML</td>
<td>Yes</td>
<td>2002</td>
<td>330</td>
</tr>
<tr>
<td>ESLINT</td>
<td>JavaScript</td>
<td>JSON</td>
<td>Yes</td>
<td>2013</td>
<td>157</td>
</tr>
<tr>
<td>JSCS</td>
<td>JavaScript</td>
<td>JSON</td>
<td>Yes</td>
<td>2013</td>
<td>116</td>
</tr>
<tr>
<td>JSHINT</td>
<td>JavaScript</td>
<td>JSON</td>
<td>No</td>
<td>2011</td>
<td>253</td>
</tr>
<tr>
<td>FOP</td>
<td>JavaScript</td>
<td>Text</td>
<td>No</td>
<td>2005</td>
<td>63</td>
</tr>
<tr>
<td>PYLINT</td>
<td>Python</td>
<td>Text</td>
<td>Yes</td>
<td>2006</td>
<td>390</td>
</tr>
<tr>
<td>RUBOCOP</td>
<td>Ruby</td>
<td>YAML</td>
<td>Yes</td>
<td>2012</td>
<td>221</td>
</tr>
</tbody>
</table>

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. It should be noted that JSHINT will refocus in an upcoming major release to functional defects and has marked many rules as deprecated in preparation of 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 of 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.

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, for example because they generate too many false positives. Without a configuration file, developers rely on the default, which might not be in line 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 reflects these wishes. 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 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, for example because they produce too many false positives. To remove 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 categories of defects: A and B, with one and two rules in those categories respectively. If the developers...
enabled the rule of category $A$ once and the combined rules of category $B$ twice, then a uniform distribution of defects would show $B$ as being twice as large as $A$. However, we can see that every rule in both categories 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 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 per default. 2) Enable a rule that was disabled per default. 3) Reconfigure a rule that was enabled per default.

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 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 custom created 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. Per 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 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 the relatively little expected information gain from it.

### 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 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 widely-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 concern 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 underproportionally 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, see Table V. The second column shows how many configuration files changed one or more default rules, that is, disabled a rule that was enabled in the default configuration or vice versa. The third column shows
Fig. 4. Normalized average means of enabled (top) and disabled (bottom) checks per ASAT for all 16 second-level GDC categories.

how many files did not change a default rule, but reconfigured a default rule. There are blanks for those tools that do 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 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 there is wide agreement with the default rules 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 a few default-disabled rules (2 and 5 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 enabled rules in a tool. For 3 out of 8 tools, this percentage is even lower than 1%. JSL is absent from these results, as users cannot write custom rules for this tool.

**Results of RQ 3.** The results for RQ 3.1, regarding how often a configuration file changes, are shown in Figure 5. A little over 80% of all configuration files never change after their initial 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 a configuration file was changed is 248, for a CHECKSTYLE configuration.

For the 19% of configuration files that were changed after their initial creation, we analyzed each change of every file to determine the size of the change. The top distribution in Figure 6 shows the results to RQ 3.2. The total number of

### Table V

<table>
<thead>
<tr>
<th>Tool</th>
<th>Changed</th>
<th>Reconfigured</th>
<th>No Deviations</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESLINT</td>
<td>80.5%</td>
<td>5.7%</td>
<td>13.8%</td>
<td>4,274</td>
</tr>
<tr>
<td>FINDBUGS</td>
<td>93.0%</td>
<td>—</td>
<td>7.9%</td>
<td>2,057</td>
</tr>
<tr>
<td>JSHINT</td>
<td>89.6%</td>
<td>0.7%</td>
<td>9.7%</td>
<td>104,914</td>
</tr>
<tr>
<td>JSL</td>
<td>94.6%</td>
<td>—</td>
<td>5.4%</td>
<td>848</td>
</tr>
<tr>
<td>PYLINT</td>
<td>53.3%</td>
<td>—</td>
<td>46.7%</td>
<td>3,951</td>
</tr>
<tr>
<td>RUBOCOP</td>
<td>79.1%</td>
<td>3.2%</td>
<td>17.7%</td>
<td>9,579</td>
</tr>
</tbody>
</table>

### Table VI

<table>
<thead>
<tr>
<th>Tool</th>
<th>Percentage of Custom Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHECKSTYLE</td>
<td>0.2%</td>
</tr>
<tr>
<td>ESLINT</td>
<td>4.1%</td>
</tr>
<tr>
<td>FINDBUGS</td>
<td>1.3%</td>
</tr>
<tr>
<td>JSCS</td>
<td>4.7%</td>
</tr>
<tr>
<td>JSHINT</td>
<td>0.1%</td>
</tr>
<tr>
<td>PMD</td>
<td>2.9%</td>
</tr>
<tr>
<td>PYLINT</td>
<td>1.1%</td>
</tr>
<tr>
<td>RUBOCOP</td>
<td>0.9%</td>
</tr>
</tbody>
</table>
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 -1126 to 2055 total changes.

The bottom 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 according to the repository analysis. Excluding projects in present in both sets, 77% of survey respondents 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 open source 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 therefore be careful when using information solely from project repositories, websites, and documentation to draw conclusions about ASAT use. This showcases the need for multiple sources of information 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 the experiences of [12], [53], [54], who claim that ASATs have not 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 insofar surprising as there are in total only three times more JavaScript projects on GitHub.

Projects in a dynamic language such as JavaScript might benefit more from an ASAT or require more static support, and therefore could use ASATs more widely.

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 work flow and even fewer mandate that the project be ASAT-warning free.

Research 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 work flow.

In order to fully benefit from ASATs, projects should include them into their standard work flow, for example as part of their continuous integration process.

Concerning RQ 1.3, we observed that most projects use only 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 not to use them [12].

For RQ 2.1, we distilled 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] hypothesize that the reason could be that ASATs do not know what code is supposed 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.
Developers enable rules that check for maintainability defects more often than those that check for functional defects.

Concerning RQ 2.2, from Figure 4 (bottom graph) we distill 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, a rule that never emitted a warning might not be worth disabling, as it might find a defect in the future.

Developers disable rules that check for maintainability defects more often than those that check for functional defects.

Thus, we observe that developers more often enable rules that check for maintainability defects, but also disable rules from maintainability categories more frequently than rules from functional defect categories. This can potentially be explained because enabling a rule makes sense when an ASAT is competent at finding such defects, which is not the case for functional defects [15], [20]–[22]. On the other hand, disabling a rule makes sense when that rule does not fit the coding style of the projects, or if it emits a lot of false positives. The former is almost exclusively an issue for mainatainability defects, because functional defects cause problems in every project, no matter the coding style. This could explain why rules that check for maintainability defects are disabled more often.

On a lower level, there are 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 a 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 mostly 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 projects 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 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 reliance on GitHub might simply be reflective of the whole OSS landscape.

There might be errors in our measurements due to the use of our analysis tool. We verified that the tool worked as expected on small 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 sizes. As a result, we could not retrieve a few hundred configuration files for most tools, and about 220,000 for JSHint. Still, our sample size of over one third of the total JSHint population is still significant.

For RQ 2.3, we assumed the current state of the default rules. 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, by adding new options instead of changing existing ones.

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

We performed this study for nine specific ASATs in four different 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.

**VIII. Future Work & Conclusion**

In this paper, we have performed an initial investigation on how a large set of open-source projects use static analysis. Our findings show that, in general, only 60% of the most popular and (therefore arguably) most advanced projects make use of ASATs. The users of ASATs typically do not have them embedded in their work flow and use them only sporadically. Our results seem to suggest that dynamic 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 adaption of one week, remain unchanged in the repository.

Our findings seem to suggest that open-source developers need to be made aware of the benefits of using ASATs, and how easy an integration into their fixed work flow 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 profitiers are:

**Researchers** can replicate the study in this paper 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** could assess the different strengths and weaknesses of ASATs by inspecting the distribution profile of the number of supported checks in each category. For example, **FINDBUGS** would be very functional-heavy.

**Tool Creators** of FINDBUGS and RUBOCOP may want to reassess the defaults for their tools in two GDC categories. Developers seem to accept the remainder of the defaults.

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

**REFERENCES**


