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

Master's Thesis

Radjino Bholanath

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

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Radjino Bholanath
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

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

Author: Radjino Bholanath
Student id: 4112083
Email: r.m.r.bholanath@student.tudelft.nl

Abstract

Static analysis is an important part of today's quality assurance process. It can be performed manually, by means of code reviews, or automatically, by automated static analysis tools (ASATs). However, there is still much unknown about the state of static analysis. This includes hard data on how prevalent static analysis is among projects. And while there have been studies on how projects use code reviews, current research has not investigated how developers configure the ASATs that they use and how these configurations evolve. In this thesis, we answer these questions by means of a large scale analysis of open source software. We found that both code reviews and ASATs are common, but not ubiquitous. Many projects do not perform code reviews for the changes of core developers and do not enforce a strict use of ASATs. Regarding the use of ASATs, developers both use and avoid maintainability defects to a greater extent than functional defects. Most configurations of developers deviate from the default and hardly contain custom rules. However, there are few default rules that are changed by a significant percentage of developers. Finally, most configuration files never change. And if they do, the changes are small, occur over the lifetime of the project, and are not triggered by ASAT version updates.

Thesis Committee:

Chair:	Dr. A. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	M. Beller, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Bacchelli, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Hauff, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

It is important when assuring the quality of a software product to not only reduce the number of defects to the minimum possible level, but also to catch defects as early in the development process as possible. Static analysis, which examines code and other development artifacts without executing them [34], allows for the earlier detection of defects than dynamic analysis [47], which runs the program in a real or virtual environment [34]. Moreover, static analysis techniques can explore all possible program behaviors, while dynamic analyses are limited by the current environment [33, 47, 50, 58, 73]. Thus, static analysis is an important part of the quality assurance process.

Static analysis can be done manually or automatically. The former is often called *code review*, or inspection. In a code review, one or more developers inspect the code of another developer to find defects and other potential improvements to the code [4]. Automated static analysis tools, or *ASATs*, inspect code automatically and use techniques such as bug patterns [33], control-flow analysis [18], or data-flow analysis [71] instead of relying on human judgment, knowledge, and reasoning. ASATs feature sets of *rules* that check for a single potential defect and emit a *warning* when they find a violation of a rule in the code.

Much is currently unknown about the state of static analysis, both in open source and industry settings. First, researchers are divided about the prevalence of static analysis techniques. Rigby et al. [67] stated that code reviews are ever present in open source settings, while Beller et al. [8] observed that many projects did not perform code reviews in a consistent and continuous manner. For ASATs, Ayewah et al. [3] stated in 2007 that they started to achieve significant adoption, but, more recently, Johnson et al. [36] and Kumar and Nori [48] stated that ASATs have struggled to become prevalent. We further discuss the related literature in Chapter 2. A quantitative analysis for any of the claims in the literature has not been performed. This leads to the following research question, which is discussed in Chapter 3:

RQ1: How common are static analysis techniques in practice?

- **RQ1.1:** What is the prevalence of code reviews?
- **RQ1.2:** What is the prevalence of ASATs?
- **RQ1.3:** Do projects use multiple ASATs to check the quality of their code?

Other than the absence of hard data regarding the prevalence of static analysis techniques, there has not been a study on how developers use ASATs in practice. Research on ASATs has mostly focused on the defects that they do or do not find [2, 13, 78, 79], and on the number of false positives that they generate [31, 36, 41, 48]. This contrasts with code reviews, where Rigby et al. performed several studies on the use of code reviews in open source settings [62, 63, 64, 65, 66, 67]. Furthermore, Panichella et al. [56] studied, for two specific ASATs, how developers handle ASAT warnings during code reviews. To obtain a more general understanding of how ASATs are being used in practice, Chapter 4 discusses the following research question:

RQ2: How are ASATs used?

- **RQ2.1:** What type of warnings do developers enable?
- **RQ2.2:** What type of warnings do developers disable?
- **RQ2.3:** Do default configurations reflect the wishes of developers?
- **RQ2.4:** How prevalent are custom rules in the configurations of developers?

Aside from studying how ASATs are currently used, Chapter 5 examines how this use changes over time. This information could be of interest to the creators of ASATs, who provide new and improved versions of such tools on a regular basis. Of particular interest is how often developers change their use of ASATs, what these changes entail, and if developers keep up with the latest versions of ASATs. Therefore, we strive to answer the following research question in Chapter 5:

RQ3: How does the use of ASATs evolve?

- **RQ3.1:** How often does a configuration file change?
- **RQ3.2:** How much does a configuration file change?
- **RQ3.3:** When does a configuration file change?
- **RQ3.4:** Do updates to an ASAT trigger developers to change their configuration file?
- **RQ3.5:** What versions of ASATs are currently being used?

We answer all of our research questions by means of a large scale analysis of open source software. For **RQ1**, we manually study project repositories and documentation to determine if they use code reviews or ASATs. We complement these results by sending a questionnaire to developers of these projects. To study the use of ASATs for **RQ2**, we examine the configurations of ASATs in open source projects. For **RQ3**, we investigate the changes made to these configurations.

Chapter 2

Related Work

In this chapter, we review the prior works that form the background of this thesis. First, we give an overview of the state of code review research and how this connects to our study on the prevalence of code reviews. Then, we review the research on ASATs and discuss how this is related to our study of the prevalence and the use of ASATs. Finally, because we use a defect classification to classify ASAT warnings, we give an overview of the classifications that ours builds upon.

2.1 Code Reviews

In 1976, Fagan [23] formalized a method of reviewing code by going through program modules on a line-by-line basis in a group meeting. This form of code review is called a *code inspection*. In the meetings, the focus was exclusively on finding defects, not discussing how to correct them [23]. Many researchers concluded that these code inspections can find a significant number of defects [68, 76], but many studies also showed that the inspection meeting had no discernible effect on the defect detection rate of inspections [19, 75]. In fact, the meetings often had a negative effect on the inspection procedure, as scheduling conflicts made it hard to set up meetings in a timely manner [37, 75]. Votta [75] indicated that these delays could run up to two weeks per meeting. If several meetings were necessary, a single inspection process could take months. These difficulties hindered the adoption of code inspections in industry settings [37].

The inefficiencies led to a desire for a more lightweight form of code inspection. This can be accomplished by sharing defects in an asynchronous manner, using the Internet or an intranet, rather than in a meeting [67]. Historically, this was done on mailing lists, but this has shifted towards issue trackers and purpose-built tools [29]. This form of review is often called *code review*. In contrast to code inspections, correcting defects is a part of the code review process [62, 67]. Developers can also discuss the change under review and try to solve problems with the author and the group of reviewers. In terms of the defect detection rate, Johnson and Tjahjono [38] and Perry et al. [57] observed that sharing defects in this manner has no negative effect on the number of defects found. More importantly, code reviews are often completed within two days [62, 67], saving a considerable amount of

time compared to formal code inspections. Intuition would suggest that, due to their ability to find a significant number of defects in a relatively short period of time, code review is prevalent among open source projects. In this study, we quantitatively investigate this claim.

Even though most developers and managers state that they consider finding functional defects the primary goal of code reviews [4], both Bacchelli and Bird [4] and Beller et al. [8] found that code reviews primarily find maintainability defects. Siy and Votta [72] and Mäntylä and Lassenius [51] observed similar results for code inspections. The reason for this might be that understanding the code under review is the main challenge in finding functional defects [4]. It causes inconsistencies in the number and type of defects found. How well someone understands the code will vary per reviewer, and, accordingly, Porter et al. [60] and Baysal et al. [7] observed that the largest source of variance in both code inspections and reviews is the reviewer. Thus, performing reviews regularly will improve the number of defects a review finds [52]. Therefore, in this study we additionally investigate whether or not the projects that perform code reviews do so consistently and continuously.

2.2 Automatic Static Analysis Tools

ASATs can be seen as a natural evolution of code inspections and code reviews. Instead of using human effort to find defects, ASATs use techniques such as data-flow analysis and control-flow analysis to find defects in code [18, 33]. However, because these techniques do not scale at large, abstractions have to be used [18]. These abstractions, plus the fact that checking common properties of programs is an undecidable problem [21], lead to *false positives*, which are warnings about defects that do not actually exist, and *false negatives*, which are absences of warnings when defects do exist in the program.

Whereas false negatives impact the effectiveness of ASATs because they miss defects in the code, false positives will cause developers to waste time because they have to analyze the warnings emitted by the tool. Deciding whether a single warning is a real defect or a false positive will cost a developer three to eight minutes [14, 32, 70]. This makes analyzing warnings a time consuming activity, especially considering there can be as much as 50 false positives for every accurate warning [41]. In general, there might be around 40 warnings per thousand lines of code [31]. This torrent of information can prove cumbersome to fully comprehend, and some developers have indicated that an overload of warnings is an important reason to refrain from using ASATs [36]. However, while researchers have studied the reasons why developers do or do not use ASATs, there is no hard data on the prevalence of ASATs in practice. In this study, we therefore quantitatively investigate the state of the adoption of ASATs in open source projects.

Whereas with code reviews the number and type of defects found varies with the reviewer, with ASATs this varies with the specific tool that performs the analysis. Many tools differ in the type of defects that they focus on, but even when tools focus on uncovering the same defects, the variance in defects found is still very large [21, 69, 77, 79]. These results indicate that using several ASATs has benefits over using just a single ASAT. However, this will increase 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. In this research, we aim to determine how many projects use multiple ASATs.

To better deal with a large number of warnings, several studies have investigated ways to rank warnings [30, 31, 43, 46, 70]. This has the advantage that a developer can decide how many warnings to analyze based on the derived importance of the warnings. In lieu of those ranking algorithms, developers can use configuration files to indicate which rules they consider to be important. This can reduce the number of warnings generated and suppress rules that are prone to emitting false positives. In this thesis, we analyze the configuration files of ASATs to discover the preferences of developers.

Another reason to study these preferences is to observe if the use of ASATs by developers reflects the potential of ASATs. Wedyan et al. [79] observed that 15% of all defects found by ASATs were functional defects, with all others being maintainability related. Ayewah et al. [3] and Wagner et al. [78] found similar results. Again, the results varied per ASAT, with some tools being better at finding functional defects than others [74]. In terms of the effectiveness at finding functional defects, a large number of studies observed that ASATs rarely find any functional defects [2, 13, 78, 79]. These results indicate that users of ASATs should expect to find maintainability defects with the tools and that most of the functional defects that the ASATs check for will be missed. In this thesis, we strive to determine if this is reflected in how developers configure their ASATs.

2.3 Defect Classifications

In 1993, the IEEE composed a standard for classifying defects [35]. The standard was the basis for a classification by IBM, the Orthogonal Defect Classification (ODC) scheme [12]. 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 [54, 81], several studies also conclude that the classification was too general and required adaptations to fit a particular use [8, 51, 77]. The specific ODC adaptation on which we based our classification started with El Emam and Wieczorek [20], who used the ODC scheme as a basis for a refined classification, which they showed had high inter-rater reliability. Mäntylä and Lassenius [51] then adapted this scheme to classify code review comments. Subsequently, Beller et al. [8] based their classification largely on this work. We further adapt these works because we classify ASAT warnings, or potential defects, instead of code review changes. Considering that ASATs primarily catch maintainability defects [2, 78, 79], we created an original classification for that class of defects, as neither the classification by Mäntylä and Lassenius [51], nor that of Beller et al. [8] had a sufficiently fine-grained classification for maintainability defects.

Chapter 3

Prevalence of Code Reviews and Automatic Static Analysis Tools

In this chapter, we investigate how common code reviews and ASATs are in open source projects. First, we outline the general research aim of the study and the high level study design. Then, we discuss the steps we took to execute this design. Afterwards, we take an in-depth look at projects on which we perform our study. Subsequently, we report on the results, and offer our interpretations in the discussion. Finally, we discuss the internal and external threats to the validity of the results and our interpretation.

3.1 Research Aim

In previous work on both code reviews and ASATs, we observed that there is no conclusive data regarding how prevalent these static analysis techniques are. For code reviews, Rigby et al. [67] stated that they are ubiquitous in open source settings. However, when searching for projects for a study of code review fixes, Beller et al. [8] observed that many projects did not perform code reviews in a consistent and continuous manner. However, no quantitative study had been performed to support either claim. As for ASATs, researchers are divided about their prevalence in software development. Ayewah et al. [3] stated in 2007 that ASATs started to become widely used in practice, but, more recently, Johnson et al. [36] and Kumar and Nori [48] stated the opposite. Still, none of these studies provided any data for their claims.

In light of this lack of data, we performed a quantitative study on the prevalence of both code reviews and ASATs in open source settings. Aside from investigating how common these quality assurance techniques are, we also set out to examine how they were used. To obtain the answers to these questions, we looked at popular open source projects and researched their use of code reviews and ASATs. More concretely, this study aims to answer the following research question and its subquestions:

RQ1: How common are static analysis techniques in practice?

- **RQ1.1:** What is the prevalence of code reviews?
- **RQ1.2:** What is the prevalence of ASATs?
- **RQ1.3:** Do projects use multiple ASATs to check the quality of their code?

3.2 Study Design

This section details the design of the study from the initial exploratory investigation, to the data collection, and the analysis of that data. The high level design of this study is shown in Figure 3.1.

In step 1, we select open source projects from code hosting services. In step 2, we filter these initial projects because not all repositories on code hosting services are software [40]. After filtering, we end up with the projects on which we perform our analysis. This analysis consists of two parts: collecting data from existing sources such as the code repository and the project website, and sending questionnaires to the developers of each project. For these two data sources, we set out to answer the same questions. Finally, in step 4, we juxtapose the results from both of these sources and investigate the differences between what the information in the existing sources indicates and the answers of developers regarding their code review and ASAT use.

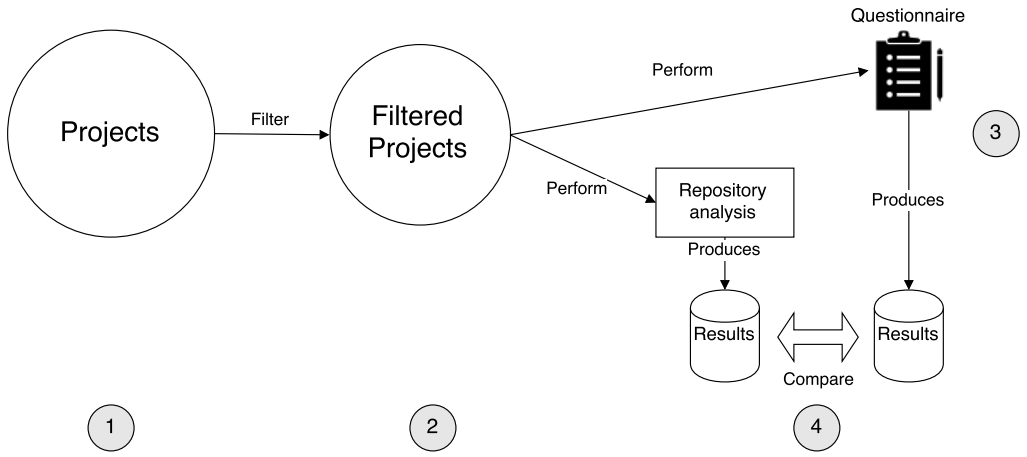

Figure 3.1: The study design for the prevalence analysis.

3.3 Study Methodology

In this section, we discuss how we executed the study design. First, we outline how we selected projects to study. Then, we discuss how we gathered information from project repositories, how we arrived at the questions for the questionnaire and the choice of partic-

ipants for our survey, and finally how we juxtaposed the data from the repository analysis with the results from the questionnaire.

Selection of Study Objects

To filter the large number of projects and repositories that we could potentially study, we selected projects based on their popularity. This was chosen as a criterion for several reasons. Most importantly, previous research on GitHub, the worlds largest code hosting service [27], showed that most projects have very few commits and are inactive [40]. While a random selection might better depict the quality assurance practices of the large tail of inactive and toy projects, the results of popular projects represent the static analysis practices of those open source projects that are well-known among developers.

Moreover, two-thirds of the repositories on GitHub are for personal use [40], as opposed to being used for collaboration purposes. Those repositories cannot have code review activity, as that would require more than one collaborator. Additionally, 93% of all projects have three contributors or less [40]. By selecting only the most popular projects, our assumption was that this would mostly include those projects with larger development teams, due to the increased interest in and use of these projects.

Study Procedure

We took a two-step approach to determine if a project used either code reviews, ASATs, or both. First, we collected data from existing sources. This corresponds to step 3 in Figure 3.1. For almost all projects, these sources consisted of the code repository and potentially the project website. Then, we sent out a questionnaire to developers of a project and asked them if they used code reviews or ASATs, as well as what their development practices were in regard to these quality assurance techniques. This corresponds to step 4 of Figure 3.1.

In this section, we first discuss in more detail how we collected data from existing sources and what we looked for while doing so. Then, we detail the questions that we asked the developers and our motivations behind the questions. Finally, we outline how we compared the data from these two sources.

Repository Analysis

To answer the research questions using data from existing sources, we studied as much data on a project as possible. The primary source of information was the code repository. Aside from the information obtained directly from the source code, many projects have documentation available in the repository itself. GitHub also provides a Wiki system, which some projects use for documentation purposes. Additionally, whenever possible, we studied the websites of projects and auxiliary development tools such as the mailing list and the issue tracker.

The development practices of a project could be described in various locations. Many GitHub projects have a *CONTRIBUTING.md* file in the root of their repository. The purpose of this file is to inform non-core contributors of the development practices of the core team

of developers. While it is possible that this information is solely meant for non-core contributors, we made the assumption that this documentation described the general development workflow in the project.

In some cases, an indication of the use of code reviews might be a link or reference to a code review tool such as Gerrit.[1] If such a tool is used and its database is public, we checked whether it showed any evidence of recent use. This was necessary because some projects claim to use code review tools, but have empty or inactive tool databases [8]. Finally, if there were no indications regarding the development workflow of the project in the documentation, we checked directly for the presence of code reviews from core developers, for example, in the form of pull requests on GitHub or patch submissions on the mailing list.

The references to ASATs in official project documentation proved to be rare. Therefore, we examined the code in project repositories to find hints that ASATs were used. For some languages and platforms, such as the Node.js runtime environment,[2] the dependencies of a project on third-party libraries and tools are listed in a file in or near the root of the code repository. Many ASATs can be included in this manner, which is an indication that they are used during development. Moreover, we scanned the repository for the presence of ASAT configuration files. If a project were to use an ASAT, it stands to reason that it would include a configuration file. This project wide configuration file is beneficial because it enforces a consistent analysis of the entire code. Use of an ASAT without a project wide configuration file can reasonably be seen as an indication that this ASAT is not used in a consistent manner, or relies on the default configuration. Finally, we searched for any occurrences of ASAT tool names in auxiliary project files. For GitHub projects, these are often the issue tracker and the pull requests. We made a distinction between core developers which used, mentioned, or recommended ASATs and non-core contributors which opened issues or pull requests due to ASAT warnings. These contributors might have used the ASATs on their own accord to enforce the quality of their changes, and therefore this use does not provide an indication of the project-wide ASAT use.

Questionnaire

Studying existing data is not enough to properly conclude that either code reviews or ASATs are used in a consistent and continuous manner. Developers can potentially bypass the official guidelines and submit changes either without submitting them for review or without using ASATs. Moreover, remnants of an ASAT might still be left in the project documentation or the repository, long after the tool has stopped being used. Therefore, to capture how these static analysis techniques were used, we sent out a questionnaire to either individual developers or mailing lists of the projects under study. We excluded those projects that had been inactive for more than a year at the time of this study. The full contents of the questionnaire can be found in Appendix A.

To decide who of the development team to send the questionnaire to, we examined the repository and website of the projects for contact information. When this information was

[1] `https://code.google.com/p/gerrit/`

[2] `https://nodejs.org/`

absent, we used the statistics provided by GitHub to find the developer who contributed the most code in the last year. For projects outside of GitHub, where we could not obtain information about the top contributors of a project, we scanned commit messages to decide whom to contact. We started out by only contacting one person, and only contacted a second developer if we did not get a response within a week. If the second developer did not respond, or if there was only one core developer for a project, we marked the project as *No Response*.

Regarding code reviews, we asked two simple questions. First of all, we inquired whether the projects used code reviews in a consistent and continuous manner for all developers, that is, for both core developers and non-core contributors. Finally, we were interested in which particular code review tools were used in the project. This last question was primarily to validate the choice for using multiple sources other than GitHub, as we expected the GitHub projects to overwhelmingly use the code review tools provided by GitHub, instead of using outside tools such as Gerrit. If this was indeed the case, then a heavy reliance on GitHub projects might have resulted in merely reflecting the use of code reviews in GitHub projects, rather than for more general open source software.

Concerning ASATs, we asked three simple questions. First, we inquired whether a project used ASATs to check the quality of their code. If they did, we asked if the results of the ASATs were a deciding factor in whether a change was deemed acceptable. Finally, we specifically inquired which ASATs were used in the project. The second question was the most important one, as it proved difficult to obtain an indication of how ASATs were used within a project. The presence of configuration files or ASAT dependencies from the repository analysis could not give any information regarding the specific use of ASATs, and the guidelines often did not mention ASATs at all. Therefore, even more so than with code reviews, the questionnaire was of vital importance to judge the use of ASATs in the projects under study.

Comparison of Results

The last step in our study was to compare the results that we collected from investigating existing data sources and the answers that we obtained from the questionnaire. We were interested in whether and where the results differed and why that might be. We wanted to know if the discrepancy came from the guidelines, in which case the core developers have a different workflow than the non-core contributors, or from hints in the code repository, meaning that there are remnants of old tools and development practices in the repository.

3.4 Study Objects

In this section, we discuss the study objects that we analyzed to asses the prevalence of code reviews and ASATs in open source projects. First, we discuss what type of projects we were looking for. Then, we go into more detail about the projects we selected.

We were looking for open source software projects. We did not place any restrictions on the type of software project, the programming language, or any other attribute of the project.

We found these projects by browsing various code hosting platforms and software directories. Because not every repository on these platforms is necessarily a software project [40], the only filtering that we did was to filter out all repositories that did not contain a software project. The reason for this filtering is simple: only software projects contain code, which means that only those development teams can perform code reviews and use ASATs.

We selected 140 projects from four different sources: GitHub,[3] OpenHub,[4] SourceForge,[5] and Gitorious.[6] We also considered other sources, primarily other code hosting services, but found them unfit for our purposes. Some were excluded because of a lack of projects, others because there was no way to filter projects on any distinguishing criteria. For example, GitLab[7] had a lack of projects, considering that the majority of the most popular repositories belonged to the GitLab company itself. On the other hand, the now discontinued Google Code[8] had no way to easily find or rank projects.

We divided the number of projects over the selected code hosting services as follows: 100 from GitHub, 10 each from OpenHub and SourceForge, and 20 from Gitorious. This distribution was chosen because of the relative popularity of GitHub to the other platforms [27]. In the case of OpenHub, we could not select more than 10 projects because OpenHub did not provide popularity information for more than 10 projects.

For GitHub, OpenHub, and SourceForge, we selected the projects based on the criteria as established in Section 3.2: their relative popularity. For GitHub, this popularity is reflected in the number of stars that a project has. The star system allows users to indicate their interest in a project. A high number of stars therefore indicates that the GitHub community considers this project noteworthy, which can be seen as a heuristic for project popularity. OpenHub has a top 10 of popular projects on their website, which is based on the number of users that a project has. This metric is similar to GitHub stars because users have to manually indicate that they use a project, rather than that the popularity is automatically calculated from actual use metrics. For SourceForge, we selected the 10 most downloaded projects of all time. Gitorious did not have a way to rank projects on popularity. Their only ranking system was that of development activity. Because this still alleviated our greatest concern when selecting projects, which is that most repositories are inactive [40], we decided to select 20 projects based on this metric.

After filtering non-software repositories and projects that appeared on multiple platforms, we were left with 122 projects. The largest number of projects excluded, 17 in total, came from GitHub, with just one project being excluded from OpenHub and none from the other two platforms. The exclusions from GitHub were all non-software projects, while the OpenHub exclusion was the Linux kernel which we already included from GitHub. The full list of projects can be found in Appendix A. The full list includes three projects that responded when we posted our questionnaire on the mailing list of another project, and one project mailing list on which we received two answers for separate parts of the project with

[3] https://github.com
[4] https://openhub.net/
[5] http://sourceforge.net/
[6] https://gitorious.org/
[7] https://about.gitlab.com/
[8] https://code.google.com/

different workflows. We excluded 18 projects from the questionnaire that had been inactive for more than a year. This brought the total number of projects for the questionnaire to 108.

3.5 Results

In this section, we discuss the results of studying the prevalence of static analysis techniques in open source projects. First, we present the results of the analysis on existing data such as repositories and project websites. Then, we review the responses from the questionnaire sent to developers or mailing lists. Finally, we juxtapose these two result sets and identify the ways in which they differ.

Repository Analysis

Regarding code reviews and **RQ1.1**, the results of analyzing the data in project repositories and websites are shown in the third column of Table 3.1. In total, 94 out of all 122 projects use code reviews in their development workflow. The full results are presented in Appendix A. Reviewing the results for each code hosting service separately, we see that 68 out of 83 GitHub projects either have a clear development workflow that includes a code review before submitting changes to the repository or core developers who submit a sizable portion of their code via pull requests. All projects, except for the Linux kernel[9], perform these code reviews via GitHub pull requests, in rare cases supplemented by other code review tools such as Phabricator.[10] The other 15 projects have no development guidelines or make no mention of code reviews in their workflow. Moreover, core developers are mostly or completely absent as submitters of pull requests. These core developers only interact with the pull requests opened by non-core contributors. A few projects have barely any pull request activity at all. On OpenHub, all 9 projects use code reviews. Four of them perform code reviews in multiple places, such as on a bug tracking tool and a mailing list. On SourceForge, half of the projects have no information about code reviews in their documentation or any other indications of code review activity. On the whole, the development practices of the SourceForge projects are not as well documented as those from GitHub or OpenHub. This might be because the popularity rating on GitHub and OpenHub reflects the interest of software developers, while the popularity on SourceForge is a reflection of the number of downloads by end-users, who are not necessarily software developers. Finally, 12 out of 20 Gitorious projects feature code reviews in their development workflow. One other project, Chakra Packages from the similarly named operating system[11] mentions the use of a code review tool, ReviewBoard. However, just three reviews were posted in the entirety of 2014. We therefore concluded that we could not conclusively determine whether this project used code reviews.

For **RQ1.2** and **RQ1.3**, the results from analyzing the information in project repositories and websites regarding ASATs are shown in the fourth and fifth column of Table 3.1. The

[9] `https://kernel.org/`
[10] `http://phabricator.org/`
[11] `http://chakraos.org/`

Source	# of Projects	Using Code Reviews	Using ASATs	Using Multiple ASATs
GitHub	83	82%	64%	30%
OpenHub	9	100%	89%	22%
SourceForge	10	50%	30%	0%
Gitorious	20	60%	35%	5%
Total	122	77%	59%	23%

Table 3.1: Summary of the **repository analysis** results regarding the prevalence of code reviews and ASATs.

fourth column shows how much projects use one or more ASATs to check their code. The fifth column only considers those projects that use multiple ASATs. Overall, 72 out of all 122 projects either mention the use of ASATs in official project documentation, or the repository contains traces of ASATs, either as configuration files or as explicitly mentioned dependencies. From those projects, 28 use multiple ASATs to check their code. The full results are available in Appendix A. 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 SourceForge projects, with only three of them using ASAT, all working with a single tool. Finally, 7 out of 20 Gitorious projects use ASATs, but only one of them uses multiple ASATs.

Questionnaire

The summarized results of the questionnaire are shown in Table 3.2. The third column shows the percentage of projects that use code reviews to verify changes from core developers. The fourth column displays the percentage of projects that use one or more ASATs to check their code, while the fifth column shows the percentage of projects that use multiple ASATs. Finally, the last column shows displays what percentage of projects uses the results of these tools as one of the factors to decide whether a change is accepted. This last column thus displays information that we could not ascertain from the repository analysis.

For **RQ1.1**, Table 3.3 shows the expanded results concerning the questions about code reviews. From all the 36 projects that responded, 19 perform code reviews on changes from core developers. Another four projects state that code review is not required for every change, but highly encouraged. In four projects, the core development team consists of just a single person, and no code review is performed on the changes made by that one developer. For the other 11 projects, code review is only performed on contributions made by non-core contributors. In total, excluding the projects that only have one core developer, almost 72% of projects use code reviews to validate changes by core developers, and almost 82% of those projects mandate this for every change.

Many respondents of projects that enforce the use of code reviews for every change note that there are some exceptions to this rule. Often mentioned is that developers can forgo

Source	# of Projects	Code Reviews on Changes from Core Developers	Using ASATs	Using Multiple ASATs	Enforced Use of ASATs
GitHub	19	42%	68%	32%	42%
OpenHub	1	0%	0%	0%	0%
SourceForge	3	0%	100%	66%	0%
Gitorious	10	50%	70%	40%	30%
Other	3	66%	100%	66%	33%
Total	36	53%	77%	36%	36%

Table 3.2: Summary of the **questionnaire** results regarding the prevalence of code reviews and ASATs.

a code review for small changes, such as a correction of a typographical error. For most projects, this exception is possible because core developers still have the ability to directly push their changes to the repository. However, even when using a tool that prohibits a direct push to the repository, a core developer can often still force their changes through without undergoing any review. This is possible if the author of a review also has the ability to approve their own change. When forcing changes to the repository, it remains to be seen if another developer will perform a post-commit review. Therefore, it is doubtful that any project has reviewed every single change made, however small it may be.

Regarding the tools that are used, the GitHub projects use the built-in pull requests functionality of their code hosting service across the board. Only React[12] and Django[13] use a supplementary tool other than these pull requests, those being Phabricator and Trac[14] respectively. There is more variation in tool use with the projects from OpenHub, SourceForge, and Gitorious. This includes code review specific tools, such as Gerrit, and issue trackers, such as Mantis.[15] In total, only seven projects have more than one place to submit code reviews. One other project, Snowdrift,[16] only uses two places for code reviews because of their migration from Gitorious to GitHub. A little under half of the projects do not use purpose-built tools at all, instead performing code review on a mailing list or forum.

For **RQ1.2** and **RQ1.3**, Table 3.4 shows the expanded results concerning the questions about ASAT use. It details whether a project uses ASATs, if the results of these tools are decisive in whether or not a change gets accepted or not, and finally which tools a project uses.

In general, we observe that ASATs are being used by 77% of all projects. Slightly less than half of the projects that use ASATs, 13 out of 28, also place strict requirements on new code. Generally, this means that code that is submitted for review or is pushed to the repository cannot cause a regression with regards to the results of the ASATs, as this

[12] `https://facebook.github.io/react/`
[13] `https://djangoproject.com/`
[14] `http://trac.edgewall.org/`
[15] `https://mantisbt.org/`
[16] `https://snowdrift.coop/`

Project Name	Code Reviews on Changes from Core Developers	Code Review Tools Used
Ace	Yes	GitHub
AppArmor	Yes	Mailing list
Async	Yes	GitHub
Brackets	Yes	GitHub
Clang	Yes	Mailing list, Phabricator
CMake	Yes	Mailing list
Diaspora	Yes	GitHub
GitLab	Yes	GitLab, GitHub
GROMACS	Yes	Gerrit
io.js	Yes	GitHub
Leaflet	Yes	GitHub
Modernizr	Yes	GitHub
OpenOCD	Yes	Gerrit
openSUSE Factory	Yes	Open Build Service
openSUSE YaST	Yes	GitHub
PDF.js	Yes	GitHub
Qt	Yes	Gerrit
React	Yes	GitHub, Phabricator
Snowdrift	Yes	GitHub, Gitorious
Django	Encouraged	GitHub, Trac
GDB	Encouraged	Mailing list
LibreOffice	Encouraged	Gerrit
Pure	Encouraged	GitHub
Bash	One core developer	Mailing list
Semantic-UI	One core developer	GitHub
Slick	One core developer	GitHub
Textmate	One core developer	GitHub
Abilian	No	GitHub
CodeIgniter	No	GitHub
Dungeon Crawl Stone Soup	No	Mantis, Gitorious
Elasticsearch	No	GitHub
FileZilla	No	Forum, Trac
Haiku	No	Mailing list, Trac
Lime	No	GitHub
VLC Media Player	No	Mailing list, Patchwork
Vuze	No	Forum

Table 3.3: Results for the code review prevalence questionnaire.

Project Name	ASATs	Enforced	Tools Used
Brackets	Yes	Yes	JSHint, JSLint, JSONLint
CMake	Yes	Yes	Clang
Elasticsearch	Yes	Yes	FindBugs, Checkstyle, PMD
GROMACS	Yes	Yes	Clang, Cppcheck, custom checker
io.js	Yes	Yes	CPPLint, Closure Linter
Leaflet	Yes	Yes	ESLint
Lime	Yes	Yes	Gofmt
Modernizr	Yes	Yes	JSHint, JSCS
openSUSE Factory	Yes	Yes	RPMlint
openSUSE YaST	Yes	Yes	RuboCop
PDF.js	Yes	Yes	JSHint
Pure	Yes	Yes	CSSLint
Snowdrift	Yes	Yes	HLint
Abilian	Yes	No	JSLint, ESLint, CSSLint, Pep8, Pyflakes
AppArmor	Yes	No	Pyflakes
Async	Yes	No	JSCS, ESLint
Django	Yes	No	Flake8
Dungeon Crawl Stone Soup	Yes	No	Clang
FileZilla	Yes	No	Coverity Scan, Cppcheck
GitLab	Yes	No	Hound
Haiku	Yes	No	Coverity Scan, custom checker
LibreOffice	Yes	No	Coverity Scan, Cppcheck, Clang
OpenOCD	Yes	No	Clang
Qt	Yes	No	Coverity Scan, Clang
React	Yes	No	JSHint, Code Climate
Semantic-UI	Yes	No	JSHint
VLC Media Player	Yes	No	Coverity Scan, Clang
Vuze	Yes	No	Coverity Scan
Ace	No	—	—
Bash	No	—	—
Clang	No	—	—
CodeIgniter	No	—	—
Diaspora	No	—	—
GDB	No	—	—
Slick	No	—	—
Textmate	No	—	—

Table 3.4: Results for the ASAT prevalence questionnaire.

can be viewed as a regression of the overall quality of the system. One project, Abilian,[17] mentions that they plan to enforce such requirements as well, but they first want to make the existing code base completely free of warnings. For the other projects, most respondents state that ASATs are only sporadically used by developers whenever they believe they need to validate the quality of their code.

Regarding the ASATs used, we observe from Table 3.5 that the Coverity Scan tool, the Clang static analyzer, and JSHint are popular among respondents. The other tools are all used less than three times. However, this prevalence per tool is dependent on the programming languages that were used among respondents. Therefore, one can only compare tools within the same language.

Tool	Analyzable Languages	Occurrences
Coverity Scan	C/C++/C#/Java	6
Clang	C/C++/Objective-C	7
Cppcheck	C++	3
Cpplint	C++	1
Hound	Coffeescript / JavaScript / Ruby / SCSS	1
CSSLint	CSS	2
Gofmt	Go	1
HLint	Haskell	1
Checkstyle	Java	1
FindBugs	Java	1
PMD	Java	1
Closure Linter	JavaScript	1
ESLint	JavaScript	3
JSCS	JavaScript	2
JSHint	JavaScript	5
JSLint	JavaScript	2
Code Climate	JavaScript / PHP / Python / Ruby	1
JSONLint	JSON	1
Flake8	Python	1
Pep8	Python	1
Pyflakes	Python	2
rpmlint	RPM package files	1
RuboCop	Ruby	1

Table 3.5: Occurrences of each ASAT in the questionnaire, grouped by the languages that a particular ASAT analyzes.

Concerning **RQ1.3**, 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 is the only project that does use more than three ASATs, but they are divided among three languages: JavaScript, CSS, and Python.

[17] https://abilian.com/

Only one other project, Brackets,[18] also checks for defects in multiple languages: JavaScript and JSON. All other projects either only use a single ASAT or multiple ASATs that check for defects in the same language.

Comparison of Results

To compare the information from both sources, we juxtaposed specific data points from the repository analysis with the results from the questionnaire. This information differs in some way for close to 20% of the projects that responded to our questionnaire. For some projects, the repository analysis indicated that the project used code reviews, while the questionnaire indicated that no code review was performed on changes from core developers. Other differences are related to the use of ASATs. Table 3.6 shows the projects for which the two sources of information show a discrepancy regarding the use of ASATs. We can see that in three cases, the analysis of project information 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[19] mentions that they previously used Coverity,[20] 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 do not use the ASAT that was present in project information, but instead they use a different ASAT entirely. 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.

3.6 Discussion

In this section, we discuss and interpret the results from the previous section and put them in a broader perspective. First, we take an in-depth look at the differences between the results of the repository analysis and the answers of the questionnaire. Then, we answer the research questions on the basis of our results. We present our main findings in boxes to highlight them.

Regarding code reviews, excluding the projects that responded to the questionnaire, 87% of repositories, websites, and other sources of information indicate the use of code reviews, while 63% of respondents mention the use of code reviews among core developers. Using a two sample t-test, we confirm that this difference is significant at $\alpha = 0.05$: $t(123) = 3.043$, $p = 0.0029$.

The most likely cause for this difference is that when we performed the repository analysis, we made the assumption that contribution guidelines for non-core contributors accurately reflect the development workflow of core developers. However, the questionnaire showed that there are many projects that noted that they did not perform code reviews for

[18] http://brackets.io/

[19] https://gnu.org/software/bash/bash.html

[20] https://coverity.com/

Project Name	ASATs in Repository	Tools Actually Used
Async	JSLint	JSCS, ESLint
Bash	Coverity Scan	—
CMake	Coverity Scan	Clang
Diaspora	JSHint	—
GDB	Coverity Scan	—
GitLab	Hound, RuboCop	Hound
Semantic-UI	JSHint, CSSLint	JSHint
Brackets	JSHint, JSLint	JSHint, JSLint, JSONLint
Dungeon Crawl Stone Soup	—	Clang
Filezilla	Coverity Scan	Coverity Scan, Cppcheck
LibreOffice	Coverity Scan	Coverity Scan, Cppcheck, Clang
Modernizr	JSHint	JSHint, JSCS
Qt	Coverity Scan	Coverity Scan, Clang
React	JSHint	JSHint, Code Climate
VLC Media Player	Coverity Scan	Coverity Scan, Clang

Table 3.6: Discrepancies between project repository information and answers from the questionnaire.

changes from core developers. Thus, the contribution guidelines only applied to non-core contributors. Therefore, we conclude that we can only judge the code review practices of a project in regard to the core developers with the results from the questionnaire.

Regarding ASATs, the percentage of projects from the questionnaire that use ASATs is higher than that of the repository analysis. Excluding projects that responded to the questionnaire, 52% of existing project resources suggest the use of ASATs, compared to 77% of the questionnaire respondents who note that ASATs are used. This difference is significant at $\alpha = 0.05$: $t(123) = 2.364$, $p = 0.0196$.

Looking more closely at the differences, we see that, concerning the questionnaire, there are three projects that do not use ASATs even though their repositories seem to indicate otherwise. Moreover, there was just a single project that indicated in response to the questionnaire that they used ASATs, but were marked by the repository analysis as having no evidence of ASAT use. Therefore, we conclude that it is unlikely that we have missed something in the analysis of existing data which could have caused us to erroneously conclude that some projects did not use ASATs. Thus, it seems more likely that the subset of respondents was simply more inclined to use ASATs. We therefore conclude that the results of the repository analysis seem to be valid as an upper bound of ASAT use in open source projects.

However, the results from the questionnaire showed that 15 out of 28 projects that use ASATs only run these tools sporadically and without enforcing their use. This means that one cannot infer the precise use of ASATs from a repository analysis, but only the intention, past or present, to use ASATs. More generally, this demonstrates that one should be careful when using information solely from project repositories, websites, and documentation when

drawing conclusions about whether a project uses a tool or technique and how they use them. These results support those from Negara et al. [55], who observed that one needs more than just the data in repositories to accurately study the evolution of software. Directly contacting developers is always necessary to validate and support the results from data that is collected from existing sources.

> How a project uses a static analysis technique or tool cannot be inferred from a repository analysis alone.

Regarding code reviews, and **RQ1.1**, the questionnaire showed that 53% of all projects perform a code review for every change of a core developer. A further 10% of projects are more lenient, and encourage, but do not mandate, code review for every change. The rest of the projects only review changes made by non-core contributors. A reason for this might be that changes made by core developers are more likely to be of a higher quality than those of non-core contributors [63].

> Code review for changes from core developers is common, but not ubiquitous in open source projects.

For the 10% of projects that only encourage code review for changes of core developers, it is up to the author himself to decide whether his change is significant enough to warrant code review. This seems antithetical to the nature of code reviews. After all, the basis of reviewing changes is that other developers can find errors in code that the author considered good enough for submission. And research has shown this to happen in a significant number of cases [67, 68, 76]. Furthermore, for the 53% of projects that perform a code review for every change, many respondents note that there is an exception for small changes. The reasoning of many respondents is that these alterations do not have enough impact to warrant review. Moreover, we might assume that many of the other beneficial factors of code review, such as knowledge dissipation [4, 9] and team awareness [4], are absent because of the size of the change. If the rules on what qualifies as an exception are clear and complete, projects can avoid having the author determine what changes warrant review.

> The majority of projects that perform code review for changes from core developers do not mandate this for every change.

Regarding ASATs and **RQ1.2**, the repository analysis showed that less than 59% of projects use ASATs in various levels of strictness. These results seems to contradict the experiences of Johnson et al. [36] and Kumar and Nori [48], who claim that ASATs have not achieved significant use among software projects.

ASATs are common, but not ubiquitous in open source projects.

However, the questionnaire showed that the way in which ASATs are used can vary. 64% of projects use ASATs sporadically and without attaching any consequences to the warning results. Therefore, we can conclude that most projects do not have ASATs integrated into their development workflow. Research suggested that an important factor of improving the adoption of ASATs was to make this integration as easy and seamless as possible [36]. The results show that the potential is there, since most projects use code review tools that ASATs can integrate into. For instance, Coverity provides both GitHub and Travis[21] integration [15, 16], making it easy to integrate static analysis into a code review workflow. This does mean that the integration and use of ASATs is dependent on the development practices concerning code reviews.

Many project that use ASATs only run the tools occasionally and without attaching strict consequences to the tool results.

Concerning **RQ1.3**, we observed that 44 out of 72 projects use one ASAT to check their code. This is in spite of the fact that the use of multiple ASATs can provide a large increase in defect detection capabilities [21, 69, 77, 79]. Maybe those developers are unaware of these benefits, or they do understand them, but do not act upon them. The reason for this inaction might be straightforward. An overload of warning messages has been identified as one of the primary reasons for developers to avoid using ASATs [36]. It is clear that for every ASAT that developers add to find more defects, they also increase the overload of warning messages. Thus, there might only be an increase in the use of multiple ASATs when the total number of warnings stays manageable.

Among projects that use ASATs, the majority uses a single ASAT to check their code.

3.7 Threats to Validity

In this section, we discuss the aspects that could threaten the validity of this study on the prevalence of code reviews and ASATs in open source software, and we show how we endeavored to mitigate these threats. First, we discuss the internal threats to validity, and then we review the external threats.

[21] `https://travis-ci.org/`

Internal Validity

Internal threats are those factors that could have an impact on the analysis of the study objects and the conclusions drawn from that analysis. There are two internal threats that we identified and that we sought to mitigate.

- A heavy dependency on information from GitHub, with 100 projects from that source to 40 from other sources, might produce a bias in our results. However, given the relative popularity of GitHub to the other code hosting services [27], and the migration of many to projects to GitHub [27], we argue that the totality of GitHub projects accurately reflects the open source landscape. To confirm this, we compared the results from GitHub with those from the other sources. We wanted to know if the number of projects either using code reviews or ASATs differed significantly between these sources. We performed a t-test with the questionnaire data for code reviews and the full results for ASATs. The outcome showed that the difference was not significant at $\alpha = 0.05$ (code reviews: $t(34) = 0.061$, $p = 0.9519$; ASATs: $t(124) = 1.511$, $p = 0.1334$). Therefore, we can conclude that the heavy reliance on GitHub projects did not affect the results.
- The Hawthorne effect [1] is not applicable to our analysis, as we did not directly observe developers to see if they used code reviews or ASATs. However, one might argue that developers that responded to the questionnaire would want to paint a favorable view of their development practices [25], and would be unlikely to admit that their quality assurance practices are insufficient or even non-existent. However, we do not believe this to be a threat, as developers were under no obligation to respond to our survey and share information about their development practices, and because we could use the data from the repository analysis to verify those projects for which developers potentially painted a misleading picture of their development practices.

External Validity

External threats are those that affect the generalizability of our results. We identified several external threats to this study.

- One could argue that given the enormous number of open source software projects, with more than 22.9 million repositories on GitHub alone [26], that 122 projects is too small of a sample size. However, to the best of our knowledge, it is the largest known manual analysis of projects repositories and websites. For the questionnaire, we observed a response rate of almost 30%, which is higher than the normal response rate for email-only questionnaires and surveys [42, 61].
- Given that this study only considers open source projects, its generalizability towards closed source projects might be limited. Research has shown that in open source projects, the core team members have to provide a balance between attracting new developers and having strict, and potentially slow, quality assurance practices [28]. We cannot readily assume that this balancing act also exists in closed source development projects, especially those where developers are paid for their services.

Therefore, the quality assurance practices in open source development might differ substantially from those in closed source development.

- Regarding the generalizability of the results towards different types of open source projects, we do not expect this be an issue. The large number of repositories from four different sources featured a diverse set of projects, both in terms of type and languages used. This can be seen in Appendix A. Of course, ASAT prevalence is dependent on the state of tools for that language and their maturity. Therefore, this can vary greatly among languages. However, we expect these results to be generalizable for other languages with a mature set of tools.

Chapter 4

Configuration of Automated Static Analysis Tools

Having obtained an understanding of how often developers in open source projects use ASATs, in this chapter we investigate how developers configure ASATs by means of their configuration files. First, we outline the research aim and the high level design of the study. Then, we discuss how we enacted this design. Afterwards, we examine both the ASATs under study and the configuration files of ASATs. Subsequently, we outline the results of the study and offer our interpretation in the discussion. Finally, we discuss the threats that could undermine the validity of the study and its results.

4.1 Research Aim

Much research has been done on the performance of ASATs, both in terms of the defects that they detect or miss [2, 13, 78, 79], and on the number of false positives that they generate [31, 36, 41, 48]. However, to the best of our knowledge, there has not been a large scale quantitative study on how developers use ASATs in practice. In light of this lack of knowledge, we endeavor to investigate the use of ASATs by open source developers through studying the configuration files that they use. More concretely, we aim to answer the following research question and its subquestions:

RQ2: How are ASATs used?

- **RQ2.1:** What type of warnings do developers enable?
- **RQ2.2:** What type of warnings do developers disable?
- **RQ2.3:** Do default configurations reflect the wishes of developers?
- **RQ2.4:** How prevalent are custom rules in the configurations of developers?

In this thesis, we study configuration files of ASATs to obtain an understanding of how developers use ASATs. We do this because developers can configure the ASAT to fit their specific needs by using a configuration file. In such a file, developers can enable the rules that check for defects that they consider important, and disable rules because they are either not interested in those defects or because those rules generate too many false

positives. Without a configuration file, developers rely on the default configuration of the tool, which might not be in line with their specific needs. Thus, we consider the contents of an ASAT configuration file to be an important indicator of how developers use ASATs to check for defects in their code. However, it cannot tell us how often developers run the ASATs and what they do with the results. We obtained that information in Chapter 3, by directly contacting developers.

Another reason to study configurations files is because, for many ASATs, this configuration is stored in a specific file that the ASAT looks for before starting its analysis. This file can then be stored in the project repository. This is especially important when collaborating with other developers, because it enforces a consistent analysis of the whole codebase. This means that we can readily retrieve the configuration files of ASATs for open source projects, which makes them a perfect fit for a large scale analysis.

4.2 Study Design

This section details the design of the study from the initial exploratory investigation, to the data collection, and the analysis of that data. The high level design of the study is shown in Figure 4.1.

To perform our analysis, we first develop a program that can analyze the configuration files of ASATs. Moreover, we create a defect classification that can be applied to the rules of any ASAT. We apply the analysis and the classification to configuration files that we retrieve from various code hosting services. The results are classified distributions of warnings that capture how developers configure their ASATs. Finally, we answer our research questions on the basis of these distributions.

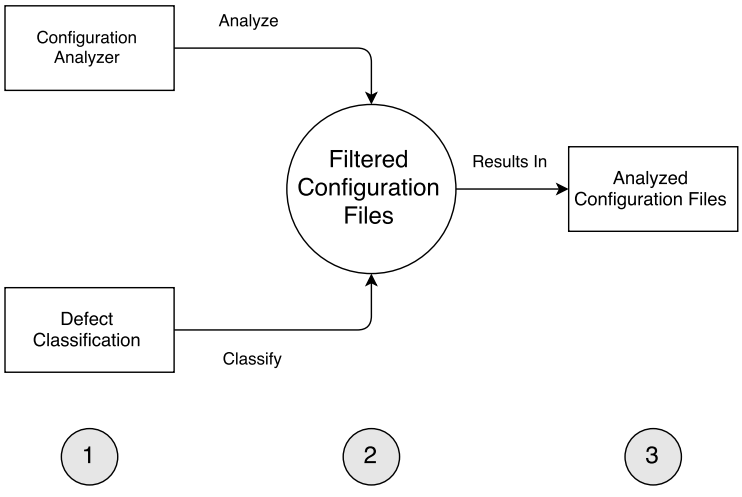

Figure 4.1: The study design for the analysis of ASAT configuration files.

4.3 Study Methodology

In this section, we outline how we executed the study design. First, we explain what information in a configuration file we were interested in. Then, we discuss how we built a classification for ASAT warning rules. Finally, we outline how we used that classification to answer our research questions.

We were interested in the directives in a configuration file that specify which warning rules to enable or disable. Many ASATs also have a second type of directive present in a configuration file. These affect the behavior of an ASAT that is not related to defect detection. An example of such a directive is one that specifies in what format to write the results to an output file. These were of no interest to us, primarily because we could not easily generalize over these types of directives. Therefore, we only considered those directives that concern warning rules. From this point forward, we will simply refer to those directives as *rules*.

Before we could analyze the configuration files, we first needed a way to generalize over all the tools. While some ASATs might check for the same type of defects, they often name their rules differently. Grouping all the rules from different ASATs that check for the same, specific defect into individual rules would result in a categorization that would be too fine-grained. Moreover, ASATs that check for the same rules often do not detect the same defects in the same code [21, 69, 77, 79]. Therefore, it might be that the tools ostensibly check for the same defect, but that in practice they detect a distinct subset of that defect. To address these issues, we created a higher level classification of defects, akin to the adapted ODC scheme [12] by Mäntylä and Lassenius [51]. Because that classification presented an inadequately fine-grained categorization of maintainability defects, we created our own classification for these defects. We did this by means of a two-person *open card sort* [5]. The card sort was performed by the author of this thesis and the thesis supervisor.

In an open card sort, there are no predefined categories at the start of the sorting process. We preferred this to a closed card sort because, as stated before, we did not want to reuse a previous classification. We used the warning rules of one ASAT as "cards", that we grouped into categories that emerged during the sorting process. Afterwards, both participants agreed on the final categories based on the individual classifications.

To classify the rules of ASATs using our classification, we used the documentation of the ASATs to determine what a specific rule checks for. In accordance with previous work, when we could classify a rule as belonging to either a functional or a maintainability category, we classified it as a functional defect [8, 51].

For **RQ2.1** and **RQ2.2**, 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 to be important by developers, and conversely, which types of warnings they avoid, possibly because they do not consider them to be important, because they generate too many false positives, or on account of those rules performing poorly at finding real defects. 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 see more clearly the relations between a category with a large number of rules and another with just one or two rules.

For **RQ2.3**, we made the assumption that the creators of the tool created a default configuration that, in their opinion, would be suitable for the majority of their users. Therefore, we were interested to see if and how the configurations of developers deviated from these default configurations, as these are indications of whether the default configuration accurately reflects the wishes of ASAT users.

We identified three possible ways in which a developer can deviate from the default configuration:

- A developer disables a rule that was enabled in the default configuration.
- A developer enables a rule that was disabled in the default configuration.
- A developer reconfigures a rule that was enabled in the default configuration.

Not all rules can be reconfigured. An example of a configurable rule is a one which checks the names of variables, functions, and classes to see if they adhere to a certain protocol. A developer can then customize the naming convention that the tool checks for, often in the form of a regular expression. Reconfiguring a rule indicates that developers want to check for this convention, but do not agree with the content of 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 **RQ2.4**, 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 tool 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.

4.4 Study Objects

In this section, we discuss the study objects that we used to asses how ASATs are configured in open source projects. First, we discuss what types of ASATs we were looking for. Then, we examine the selected ASATs in detail. Finally, we review the configuration files that we perform the actual analysis on.

ASATs Under Study

Because, we intended to perform this analysis on ASAT configuration files, 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. This is necessary because we could not consistently extract the configuration information if the ASAT needs to be configured in another way, for instance with 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. This is required because it is impossible to consistently and completely extract information from configurations in other forms, such as shell scripts.

When searching for ASATs, we used the tools that we encountered during the prevalence analysis in Chapter 3 as a starting point. We expanded our search with search engines and programming support sites such as Stack Overflow.[1] We did this by explicitly searching for alternatives for the already encountered tools, or by searching for a static analyzer for a specific programming language. For each tool that we found, we investigated if and how its behavior could be configured.

Table 4.1 lists the nine tools which fit the criteria that we set. From this table, we can see that most of the tools use standard formats to store their configuration information. Two tools, JSL and Pylint, use key-value pairs in plain text format for their configuration. 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 a specific 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.

Tool	Analyzed Language	Configuration File Language	Extendable	First Release	# of Rules
Checkstyle [11]	Java	XML	Yes	2001	179
FindBugs [24]	Java	Text	Yes	2003	160
PMD [59]	Java	XML	Yes	2002	330
ESLint [22]	JavaScript	JSON	Yes	2013	157
JSCS [39]	JavaScript	JSON	Yes	2013	116
JSHint [44]	JavaScript	JSON	No	2011	253
JSL [53]	JavaScript	Text	No	2005	63
Pylint [49]	Python	Text	Yes	2006	390
RuboCop [6]	Ruby	YAML	Yes	2012	221

Table 4.1: Basic information for the ASATs that are analyzed in this chapter.

[1] `https://stackoverflow.com/`

One things 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 [45]. 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.

Table 4.2 lists the latest version of the tools at the time of the analysis in mid-March of 2015. New versions of a tool could introduce or remove rules from the tool. Therefore, the results for one version of a tool might not be representative for another version, depending on the size and gravity of the changes. Thus, the table lists the versions for which our results are valid.

Tool	Latest Version at Time of Analysis	Released On
Checkstyle	6.4.1	2015/03/04
FindBugs	3.0.1	2015/03/08
PMD	5.2.3	2014/12/21
ESLint	0.16.1	2015/03/08
JSCS	1.11.3	2015/02/11
JSHint	2.6.3	2015/02/28
JSL	0.3.0	2006/11/03
Pylint	1.4.1	2015/01/16
RuboCop	0.29.1	2015/02/13

Table 4.2: The latest version of an ASAT at the time of the analysis in mid-March of 2015.

Configuration Files Under Study

After selecting the ASATs to study, we needed to retrieve as many configuration files as possible for every tool. We expected to find enough configuration files on GitHub. However, to further augment the study and to have as much data as possible, we also collected data from Krugle[2] and OpenHub. The use of OpenHub provided a small complication because the projects that were listed on OpenHub could host their code on GitHub. To eliminate possible duplicates, we excluded OpenHub results which hosted their code on GitHub.

The number of configuration files that we retrieved are shown in Table 4.3. We can see that OpenHub provides a sizable number of configuration files for the Java tools and JSHint. For the other tools, the increases in the number of configuration files are minimal. The

[2] `http://krugle.org/`

number of added files from Krugle is minimal for all tools. Moreover, for some tools, there were many more configuration files hosted on GitHub. However, we were unable to retrieve those because of limitations in the GitHub search system. Thus, there is a discrepancy between the total number of configuration files hosted on GitHub and the number of files that we retrieved.

Tool	GitHub	OpenHub	Krugle	Total
Checkstyle	16271	2492	22	18785
FindBugs	1575	514	1	2090
PMD	5562	1888	8	7458
ESLint	4427	5	3	4435
JSCS	11656	20	1	11677
JSHint	105619	3086	65	108770
JSL	862	0	0	862
Pylint	3941	123	7	4071
RuboCop	10063	0	3	10066
Total	159976	8128	110	168214

Table 4.3: Number of configuration files for each ASAT, grouped by source.

4.5 Results

In this section, we discuss the results of analyzing how ASATs are configured. First, we examine the classification used to categorize warning rules. Then, we review which types of defects are enabled or disabled in the configuration files of developers. Subsequently, we investigate if and how the configurations of developers deviate from the default configurations. Finally, we examine how prevalent custom rules are in the configurations of developers.

Classification

Figure 4.2 shows our refined classification for ASAT rules. We observe that any defect is first placed into one of two categories. It is either a functional defect if it can be the cause of a program failure, or a maintainability defect if it cannot.

The functional defect classification closely follows that of Mäntylä and Lassenius [51]. There are two categories from that classification that are absent in ours. The *Larger Defect* category was removed because ASATs check for single, well-defined defects. The *Support* category was merged in the *Interface* category, as there were almost no rules that fit in that category. Additionally, we renamed *Timing* to *Concurrency*, as we felt that better reflected the category. Finally, we added the *Migration* category, for those rules that check for instances where a migration between two programming language versions can cause defects.

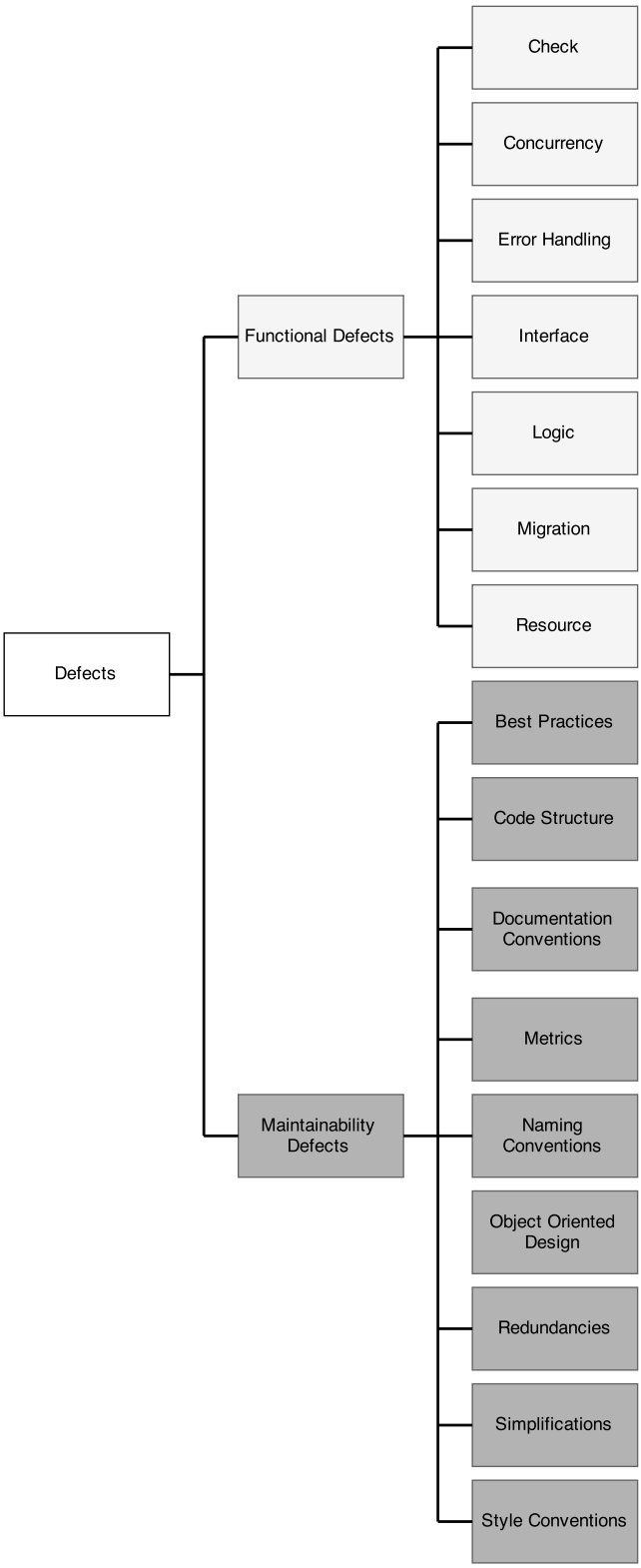

Figure 4.2: Our refined defect classification.

The maintainability defect classification resulted from the open card sort as described in Section 4.3. The most important difference with the classification of Mäntylä and Lassenius [51] concerns their *Structure* category. That category consisted of defects that concerned the compiled source code. In their study, 55% of all maintainability defects belonged to this category [51]. In contrast, categories in our classification are not defined by whether they concern the compiled source code or not. Rather, they are defined by what a rule checks for. The only exceptions are the *Best Practices* and *Style Convention* categories. These categories both contain rules that check for code conventions, but the former concerns those that affect the compiled output of the code, while the latter concerns those that do not. The *Style Convention* category therefore closely mirrors the *Visual Representation* category of Mäntylä and Lassenius [51]. For our classification, defects in the *Code Structure* category are those that are related to the structure of the file system and the coupling of parts of the system. Many tools also compute *Metrics*, which can emit a warning when a threshold is crossed. *Object Oriented Design* defects, as the name suggests, are those that check for a violation of object oriented design principles. A *Redundancy* warning points out duplicate code, or a piece of code that is otherwise unnecessary. *Simplification* warnings suggest where a piece of code might be simplified to improve readability. Finally, our classification groups those rules that check for *Documentation Conventions* and *Naming Conventions*.

Rules Used or Avoided

RQ2.1 concerns the warning rules that developers enable in their configurations. As we discussed in Section 4.3, we normalized the distribution of the enabled warning rules according to the number of rules in a category. Figure 4.3 shows these results for every tool. The underlying data for this figure, and all others in this section, are presented in Appendix B. Moreover, Figure B.2 in Appendix B shows the results from which we performed the normalization.

RQ2.2 involves the rules that developers disable in their configuration files. The normalized distribution of disabled rules are presented in Figure 4.4. Figure B.3 in Appendix B shows the results from which we performed the normalization.

In Figures 4.3 and 4.4, every color represents an ASAT. Every bar then displays the percentage of normalized rules that belong to a specific category in our classification. As an example, from Figure 4.3 we see that almost 10% of the normalized rules that are enabled in FindBugs configurations belong to the *Check* category. The figures allow us to easily spot those 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. Moreover, we observe in both figures that some tools have categories with no enabled or disabled rules.

Figures 4.3 and 4.4 present a distribution for each tool. To abstract over the tools, we computed, from this data, the median value for each category. We preferred this over the mean because of the outliers that are present for specific tools. These results are shown in Figure 4.5 for the rules that developers enable. Figure 4.6 shows these results for the rules that developers disable. Thus, the summation of the bars in these figures does not equal 100%. Rather, each bar is an individual median of the data in Figures 4.3 and 4.4 respectively.

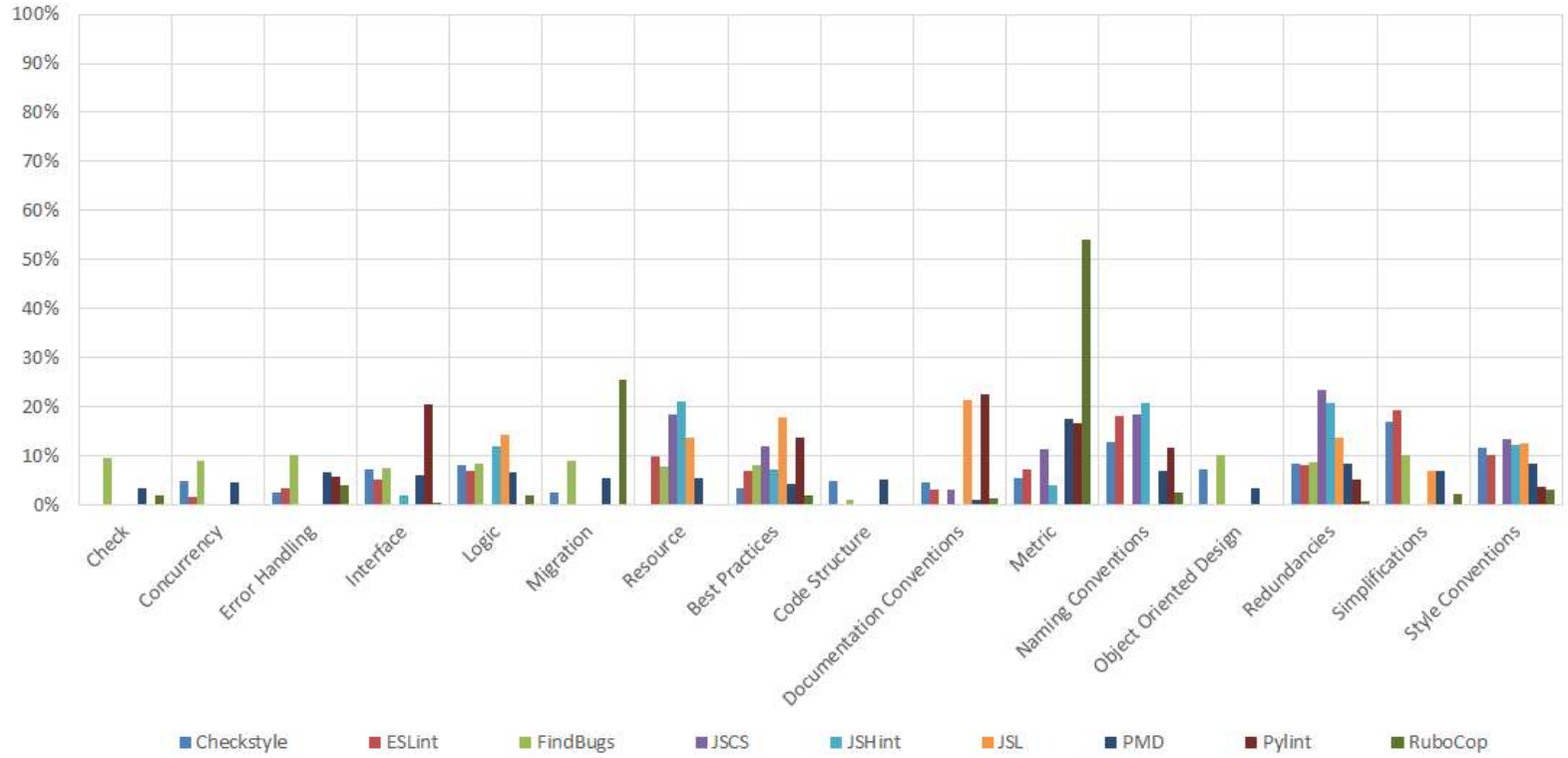

Figure 4.3: The distribution of rules that are enabled by developers per tool, according to our classification. Normalized to the number of rules in a category.

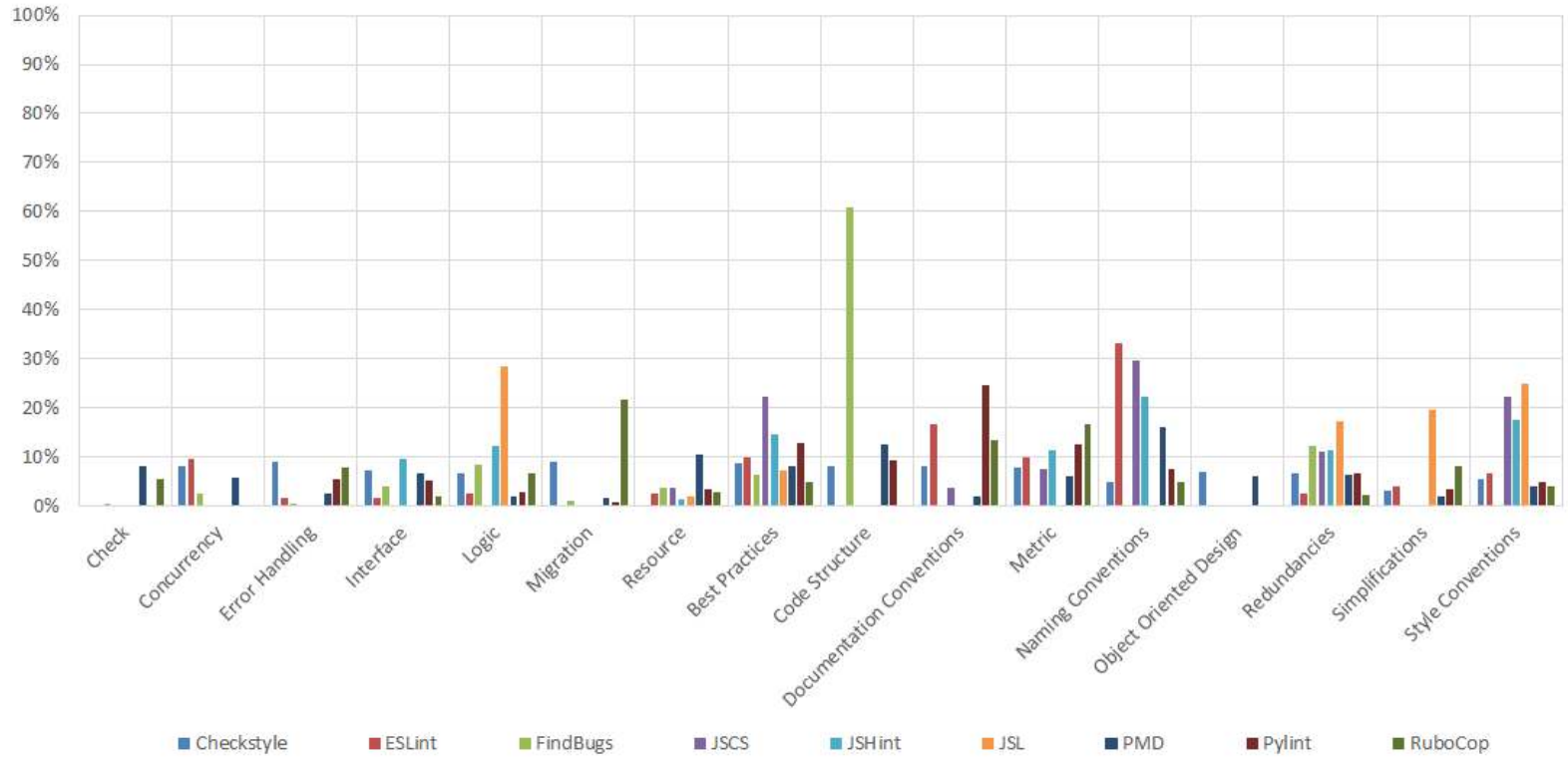

Figure 4.4: The distribution of rules that are disabled by developers per tool, according to our classification. Normalized to the number of rules in a category.

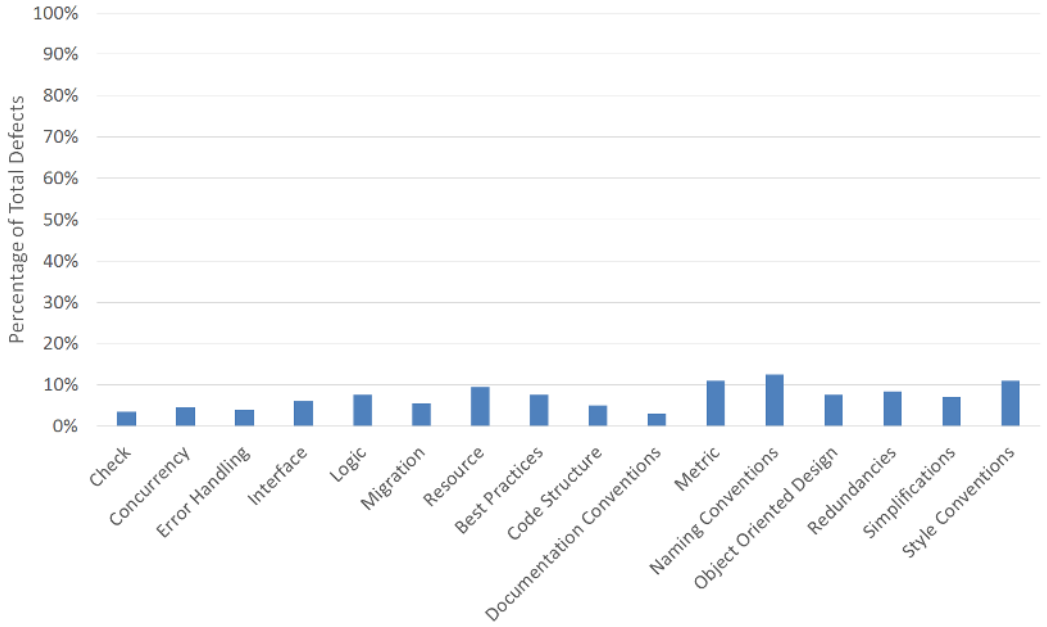

Figure 4.5: Median values for the distribution of rules that are enabled by developers, according to our classification. Normalized to the number of rules in a category.

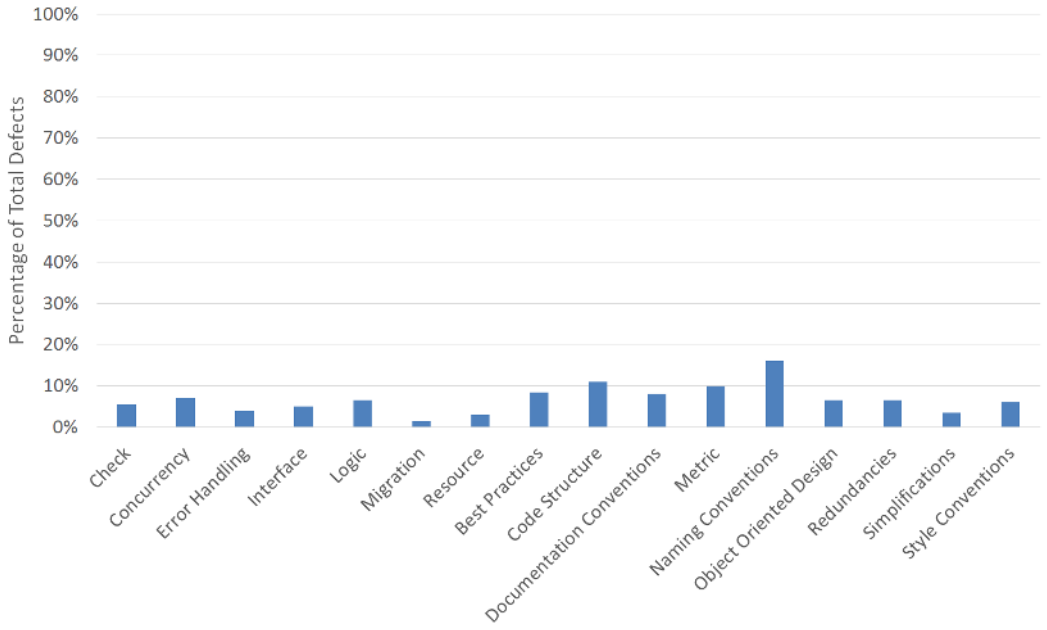

Figure 4.6: Median values for the distribution of rules that are disabled by developers, according to our classification. Normalized to the number of rules in a category.

Default Configurations

For **RQ2.3**, we calculated how many configurations deviated from the default. These results are displayed in Table 4.4. 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 how many files did not change a default rule, but possibly 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.

Tool	Changed Default Rules	Only Reconfigured Default Rules	No Deviations for Default Rules	Total %	Total
ESLint	80.5%	5.7%	13.8%	100%	4274
FindBugs	93.0%	—	7.0%	100%	2057
JSHint	89.6%	0.7%	9.7%	100%	104914
JSL	94.6%	—	5.4%	100%	848
Pylint	53.3%	—	46.7%	100%	3951
RuboCop	79.1%	3.2%	17.7%	100%	9579

Table 4.4: Summary of whether developer configurations deviate from the default configuration.

Furthermore, we 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. These results are shown in Table 4.5. Subsequently, for every rule that was turned off by default, we calculated in how many developer configurations that rule was enabled. The results for this analysis are shown in Table 4.6. Finally, for every rule that was enabled in the default configuration and which could be configured, we calculated how many configurations possibly reconfigured them. These results are shown in Table 4.7.

Tool	>25%	>50%	>75%	# of Default Rules
ESLint	3%	2%	0%	88
FindBugs	0%	0%	0%	121
JSHint	0%	0%	0%	16
JSL	0%	0%	0%	42
PMD	0%	0%	0%	297
Pylint	2%	0%	0%	195
RuboCop	0%	0%	0%	201

Table 4.5: The percentage of default rules that are disabled by developers in more than 25%, 50%, and 75% of all configuration files.

Tool	>25%	>50%	>75%	# of Default Rules
ESLint	14%	1%	0%	61
FindBugs	50%	25%	25%	8
JSHint	15%	10%	0%	52
JSL	0%	0%	0%	4
Pylint	0%	0%	0%	49
RuboCop	0%	0%	0%	8

Table 4.6: The percentage of by default disabled rules that are enabled by developers in more than 25%, 50%, and 75% of all configuration files.

Tool	>25%	>50%	>75%	# of Default Rules
ESLint	6%	2%	0%	88
JSHint	13%	6%	0%	16
RuboCop	1%	0%	0%	201

Table 4.7: The percentage of default rules that are configured by developers in more than 25%, 50%, and 75% of all configuration files.

The tables do not include Checkstyle because it does not provide a default configuration. Rather, it provides two configurations that reflect the conventions of Google[3] and Sun[4] (now Oracle, but the convention is still named after Sun). Similarly, JSCS does not provide a default configuration, but provides eight configurations used by companies such as Google and Airbnb. Checking whether their conventions are similar to those of open source developers is outside of the scope of our study. As described in Section 4.3, our aim is to see if ASAT creators create default configurations that reflect the wishes of their users. Table 4.4 does not include PMD because our analysis only considered explicit exclusions, but not implicit exclusions that can be used in PMD to exclude entire rule sets. Table 4.6 does not include PMD because the default configuration enables every rule. Finally, Table 4.7 only includes those tools from Table 4.5 that allow individual rules to be reconfigured.

Custom Rules

For **RQ2.4**, we calculated the percentage of custom rules in the configuration of developers. We defined what a custom rule is in Section 4.3. The results are shown in Table 4.8. We observe that 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.

[3] `https://google-styleguide.googlecode.com/svn/trunk/javaguide.html`
[4] `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`

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%

Table 4.8: Percentage of custom rules in the configuration files of developers. Grouped by tool.

4.6 Discussion

In this section, we discuss and interpret the results from the previous section and put them in a broader perspective. We present our main findings in boxes to highlight them. We connect our findings on how developers configure ASATs with previous results concerning the type of defects that ASATs find in practice.

For **RQ2.1**, Figure 4.3 shows that, on a high level, 65% of all enabled rules belong to a maintainability defect category. The other 35% of rules belong to a functional defect category. A reason for this might be that, in general, ASATs perform poorly at finding functional defects [2, 13, 78, 79]. Ayewah et al. [3] and Wagner et al. [78] hypothesize that the reason for this is 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 and discern that it is more worthwhile to manually try to find such defects, 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.

On a lower level, there are some outliers for individual tools. For instance, the *Metric* category for RuboCop and the *Logic* category for Pylint. These outliers indicate that, for a single tool, developers sometimes consider a specific category of defects as being less or more important than others. However, from Figure 4.5 we do not observe any defect category that stands out when comparing functional and maintainability defect categories among each other. There is some variation between the categories, but this is contained within a few percentage points.

Rules in some defect categories might be noticeably less or more enabled than others for individual tools, but not in general.

Concerning **RQ2.2**, from Figure 4.4 we observe that 75% of all the disabled rules are maintainability defects. This ratio is more in favor of maintainability defects than it is for the enabled rules. We used a two sample t-test to determine if this difference was significant. This proved not to be the case at $\alpha = 0.05$ ($t(15) = 0.488$, $p = 0.6607$). Thus, we can conclude that 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. Therefore, we see that even though the ability of ASATs to find functional defects is limited [2, 13, 78, 79], the rules that check for functional defects are not widely disabled. A potential reason for this might be that these rules do not emit a lot of false positives. As these are an important reason for why developers do not use ASATs [36], one can assume that a large number of false positives from a single rule would be cause for a developer to disable that rule. A rule that does not ever emit a warning might not be worth disabling, as it causes a developer no displeasure and might still find a real defect at a later time.

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

On a lower level, the most apparent outliers are the FindBugs *Code Structure* category and the RuboCop *Migration* category. However, from Figure 4.6 we again observe that there are no large differences between categories when we abstract the results over all tools. As is the case for the enabled rules, the differences are contained within a few percentage points.

Rules in some defect categories might be noticeably less or more disabled than others for individual tools, but not in general.

From the results in Section 4.5 regarding **RQ2.3**, Table 4.4 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.

The results from Tables 4.5 to 4.7 indicate that there are few rules that a noticeable percentage of all developers change or reconfigure. 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 those 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 of these rules.

Developers only widely disagree with a few rules in default configurations.

Finally, regarding **RQ2.4**, the results show that custom rules do not comprise a sizable segment of all rules that are used by developers, amounting to less than 5% for all tools in this study. This can indicate that developers do not consider the ASATs in this study to be incomplete, in the sense that they create a considerable number of custom rules to check for those defects that are not included in the built-in rule set of an ASAT. 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.

4.7 Threats to Validity

In this section, we review the threats that could impact the validity of this study on how developers configure ASATs. For every threat, we further discuss how we endeavored to mitigate that threat. First, we discuss the internal threats, then the threats to the construct validity, and finally we review the external threats to validity.

Internal Validity

Internal threats are those factors that affect the validity of our analysis on the study objects and the conclusions that we draw from our measurements. There are two internal threats that we identified and that we sought to mitigate.

- A heavy dependency on GitHub to provide the configuration files for our analysis, as is evident from Table 4.3, might produce a bias in our results. However, we tried to mitigate this as best we could. We looked for all code hosting services and search engines that allowed us to find code based on either a filename or a specific XML tag. We only found three sources: GitHub, OpenHub, and Krugle. We attempted to collect every configuration file from these sources. OpenHub and Krugle simply could not provide us with more configuration files. In fact, as discussed in Section 4.4,

there were more files on GitHub than we were able to retrieve. This left only two options: perform the analysis with as much data as possible, or take a random sample of GitHub files equal to the number of files from the other sources. However, Table 4.3 shows that the second option would further limit the number of ASATs that we could study. For instance, the other two sources only provided 8 unique configuration files for ESLint.

- There might be errors in our measurements due to the use of our analysis tool. However, we tried to mitigate this through the use of automated tests. As such, we could verify that the tool worked as expected on small sample sizes. Moreover, we programmed our tool defensively, that is, the tool skips those configuration files that do not conform to a strict specification of how a configuration file should be formatted. However, there is always the chance that our tests and verifications missed something and that this error influenced our results.

Construct Validity

Threats to the construct validity of the study are those that have an impact on if our constructs accurately measure what they are supposed to measure. We identified one threat to the construct validity of this study.

For our analysis, we performed a classification of ASAT warning rules according to a partially custom classification. This is subjective because an individual performs this classification. We sought to mitigate this subjectivity in two ways. First, for the functional defects, we used the classification of Mäntylä and Lassenius [51], who in turn based their classification on that of El Emam and Wieczorek [20]. Both of these classifications proved to be reliable and distinguishable [20, 51]. The adaptations we made were out of necessity. As described in Section 4.5, we excluded two categories because there were little to no defects in those categories. Abstracting over a very small set of rules would be pointless as one poorly performing rule would reflect heavily on the entire category, even though it might not be representative. Finally, we only classified a rule as belonging to the new *Migration* category if the documentation of that rule explicitly stated that it checked for distinct behavior between language versions, removing subjectivity from that categorization as much as possible. For the maintainability defects, we used a open card sort with two participants on all the warnings rules of one tool to create reliable and distinguishable categories. This technique is widely used in information architecture for this purpose [5].

External Validity

External threats are those that affect the generalizability of our results and conclusions. We identified two external threats to this study.

- This study only considers the configurations of ASATs from open source projects. As such, its generalizability towards closed source projects might be limited. Research has shown that core developers of open source projects have to keep the interests of their non-core contributors in mind when they decide how strict their quality assurance practices should be [28]. Therefore, we cannot readily assume that these results

for open source projects are accurate indicators of the use of ASATs in closed source projects.

- We performed this study for nine specific ASATs in four different programming languages. As can be seen from Section 4.4, the nine selected ASATs represent a diverse set of tools. Their first releases are spread out, they have a varying number of rules, and different focuses regarding which defects they want to find. 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.

Chapter 5

Evolution of Automated Static Analysis Tool Configurations

Having attained an understanding of how developers configure the ASATs that they use, in this chapter we examine how these configurations change over time. First, we discuss the research aim of the study and the high level study design. Afterwards, we outline the steps we took to perform this design. Subsequently, we examine the study objects that we use in this chapter. Then, we report on the results of the study and offer our interpretation in the discussion. Finally, we discuss the internal and external threats to the validity of the study and its results.

5.1 Research Aim

The research aim of this chapter directly follows from the research questions of Chapter 4. There, we studied how developers configure ASATs, motivated by a lack of knowledge on how developers use ASATs. That analysis only focused on the present status of ASAT configurations. In this chapter, we examine if and how these configurations change over time. To further study the evolution of ASAT use, we examine if changes to ASAT configurations are related to updates to an ASAT. Finally, we study what versions of ASATs developers use. More concretely, we strive to answer these research questions:

RQ3: How does the use of ASATs evolve?

- **RQ3.1:** How often does a configuration file change?
- **RQ3.2:** How much does a configuration file change?
- **RQ3.3:** When does a configuration file change?
- **RQ3.4:** Do updates to an ASAT trigger developers to change their configuration file?
- **RQ3.5:** What versions of ASATs are currently being used?

5.2 Study Design

In this section, we detail the design of the study from the initial exploratory investigation, to the data collection, and the analysis of that data.

The high level study design for **RQ3.1** to **RQ3.4** is shown in Figure 5.1. To perform our analysis, we develop a program that retrieves all the information of the changes to a particular configuration file and accumulates this information over all the files. In step 3, we have obtained several distributions showing if and how configuration files change over time. We use this information to answer **RQ3.1** to **RQ3.3**. Additionally, we collect the release dates for every version of an ASAT. We then connect a change of a configuration file with a particular ASAT version. We use this information to answer **RQ3.4**.

The study design for **RQ3.5** is shown in Figure 5.2. In step 1, we develop a program that parses the dependencies of a project, as listed in a dependency file, and retrieves the version information of the ASATs under study. We then use these results to answer the research question.

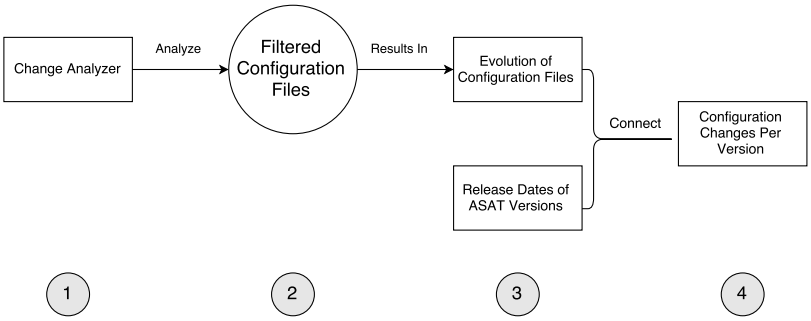

Figure 5.1: The study design of RQ3.1 to RQ3.4.

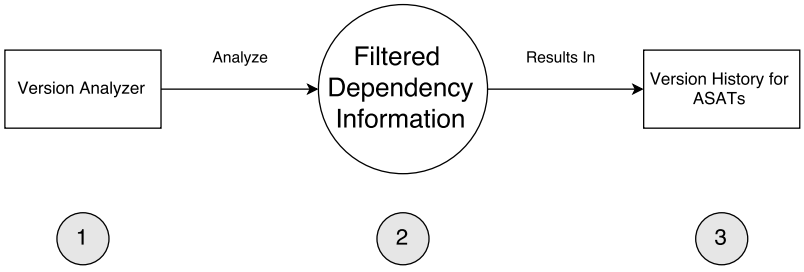

Figure 5.2: The study design of RQ3.5.

5.3 Study Methodology

In this section, we discuss how we executed the study design. First, we detail how we obtained the information about the changes to configuration files. Then, we discuss how we used this data to study how configuration files change over time. Finally, we discuss how we obtained the dependency information for projects and how we used this to examine which versions of ASATs projects use.

To investigate the evolution of ASAT configuration files, we were interested in the information of the commits that changed a configuration file. We used the GitHub API to retrieve this information. This allowed us to request the history of a file in a repository. A consequence of this reliance on the GitHub API is that we could not reuse the configuration files from OpenHub and Krugle, as there was no way for us to obtain information about the history of a file from these sources.

We computed several metrics from this data. The first metric, for **RQ3.1**, is simply how often a file was changed. This can show us if developers have a need to adapt their configuration, either because the ASAT was updated or because of changing needs among developers. Furthermore, for **RQ3.2**, 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 probably means that there are only lines modified, which count as both an addition and a deletion in the information of a change. We could not compute other measures, such as the Levenshtein distance, because we did not examine the contents of a change. The last metric we computed, for **RQ3.3**, was the difference between the creation date of the file and the date of a particular change. This can be an indication of whether ASAT configurations are continuously changed or if they are created, possibly corrected once or twice, and then never changed again.

For **RQ3.4**, we calculated, for each version of each tool, how many configuration files were changed when a certain version was the latest one available. This can show us if developers tend to change their configuration when a tool is updated, or whether these updates seem to be unrelated to ASAT updates. To eliminate the variances in update frequency, we divided this number by the number of days that version was the latest version. However, an increased or decreased general use of ASATs over time could cloud the results, as this would almost certainly influence the total number of changes to configuration files. To negate this effect, we divided the results once more, this time by the number of created configuration files. To summarize, we computed the number of changes per day, per created configuration file, in the timespan that a version was the latest available version.

For **RQ3.5**, we examined the versions of ASATs that are currently in use. Often, an ASAT does not include version information in a configuration file. Therefore, to retrieve the version numbers of the ASATs used, we needed a list of dependencies. We collected this information from package management systems that handle dependencies for projects. Via these systems, users can include the ASATs that they want to use and specify the version number that they require. We then extract this information to answer the research question.

5.4 Study Objects

In this section, we discuss the study objects that we used to asses the evolution of ASAT use in open source projects.

For **RQ3.1** to **RQ3.4**, we needed configuration files to study. Because we already collected as many configuration files as we could for the study of the current use of ASATs in Chapter 4, we simply reused those configuration files. As we described in Section 5.3, we relied on the GitHub API and could therefore only use the configuration files from GitHub. Table 5.1 recaps the number of configuration files we used for the analysis, grouped by tool.

Tool	GitHub
Checkstyle	16271
FindBugs	1575
PMD	5562
ESLint	4427
JSCS	11656
JSHint	105619
JSL	862
Pylint	3941
RuboCop	10063
Total	159976

Table 5.1: Recap of the number of configuration files for each ASAT

However, not all of these ASATs could be used as study objects for **RQ3.4**. The reason for this is that not all tools are directly included as a dependency. Rather, developers can often include an ASAT via another artifact which contains it. For example, we observed that many developers use PMD via the SonarQube[1] plugin. These different artifacts can have entirely different versioning schemes and often do not directly reveal which ASAT version they use underneath. Thus, we limited our analysis to those tools that, for the most part, are directly included as a dependency. The tools that fit these requirements were *ESLint* and *Pylint*.

For **RQ3.5**, we needed dependency files from package management systems to study. For the same reason as described for RQ3.4, we exclude those ASATs that are generally included via different artifacts than by way of the main release. However, we were also unable to use Pylint as study object, as the oft-used *pip* package manager[2] for Python does not produce dependency files. This left only *ESLint*, for which we analyzed 10419 dependency files from the Node.js runtime environment, which includes a package manager.

[1] `http://www.sonarqube.org/`
[2] `https://pip.pypa.io/en/stable/`

5.5 Results

In this section, we discuss the results of analyzing how the use of ASAT evolves. We study how configuration files evolve after their initial creation, if they evolve at all. To do this, we first investigate how often ASAT configuration files are changed. Then, we examine the size of the change and when the change occurred. Subsequently, we investigate if an ASAT update triggers developers to change their configurations. Finally, we outline which versions of ASATs are currently in use.

Number of Changes

The results for **RQ3.1**, regarding how often a configuration file changes, are shown in Figure 5.3. The underlying results for this figure, and all others in this section, are presented in Appendix C. We see that a little over 80% of all configuration files are never changed after their initial creation. The range shown in the chart represents 99.5% of the total data. The median of the data is 0 and the mean is 0.5. Both the first and third quartile are 0. 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.

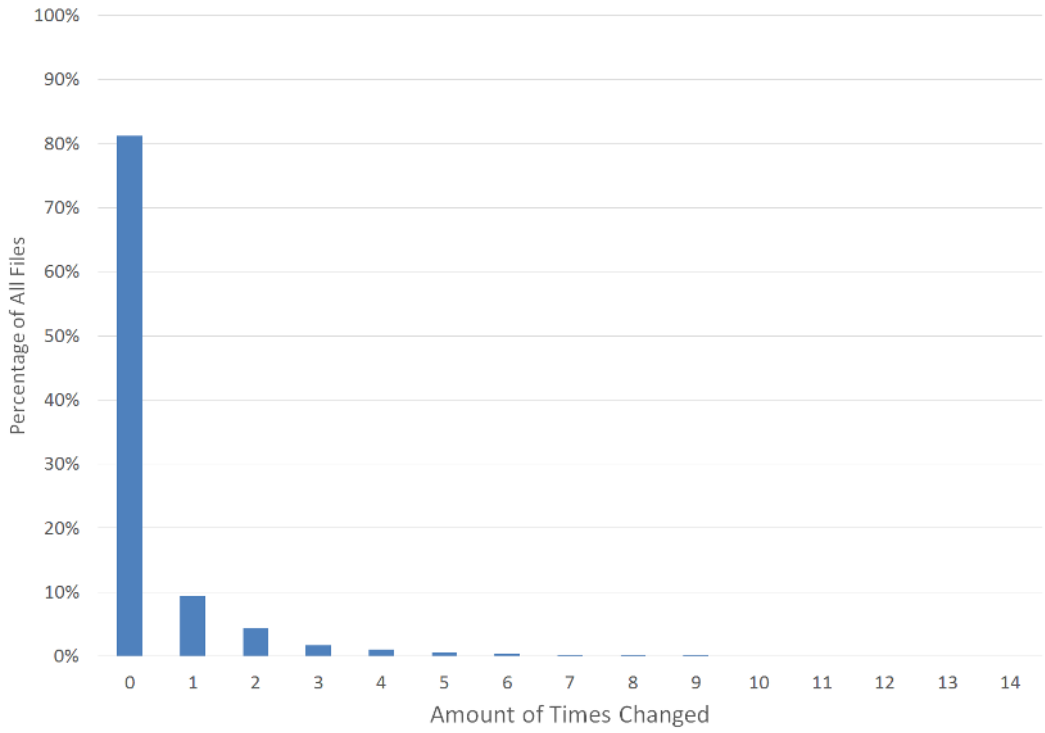

Figure 5.3: Distribution of the number of times a configuration file is changed.

Size of a Change

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. As discussed in Section 5.3, we define this as the *number of additions in a file - number of deletions in a file*. The results, for **RQ3.2**, are shown in Figure 5.4. We observe that the total number of changes is zero for more than 25% of all files. This means that either all lines that were changed were only modified (which counts as an addition and a deletion), or that there were as many lines added as there were deleted. Furthermore, we observe that there is a greater chance that a change has more additions than deletions. The range shown 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 median of the total data is 1, the mean is 1.64, the first quartile is 0, and the third quartile is 2.

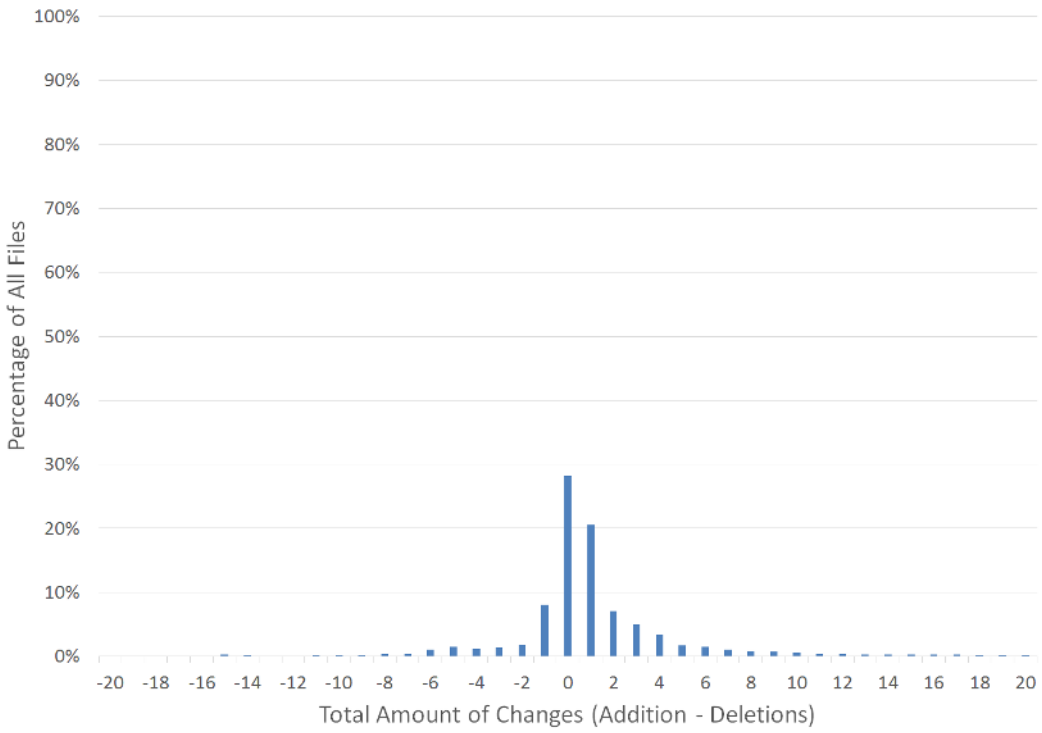

Figure 5.4: Distribution of the size of a change, defined as $Lines_{ADDED} - Lines_{DELETED}$.

Change Interval

For **RQ3.3**, we computed, for each change of every file, the time between the creation of a file and when the change was made. The results are shown in Figure 5.5. 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 median of the total data is 32, the mean is 150.67, the first quartile is 3, and the third quartile is 163. The maximum is 4251 days, more than 11.5 years, for a Checkstyle configuration.

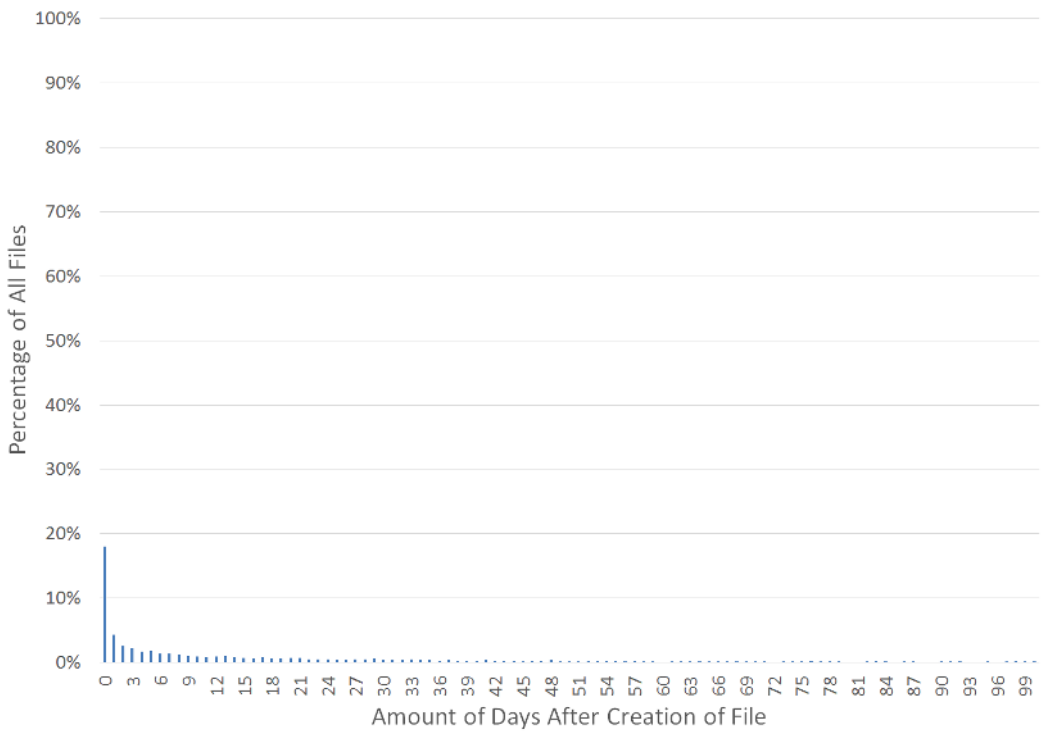

Figure 5.5: Distribution of the time between the creation of a file and the time when the file was changed.

Change Correlation with Version Updates

The results for **RQ3.4**, whether a change in an ASAT configuration might be related to an update of the ASAT, are shown in Figures 5.6 and 5.7 for ESLint and Pylint respectively. For every version, the bar represents the number of changes per day, per created configuration file, in the timespan that a version was the latest available version. Generally, we observe that the values vary between zero and four changes.

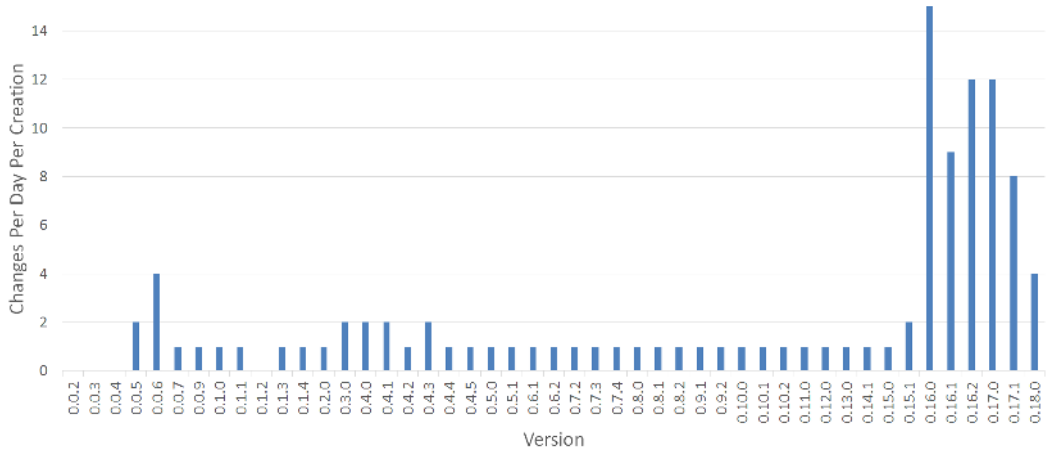

Figure 5.6: The number of changes per day, per creation, for each version of ESLint.

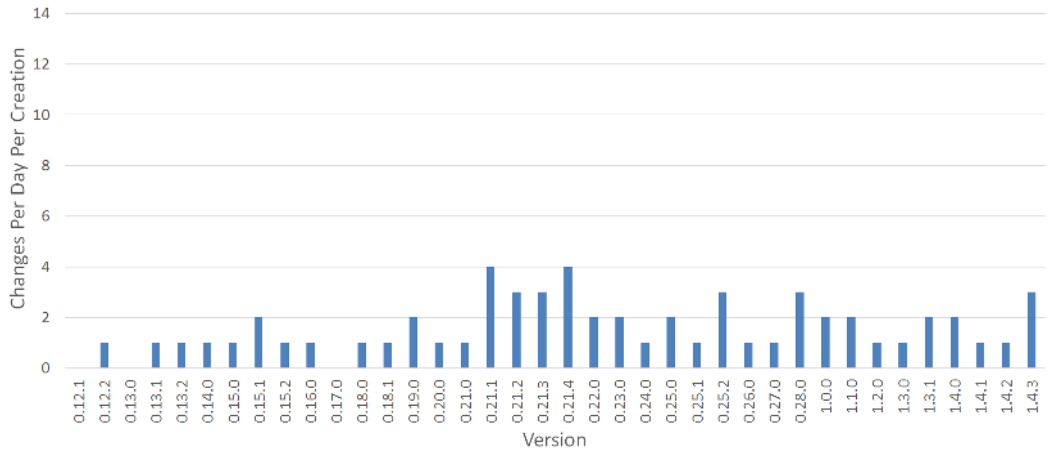

Figure 5.7: The number of changes per day, per creation, for each version of Pylint.

Versions Used

The results for **RQ3.5**, which versions of ESLint are currently in use among open source developers, are shown in Table 5.2. We observe that about 2.5% of all users choose to always have the latest version of ESLint. Nearly 50% of users indicate that they want to use the most recent minor update to the *0* release. Currently, this has the same effect as requesting the latest version of the tool. However, when the next major version of ESLint is released, only the users who indicated that they want the latest version will automatically get upgraded to that new release. Otherwise, we see that only the *0.6* and *0.4* releases are used by more than 5% of all users.

Version	Date	Times Used
Latest	2015/04/24	255
0	2015/04/24	5098
0.20	2015/04/24	75
0.19	2015/04/11	250
0.18	2014/03/28	262
0.17	2015/03/18	175
0.16	2015/03/11	76
0.15	2015/02/26	145
0.14	2015/02/08	87
0.13	2015/01/24	185
0.12	2015/01/17	172
0.11	2014/12/30	92
0.10	2014/12/13	150
0.9	2014/11/01	231
0.8	2014/09/20	171
0.7	2014/07/10	120
0.6	2014/05/23	1270
0.5	2014/04/17	141
0.4	2014/03/29	917
0.3	2014/01/21	58
0.2	2014/01/01	129
0.1	2013/12/06	354
0.0	2013/10/06	6
Total	—	10419

Table 5.2: Version use for ESLint.

5.6 Discussion

In this section, we discuss and interpret the results from the previous section. We review how the evolution of configuration files can impact the upgrades and fixes that ASAT developers provide with new versions of their software. We present our main findings in boxes to highlight them.

Regarding **RQ3.1**, 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 are never changed.

Looking at only the files that are changed, the results for **RQ3.2** 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 **RQ3.3** show that a configuration files is most likely to be changed on the same day it was created. Looking further one week, we see that slightly over a third of all changes were made in this timespan. 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 the changes to configuration files happen in the first week after the creation of a file.

From the results for **RQ3.4**, we observe that the number of changes per version mostly stays stable, varying no more than four changes per day. Interestingly, there is no spike in changes when the major *1.0* version of Pylint was released. We do see spikes among recent versions for ESLint. The release notes for these releases mention several breaking changes which might be the cause for these spikes. However, many previous versions also introduced breaking changes, so it is unlikely that the changes in recent versions were the reason for the spikes. A possible reason, as can be seen from the data in Appendix C, is the low number of created configuration files for those versions. It is possible that the low number of created files during that timespan is not reflective of the actual frequency of file creations, but rather a symptom of how frequently GitHub indexes files.

Changes to configuration files are most likely not related to ASAT version updates.

The results for **RQ3.5** show that about half of all developers who use ESLint choose a specific version of the tool, while the other half receives minor updates only. Only 2% of all developers choose to automatically receive major updates, which could force them to change their configuration files. Of course, automatically upgrading to a new major version might not be desirable, as major versions can introduce breaking changes. The other 98% can either stay on one version, or receive minor upgrades without being forced to change their configuration.

Half of the users of ESLint want minor version updates, while the other half requires a specific version.

These results show that the creators of ASATs should keep in mind that most of their users will either not upgrade to a new version of the ASAT, or they upgrade without changing their configuration. Assuming that a new version adds, removes, or changes some rules, it would stand to reason that developers who upgrade would inspect these changes and change their configuration accordingly. Yet, this will probably not happen. Thus, an ASAT update which changes the semantics of what a rule checks for will have unintended side effects. Developers that never change their configuration when updating will, unknowingly, check for different defects when they update. Furthermore, those rules that are added will remain unused by the majority of the existing user base.

ASAT developers should make sure that rule changes never alter the semantics of what that rule checks for.

5.7 Threats to Validity

In this section, we review the threats that could impact the validity of this study on the evolution of ASAT use. Furthermore, we discuss how we endeavored to mitigate the threats. First, we review the internal threats, and then we outline the external threats to validity.

Internal Validity

Internal threats are those factors that affect the validity of our analysis on the study objects and the conclusions that we draw from our measurements. There are two internal threats that we identified and that we sought to mitigate.

- Considering that most repositories on GitHub are inactive [27], it stands to reason that most of the collected configuration files belong to inactive projects. This might produce a bias in our results. Gousios et al. [27] observed that 32% of all GitHub

projects have only been active one day. If those projects took the time to initialize their ASAT configurations, which seems unlikely, then, obviously, these configurations would never change. However, we argue that the results as presented in this study accurately reflect the totality of open source ASAT configurations on GitHub, whether those projects are active or inactive, new or mature, or with many or few collaborators. Moreover, we report the results as they are. We do not try to infer reasons for why developers do or do not change their configuration files.
- Our analysis tool might have introduced errors in our measurements. However, we tried to mitigate this by manually verifying that the tool produced the expected results when used on a small amount of data. As such, we expect the tool to work properly on large amounts of data as well. However, there is always a chance that we missed something while verifying the workings of the tool and that this error impacted our results.

External Validity

External threats are those that affect the generalizability of our results and conclusions. We identified three external threats to this study.

- Because of our reliance on the GitHub API, the results in this chapter only represent the evolution of ASAT use for GitHub projects. However, given the relative popularity of GitHub to other code hosting services [27], the migration of many to projects to GitHub [27], and because of the large number of projects that we collected configuration files for, we expect the results to further generalize to projects on other code hosting services.
- This study only considers the evolution of ASAT use in open source projects. As such, its generalizability towards closed source projects might be limited. Research has shown that core developers of open source projects have to keep the interests of their non-core contributors in mind when they decide what their quality assurance practices should be [28]. This might also be reflected in the evolution of those quality assurance practices. Therefore, we cannot readily assume that these results for open source projects are accurate indicators of how the use of ASATs in closed source projects evolves.
- The results for **RQ3.1** to **RQ3.3** were obtained from the configuration files of nine ASATs from four different programming langauges, as outlined in Section 5.4. Because they represent a diverse set of tools, we expect the results to further generalize to other ASATs. However, we performed the study of **RQ3.4** for two tools and that of **RQ3.5** for one tool. From this small sample size, we cannot readily conclude that these results will apply to other tools as well.

Chapter 6

Conclusion

In this thesis, we have conducted a large scale evaluation of the state of static analysis in open source projects. We focused on those areas where the current knowledge about the state of static analysis is missing or incomplete. We set out to answer three sets of research questions, the first of which was discussed in Chapter 3:

RQ1: How common are static analysis techniques in practice?

We answered this question by means of a repository analysis complemented with a questionnaire. We found that code review is common among open source projects, but not universally used among core developers. Many projects still only review code of non-core contributors, or review changes by core developers only occasionally. Similarly, we observed that while ASATs are not ubiquitous among open source projects, they are used in various levels of strictness by the majority of the projects. This contradicted with some claims in the literature that ASATs have not achieved significant adoption among developers. Finally, we observed that, by and large, developers do not use multiple ASATs.

In Chapter 4, we performed an in-depth quantitative study on the use of ASATs. More concretely, we strove to answer the following research question:

RQ2: How are ASATs used?

To answer this question, we analyzed ASAT configuration files of developers by means of a defect classification. We observed that developers both use and avoid maintainability defects to a greater extent than functional defects. However, there was no great difference between the defect categories on a lower level. Significant outliers were only found on the level of individual tools, showing that some tools might perform better or worse in specific areas, confirming the results found in other studies. Furthermore, we found that most configuration files deviate from the default configuration. However, there are hardly any individual rules in the default configurations that developers widely disagree with. Our results indicate that ASAT developers would only need to critically evaluate a small number of rules in their default configurations to better fit the wishes of their users. Finally, we found that developers seem to be satisfied with the completeness of the ASATs, as custom rules only comprised a small portion of all rules in the configurations of developers.

After studying the use of ASATs, we examined how this use evolved in Chapter 5. The research question was the following:

RQ3: How does the use of ASATs evolve?

We found that most configuration files, representing the use of ASATs by developers, never change. And if they do change, the total number of changed lines is small. The changes are spread out over the lifetime of the project, with a tendency for changes to happen within the first week after the creation of a configuration file. Moreover, we observed that it is unlikely that the changes to configuration files are related to the version updates of ASATs. Finally, regarding the ASAT versions that are used, an analysis for ESLint showed that the user base is split almost evenly between those that prefer the latest major version and those that want to stay at a specific minor version.

In conclusion, we have seen that popular open source projects commonly use code reviews and ASATs to verify the quality of their code. However, code review policies for core developers are more lenient than those for non-core contributors. A typical project will use one ASAT, that is configured once. The configuration of the ASAT deviates from the default configuration, does not contain custom rules, and both enables and disables maintainability defects to a greater extent than functional defects. Developers will only run the ASAT occasionally and without attaching any strict consequences to its results.

Future Work

In this study, we examined the prevalence of code reviews and ASATs in open source projects. While current research has presented reasons why developers may or may not use these quality assurance practices, as discussed in Chapter 2, the results still bring up several unanswered questions.

First, we saw that some projects choose to use code reviews for changes of core developers, but do not make them mandatory. Future research could investigate why these projects employ this lenient form of code review, rather than, as many other projects do, provide developers with exceptions for small changes and typos. Arguably most importantly, if code review is only encouraged but not enforced, how does an author decide which changes to submit for review and which changes to immediately commit to the repository?

For ASATs, we observed that many projects did use ASATs, but not in a strict and continuous manner. Previous research indicated that an overload of warnings and poor integration into the development processes were the primary reasons for not using ASATs [36]. However, an overload of warnings seems to be related only to the decision of whether to use ASATs or not. It cannot be a reason for using ASATs sporadically, because the warning overload will be much smaller when the tool is used more frequently. Therefore, future work could take a more in-depth look at what stops many projects from integrating ASATs into their development workflow in a manner that forces a continuous use of ASATs. Additionally, future work could perform a more in-depth investigation of how frequently developers run ASATs and where they run them (via an IDE, a continuous integration tool, or somewhere else).

Regarding the use of ASATs, our results showed which types of defects developers disable, but our quantitative analysis could not allow us to know why those rules are disabled. When enabling rules, one could reasonably assume that developers consider them to be important. But regarding the disabling of a rule, we currently do not know what developers consider to be decisive. If a rule clashes with the conventions in the projects, it makes sense for the developers to disable that rule. But otherwise, it is unclear whether a developer is more likely to disable a rule because of an overload of false positives, an inability to find real defects, or maybe because it is unclear what a rule checks for and why it is important. Thus, future work could examine the reasons why developers disable rules.

Furthermore, we observed that most developers make changes to a default configuration, but that few rules are changed by a significant percentage of all developers. However, this does not always mean that they are satisfied with this configuration. A recent discussion in a pull request on the ESLint GitHub repository,[1] indicated that many users would prefer a default configuration that did not enable any rules. Consequently, ESLint will switch to this behavior in the *1.0* release [80]. New users will be provided a way to generate a configuration based on rules recommended by the ESLint team. We do not know why users prefer this type of default configuration. Future work could investigate what the optimal type of default configuration is and why that is the case.

Our study indicated that developers generally do not create custom rules for ASATs. Future work could investigate why they do not do so. Are developers satisfied with the completeness of ASATs? Or do they check for defects manually or with another ASAT rather than create a custom rule? For developers that do create their own rules, future work could investigate if these rules check for highly specific defects, or if these developers implement rules that could be also be applied to other projects.

Regarding the evolution of ASAT configuration files, we observed that most configurations did not change. Research could examine why this is the case. Is it because developers are satisfied with the performance of the ASAT or because they do not believe that a change of configuration will have a noticeable positive impact? And when a developer does make a change, is that motivated by a poorly performing rule or maybe because of changing conventions in the project?

Similarly, future work could investigate what would prompt developers to upgrade their ASAT. Does this differ for major and minor versions? Potentially, it is closely related to being forced to make changes to a configuration file, as might be the case for a major release. And if developers upgrade to a version that added warning rules, do they actually make use of these new rules or do they ignore them?

[1] `https://github.com/eslint/eslint/issues/2100`

Key Contributions

We made the following contributions in this thesis:

- **Performed a study on the prevalence of static analysis techniques:** By means of a repository analysis and a questionnaire, we provided hard data on how prevalent code reviews and ASATs are in open source projects. We showed that both techniques are common, but not ubiquitous.
- **Built a classification schema for ASAT warnings:** We created a defect classification, with a fine-grained classification for maintainability warnings, to better capture what ASAT rules check for. This classification can be used to understand both the use of ASATs, which we did in this thesis, and their performance.
- **Examined the use of ASATs among open source developers:** By studying the configuration files of ASATs, we determined that developers both use and avoid maintainability defects to a greater degree than functional defects. Moreover, there can be large differences between how developers use ASATs on the level of individual tools.
- **Studied the accuracy of default ASAT configurations:** Through comparing the configurations of developers with the default configurations of ASATs, we established that most configurations deviate in some way from the default configuration. However, on the level of individual rules, developers only widely disagree with a few rules.
- **Examined the evolution of ASAT use:** By analyzing the changes to configuration files of ASATs, we ascertained that most configuration files do not change. And if they do, they change gradually and over the lifetime of a project. Moreover, changes in configurations do not seem to be related to ASAT version updates.

Bibliography

[1] John G Adair. The hawthorne effect: A reconsideration of the methodological artifact. *Journal of applied psychology*, 69(2):334, 1984.

[2] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252. ACM, 2010.

[3] Nathaniel Ayewah, William Pugh, David Morgenthaler, John Penix, and YuQian 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*, pages 1–8. ACM, 2007.

[4] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.

[5] Iain Barker. What is information architecture, 2005. URL `http://www.steptwo.com.au/papers/kmc_whatisinfoarch/`. Accessed on: June 18th, 2015.

[6] Bozhidar Batsov. Rubocop — a ruby static code analyzer, 2015. URL `http://batsov.com/rubocop/`. Accessed on: May 7th, 2015.

[7] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael Godfrey. The influence of non-technical factors on code review. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 122–131. IEEE, 2013.

[8] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 202–211. ACM, 2014.

[9] Mario Bernhart, Andreas Mauczka, and Thomas Grechenig. Adopting code reviews for agile software development. In *Agile Conference (AGILE), 2010*, pages 44–47. IEEE, 2010.

[10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2): 66–75, 2010.

[11] Oliver Burn. checkstyle - checkstyle 5.9-snapshot, 2014. URL `http://checkstyle.sourceforge.net/`. Accessed on: October 14, 2014.

[12] Ram Chillarege, Inderpal Bhandari, Jarir Chaar, Michael Halliday, Diane Moebus, Bonnie Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *Software Engineering, IEEE Transactions on*, 18(11): 943–956, 1992.

[13] Cesar Couto, João Montandon, Christofer Silva, and Marco Tulio Valente. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, 21(2):241–257, 2013.

[14] Coverity Inc. Effective management of static analysis vulnerabilities and defects. White paper, Coverity Inc., 2009.

[15] Coverity Inc. Coverity scan - github integration, 2015. URL `https://scan.coverity.com/github`. Accessed on: May 21st, 2015.

[16] Coverity Inc. Coverity scan - travis ci integration, 2015. URL `https://scan.coverity.com/travis_ci`. Accessed on: May 21st, 2015.

[17] Douglas Crockford. Jslint, the javascript code quality tool, 2014. URL `http://www.jslint.com`. Accessed on: October 22, 2014.

[18] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, 2008.

[19] Stephen Eick, Clive Loader, David Long, Lawrence Votta, and Scott Vander Wiel. Estimating software fault content before coding. In *Proceedings of the 14th international conference on Software engineering*, pages 59–65. ACM, 1992.

[20] Khaled El Emam and Isabella Wieczorek. The repeatability of code defect classifications. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 322–333, 1998.

[21] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217(0):5–21, 2008.

[22] ESLint. Eslint - pluggable javascript linter, 2015. URL `http://eslint.org/`. Accessed on: May 7th, 2015.

[23] Michael Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[24] FindBugs. Findbugs - find bugs in java programs, 2014. URL `http://findbugs.sourceforge.net/`. Accessed on: October 2nd, 2014.

[25] Adrian Furnham. Response bias, social desirability and dissimulation. *Personality and Individual Differences*, 7(3):385 – 400, 1986.

[26] GitHub. Press, 2015. URL `https://github.com/about/press`. Accessed on: May 22st, 2015.

[27] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 384–387. ACM, 2014.

[28] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrators perspective. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE 2015, pages 358–368. IEEE, 2015.

[29] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. Communication in open source software development mailing lists. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 277–286. IEEE Press, 2013.

[30] Sarah Heckman. Adaptively ranking alerts generated from automated static analysis. *Crossroads*, 14(1):1–11, 2007.

[31] Sarah Heckman and Laurie 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*, pages 41–50. ACM, 2008.

[32] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, 2011.

[33] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12): 92–106, 2004.

[34] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

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

[36] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

[37] Philip Johnson. Reengineering inspection. *Commun. ACM*, 41(2):49–52, 1998.

[38] Philip Johnson and Danu Tjahjono. Does every inspection really need a meeting? *Empirical Software Engineering*, 3(1):9–35, 1998.

[39] JSCS. Jscs - about, 2015. URL `http://jscs.info/`. Accessed on: May 7th, 2015.

[40] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101. ACM, 2014.

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

[42] Michael Kaplowitz, Timothy Hadlock, and Ralph Levine. A comparison of web and mail survey response rates. *Public opinion quarterly*, 68(1):94–101, 2004.

[43] Sunghun Kim and Michael 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*, pages 45–54. ACM, 2007.

[44] Anton Kovalyov. Jshint, a javascript code quality tool, 2015. URL `http://jshint.com/`. Accessed on: May 7th, 2015.

[45] Anton Kovalyov. Jshint option reference, 2015. URL `http://jshint.com/docs/options/`. Accessed on: May 8th, 2015.

[46] Ted Kremenek and Dawson Engler. *Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations*, volume 2694 of *Lecture Notes in Computer Science*, book section 16, pages 295–315. Springer Berlin Heidelberg, 2003.

[47] Melina Kulenovic and Dzenana Donko. A survey of static code analysis methods for security vulnerabilities detection. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*, pages 1381–1386, 2014.

[48] Rahul Kumar and Aditya Nori. The economics of static analysis tools. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 707–710. ACM, 2013.

[49] Logilab. Pylint - code analysis for python — www.pylint.org, 2015. URL `http://pylint.org/`. Accessed on: May 7th, 2015.

[50] Brad Long, Paul Strooper, and Luke Wildman. A method for verifying concurrent java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 19(3):281–294, 2007.

[51] Mika Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *Software Engineering, IEEE Transactions on*, 35(3):430–448, 2009.

[52] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed Hassan. The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.

[53] Matthias Miller. Javascript lint, 2015. URL `http://www.javascriptlint.com/`. Accessed on: May 7th, 2015.

[54] Nachiappan Nagappan, Laurie Williams, John Hudepohl, Will Snipes, and Mladen Vouk. Preliminary results on using static analysis tools for software inspection. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 429–439, 2004.

[55] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph Johnson, and Danny Dig. *Is it dangerous to use version control histories to study source code evolution?*, pages 79–103. ECOOP 2012–Object-Oriented Programming. Springer, 2012.

[56] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 161–170. IEEE, 2015.

[57] Dewayne Perry, Adam Porter, Michael Wade, Lawrence Votta, and James Perpich. Reducing inspection interval in large-scale software development. *Software Engineering, IEEE Transactions on*, 28(7):695–705, 2002.

[58] Marco Pistoia, Satish Chandra, Stephen Fink, and Eran Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

[59] PMD. Pmd, 2014. URL `http://pmd.sourceforge.net/`. Accessed on: October 2nd, 2014.

[60] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. Understanding the sources of variation in software inspections. *ACM Trans. Softw. Eng. Methodol.*, 7(1): 41–79, 1998.

[61] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. Conducting online surveys in software engineering. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, pages 80–88. IEEE, 2003.

[62] Peter Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.

[63] Peter Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 541–550. ACM, 2011.

[64] Peter Rigby, Daniel German, and Margaret-Anne Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008.

[65] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. Contemporary peer review in action: Lessons from open source development. *Software, IEEE*, 29(6):56–61, 2012.

[66] Peter Rigby, Alberto Bacchelli, Georgios Gousios, and Murtuza Mukadam. *A Mixed Methods Approach to Mining Code Review Data: Examples and a study of multi-commit reviews and pull requests*. Morgan Kaufmann, 2014. To Appear.

[67] Peter Rigby, Daniel German, Laura Cowen, and Margaret-Anne Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Trans. Softw. Eng. Methodol.*, 23(4):1–33, 2014.

[68] Per Runeson, Carina Andersson, Thomas Thelin, Anneliese Andrews, and Tomas Berling. What do we know about defect detection methods? *Software, IEEE*, 23 (3):82–90, 2006.

[69] Nick Rutar, Christian Almazan, and Jeffrey Foster. A comparison of bug finding tools for java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256, 2004.

[70] Joseph Ruthruff, John Penix, David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, pages 341–350. ACM, 2008.

[71] David Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48. ACM, 1998.

[72] Harvey Siy and Lawrence Votta. Does the modern code inspection have value? In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 281. IEEE Computer Society, 2001.

[73] Hannes Tribus, Irene Morrigl, and Stefan Axelsson. *Using Data Mining for Static Code Analysis of C*, volume 7713 of *Lecture Notes in Computer Science*, book section 50, pages 603–614. Springer Berlin Heidelberg, 2012.

[74] Akash Kumar Tripathi and Atul 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*, page 23. ACM, 2014.

[75] Lawrence Votta. Does every inspection need a meeting? *SIGSOFT Softw. Eng. Notes*, 18(5):107–114, 1993.

[76] Stefan Wagner. A literature survey of the quality economics of defect-detection techniques. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 194–203. ACM, 2006.

[77] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. *Comparing Bug Finding Tools with Reviews and Tests*, volume 3502 of *Lecture Notes in Computer Science*, book section 4, pages 40–55. Springer Berlin Heidelberg, 2005.

[78] Stefan Wagner, Florian Deissenboeck, Michael Aichner, Johann Wimmer, and Markus Schwalb. An evaluation of two bug pattern tools for java. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 248–257, 2008.

[79] Fadi Wedyan, Dalal Alrmuny, and James Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 141–150. IEEE, 2009.

[80] Nicholas Zakas. Make –reset the default behavior, remove flag issue #2100 eslint/eslint, 2015. URL `https://github.com/eslint/eslint/issues/2100#issuecomment-98030925`. Accessed on: June 19th, 2015.

[81] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John Hudepohl, and Mladen Vouk. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, 32(4):240–253, 2006.

Appendix A

Prevalence Survey Results

This appendix presents the full data underlying the results of Chapter 3. First, we detail the exact wording of the questions that comprised the questionnaire. Then, Tables A.1 to A.6 show the results of the prevalence analysis for code reviews. The results for ASATs are presented in Tables A.7 to A.12.

Questionnaire

1. Do all developers (contributors and core developers) have to submit a code review for every change? Im asking because many projects only review changes made by contributors.
2. Which code review tools are used?
3. Are static analyzers used? If they are used:
    a) Is passing the checks of the static analyzers necessary for a change to be accepted?
    b) Which static analyzers are used?

Tool	[R] Code Review	[Q] Code Reviews on Changes from Core Developers	[R] Code Review Tools Used	[Q] Code Review Tools Used	Source
7-Zip	Inconclusive	—	—	—	SourceForge
Abilian	—	No	—	GitHub	—
Ace	Yes	Yes	GitHub	GitHub	GitHub
AFNetworking	Yes	—	GitHub	—	GitHub
Angular.js	Yes	—	GitHub	—	GitHub
Animate.css	Yes	—	GitHub	—	GitHub
Ansible	Yes	—	GitHub	—	GitHub
Apache	Yes	—	Bugzilla	—	OpenHub
AppArmor	—	Yes	—	Mailing list	—
Ares Galaxy	Inconclusive	—	—	—	SourceForge
Async	Inconclusive	Yes	—	GitHub	GitHub
Atom	Yes	—	GitHub	—	GitHub
Back in a Minute	Inconclusive	—	—	—	Gitorious
Backbone	Yes	—	GitHub	—	GitHub
Bash	Yes	One core contributor	Mailing list	Mailing list	OpenHub
Bootstrap	Yes	—	GitHub	—	GitHub
Bower	Yes	—	GitHub	—	GitHub
Brackets	Yes	Yes	GitHub	GitHub	GitHub
Brewtarget	Inconclusive	—	—	—	Gitorious
Chakra Packages	Inconclusive	—	—	—	Gitorious
Chart.js	Inconclusive	—	—	—	GitHub
Chosen	Yes	—	GitHub	—	GitHub
Clang	Yes	Yes	Mailing list	Mailing list, Phabricator	Gitorious
CMake	Yes	Yes	Mailing list	Mailing list	Gitorious

Table A.1: Full results for the prevalence analysis for code reviews.

Tool	[R] Code Review	[Q] Code Reviews on Changes from Core Developers	[R] Code Review Tools Used	[Q] Code Review Tools Used	Source
CodeIgniter	Yes	No	GitHub	GitHub	GitHub
Coffeescript	Yes	—	GitHub	—	GitHub
D3	Yes	—	GitHub	—	GitHub
Devise	Inconclusive	—	—	—	GitHub
Diaspora	Yes	Yes	GitHub	GitHub	GitHub
Discourse	Yes	—	GitHub	—	GitHub
Django	Yes	Encouraged	GitHub, Trac	GitHub, Trac	GitHub
Docker	Yes	—	GitHub	—	GitHub
Dungeon Crawl Stone Soup	Yes	No	Mantis	Mantis, Gitorious	Gitorious
Effeckt.css	Yes	—	GitHub	—	GitHub
Elasticsearch	Yes	No	GitHub	GitHub	GitHub
element	Inconclusive	—	—	—	Gitorious
Ember.js	Yes	—	GitHub	—	GitHub
eMule	Yes	—	Forum	—	SourceForge
ERP5	No	—	—	—	Gitorious
Express	Inconclusive	—	—	—	GitHub
FileZilla	Yes	No	Trac	—	SourceForge
Firebug	Yes	—	GitHub	—	OpenHub
Firefox	Yes	—	ReviewBoard, Splinter	—	OpenHub
Flask	Yes	—	GitHub	—	GitHub
Flat—UI	Inconclusive	—	—	—	GitHub
FlightGear	Yes	—	Gitorious	—	Gitorious
Font Awesome	Yes	—	GitHub	—	GitHub

Table A.2: Full results for the prevalence analysis for code reviews (continued).

Tool	[R] Code Review	[Q] Code Reviews on Changes from Core Developers	[R] Code Review Tools Used	[Q] Code Review Tools Used	Source
Foundation	Yes	—	GitHub	—	GitHub
GDB	Yes	Encouraged	Mailing list	Mailing list	Gitorious
Ghost	Yes	—	GitHub	—	GitHub
Gitflow	Yes	—	GitHub	—	GitHub
GitLab	Yes	Yes	GitHub, GitLab	GitHub, GitLab	GitHub
GNU social	Inconclusive	—	—	—	Gitorious
GROMACS	—	Yes	—	Gerrit	—
Grunt	Yes	—	GitHub	—	GitHub
Gulp	Yes	—	GitHub	—	GitHub
Haiku	Yes	No	Trac	Mailing list, Trac	Gitorious
Hammer.js	Yes	—	GitHub	—	GitHub
HHVM	Yes	—	Phabricator, GitHub	—	GitHub
Homebrew	Yes	—	GitHub	—	GitHub
HTML5 Boilerplate	Yes	—	GitHub	—	GitHub
Httpie	Inconclusive	—	—	—	GitHub
Impress.js	Yes	—	GitHub	—	GitHub
io.js	Yes	Yes	GitHub	GitHub	GitHub
Ionic	Yes	—	GitHub	—	GitHub
Jekyll	Yes	—	GitHub	—	GitHub
jQuery	Yes	—	GitHub	—	GitHub
jQuery File Upload	Yes	—	GitHub	—	GitHub
jQuery-Mobile	Yes	—	GitHub	—	GitHub
Kitware	Inconclusive	—	—	—	Gitorious

Table A.3: Full results for the prevalence analysis for code reviews (continued).

Tool	[R] Code Review	[Q] Code Reviews on Changes from Core Developers	[R] Code Review Tools Used	[Q] Code Review Tools Used	Source
Laravel	Yes	—	GitHub	—	GitHub
Leaflet	Yes	Yes	GitHub	GitHub	GitHub
Less.js	Inconclusive	—	—	—	GitHub
LibreOffice	Yes	Encouraged	Gerrit	Gerrit	Gitorious
Lime	Inconclusive	No	—	GitHub	GitHub
Linux	Yes	—	Mailing list	—	GitHub
Meteor	Yes	—	GitHub	—	GitHub
MinGW	Yes	—	SourceForge	—	SourceForge
Modernizr	Inconclusive	Yes	—	GitHub	GitHub
Moment	Yes	—	GitHub	—	GitHub
MySQL	Yes	—	Mailing list, Bug tracker	—	OpenHub
Neovim	Yes	—	GitHub	—	GitHub
Node.js	Yes	—	GitHub	—	GitHub
Normalize.css	Yes	—	GitHub	—	GitHub
NotePad++ Plugin Manager	Yes	—	Unknown	—	SourceForge
NW.js	Inconclusive	—	—	—	GitHub
Oh My Zsh	Inconclusive	—	—	—	GitHub
OpenOCD	Yes	Yes	Gerrit	Gerrit	Gitorious
OpenOffice	Yes	—	Bugzilla	—	OpenHub
openSUSE Factory	Yes	Yes	Open Build Service	Open Build Service	Gitorious
openSUSE YaST	—	Yes	—	GitHub	—

Table A.4: Full results for the prevalence analysis for code reviews (continued).

Tool	[R] Code Review	[Q] Code Reviews on Changes from Core Developers	[R] Code Review Tools Used	[Q] Code Review Tools Used	Source
PDF.js	Yes	Yes	GitHub	GitHub	GitHub
PhantomJS	Yes	—	GitHub	—	GitHub
PHP	Yes	—	GitHub, Mailing list, Bug tracker	—	OpenHub
pJax	Inconclusive	—	GitHub	—	GitHub
Pop	Yes	—	GitHub	—	GitHub
PortableApps	Inconclusive	—	—	—	SourceForge
Profanity	Inconclusive	—	—	—	Gitorious
Pure	Yes	Encouraged	GitHub	GitHub	GitHub
Qt	Yes	Yes	Gerrit	Gerrit	Gitorious
Rails	Yes	—	GitHub	—	GitHub
Ratchet	Yes	—	GitHub	—	GitHub
React	Yes	Yes	GitHub	GitHub, Phabricator	GitHub
Redis	Yes	—	GitHub	—	GitHub
Requests	Yes	—	GitHub	—	GitHub
Reveal.js	Yes	—	GitHub	—	GitHub
Rust	Yes	—	GitHub	—	GitHub
Sails	Yes	—	GitHub	—	GitHub
Select2	Yes	—	GitHub	—	GitHub
Semantic-UI	Inconclusive	One core contributor	—	GitHub	GitHub
Skrollr	Inconclusive	—	—	—	GitHub
SlapOS	Yes	—	Mailing list, Forum	—	Gitorious
Slick	Inconclusive	One core contributor	—	GitHub	GitHub

Table A.5: Full results for the prevalence analysis for code reviews (continued).

Tool	[R] Code Review	[Q] Code Reviews on Changes from Core Developers	[R] Code Review Tools Used	[Q] Code Review Tools Used	Source
Snowdrift	Yes	Yes	Gitorious	Gitorious, GitHub	Gitorious
Socket.io	Yes	—	GitHub	—	GitHub
Subversion	Yes	—	Mailing list	—	OpenHub
Sweetalert	Yes	—	GitHub	—	GitHub
Symfony	Yes	—	GitHub	—	GitHub
Textmate	Yes	One core contributor	GitHub	GitHub	GitHub
Three.js	Yes	—	GitHub	—	GitHub
TrueType core fonts	Inconclusive	—	—	—	SourceForge
Typeahead.js	Yes	—	GitHub	—	GitHub
Ubuntu	Yes	—	Launchpad	—	OpenHub
Underscore	Yes	—	GitHub	—	GitHub
VLC Media Player	Yes	No	Mailing list	Mailing list, Patchwork	SourceForge
Vuze	Inconclusive	No	—	Forum	SourceForge

Table A.6: Full results for the prevalence analysis for code reviews (continued).

Tool	[R] ASATs	[Q] ASATs	[R] Tools Used	[Q] Tools Used	[Q] Enforced	Source
7-Zip	No Evidence	—	—	—	—	SourceForge
Abilian	—	Yes	—	JSLint, ESLint, CSSLint, Pep8, Pyflakes	No	—
Ace	No Evidence	No	—	—	—	GitHub
AFNetworking	No Evidence	—	—	—	—	GitHub
Angular.js	Yes	—	JSHint, JSCS	—	—	GitHub
Animate.css	No Evidence	—	—	—	—	GitHub
Ansible	No Evidence	—	—	—	—	GitHub
Apache	Yes	—	Coverity Scan	—	—	OpenHub
AppArmor	—	Yes	—	Pyflakes	No	—
Ares Galaxy	No Evidence	—	—	—	—	SourceForge
Async	Yes	Yes	JSLint	JSCS, ESLint	No	GitHub
Atom	Yes	—	Coffeelint	—	—	GitHub
Back in a Minute	No Evidence	—	—	—	—	Gitorious
Backbone	Yes	—	JavaScriptLint	—	—	GitHub
Bash	Yes	No	Coverity Scan	—	—	OpenHub
Bootstrap	Yes	—	JSHint, JSCS, CSSLint, HTML Validation	—	—	GitHub
Bower	Yes	—	JSHint	—	—	GitHub
Brackets	Yes	Yes	JSHint, JSLint	JSHint, JSLint, JSONLint	Yes	GitHub
Brewtarget	No Evidence	—	—	—	—	Gitorious
Chakra Packages	No Evidence	—	—	—	—	Gitorious
Chart.js	Yes	—	JSHint, HTML Validator	—	—	GitHub

Table A.7: Full results for the prevalence analysis for ASATs.

Tool	[R] ASATs	[Q] ASATs	[R] Tools Used	[Q] Tools Used	[Q] Enforced	Source
Chosen	No Evidence	—	—	—	—	GitHub
Clang	No Evidence	No	—	—	—	Gitorious
CMake	Yes	Yes	Coverity Scan	Clang	Yes	Gitorious
CodeIgniter	No Evidence	No	—	—	—	GitHub
Coffeescript	No Evidence	—	—	—	—	GitHub
D3	No Evidence	—	—	—	—	GitHub
Devise	No Evidence	—	—	—	—	GitHub
Diaspora	Yes	No	JSHint	—	—	GitHub
Discourse	Yes	—	JSHint	—	—	GitHub
Django	Yes	Yes	Flake8	Flake8	No	GitHub
Docker	No Evidence	—	—	—	—	GitHub
Dungeon Crawl Stone Soup	No Evidence	Yes	—	Clang	No	Gitorious
Effeckt.css	No Evidence	—	—	—	—	GitHub
Elasticsearch	Yes	Yes	FindBugs, PMD, Checkstyle	FindBugs, Checkstyle, PMD	Yes	GitHub
element	No Evidence	—	—	—	—	Gitorious
Ember.js	Yes	—	JSHint, JSCS	—	—	GitHub
eMule	No Evidence	—	—	—	—	SourceForge
ERP5	No Evidence	—	—	—	—	Gitorious
Express	No Evidence	—	—	—	—	GitHub
FileZilla	Yes	Yes	Coverity Scan	Coverity Scan, Cppcheck	No	SourceForge
Firebug	Yes	—	JSHint	—	—	OpenHub
Firefox	Yes	—	Coverity, Klockwork	—	—	OpenHub

Table A.8: Full results for the prevalence analysis for ASATs (continued).

Tool	[R] ASATs	[Q] ASATs	[R] Tools Used	[Q] Tools Used	[Q] Enforced	Source
Flask	No Evidence	—	—	—	—	GitHub
Flat—UI	Yes	—	JSHint, JSCS, CSSLint	—	—	GitHub
FlightGear	No Evidence	—	—	—	—	Gitorious
Font Awesome	No Evidence	—	—	—	—	GitHub
Foundation	Planned	—	—	—	—	GitHub
GDB	Yes	No	Coverity Scan	—	—	Gitorious
Ghost	Yes	—	JSHint, JSCS	—	—	GitHub
Gitflow	No Evidence	—	—	—	—	GitHub
GitLab	Yes	Yes	Hound, Rubocop	Hound	No	GitHub
GNU social	No Evidence	—	—	—	—	Gitorious
GROMACS	—	Yes	—	Clang, Cppcheck, Custom scripts	Yes	—
Grunt	Yes	—	JSHint	—	—	GitHub
Gulp	Yes	—	JSHint	—	—	GitHub
Haiku	Yes	Yes	Coverity Scan, Custom style checker	Coverity Scan, Custom style checker	No	Gitorious
Hammer.js	Yes	—	JSHint, JSCS	—	—	GitHub
HHVM	Yes	—	Coverity Scan, FBLint	—	—	GitHub
Homebrew	No Evidence	—	—	—	—	GitHub
HTML5 Boilerplate	Yes	—	JSHint, JSCS	—	—	GitHub
Httpie	No Evidence	—	—	—	—	GitHub
Impress.js	Yes	—	JSHint	—	—	GitHub
io.js	Yes	Yes	JSLint, CPPLint, Closure Linter	CPPLint, Closure Linter	Yes	GitHub

Table A.9: Full results for the prevalence analysis for ASATs (continued).

Tool	[R] ASATs	[Q] ASATs	[R] Tools Used	[Q] Tools Used	[Q] Enforced	Source
Ionic	Yes	—	JSHint, JSCS	—	—	GitHub
Jekyll	No Evidence	—	—	—	—	GitHub
jQuery	Yes	—	JSHint, JSCS, JSONLint	—	—	GitHub
jQuery File Upload	Yes	—	JSHint	—	—	GitHub
jQuery-Mobile	Yes	—	JSHint	—	—	GitHub
Kitware	No Evidence	—	—	—	—	Gitorious
Laravel	No Evidence	—	—	—	—	GitHub
Leaflet	Yes	Yes	ESLint	ESLint	Yes	GitHub
Less.js	Yes	—	JSHint, JSCS	—	—	GitHub
LibreOffice	Yes	Yes	Coverity Scan	Coverity Scan, Cppcheck, Clang	No	Gitorious
Lime	Yes	Yes	Gofmt	Gofmt	Yes	GitHub
Linux	Yes	—	Smatch, Coccinelle, Coverity Scan	—	—	GitHub
Meteor	Yes	—	ESLint	—	—	GitHub
MinGW	No Evidence	—	—	—	—	SourceForge
Modernizr	Yes	Yes	JSHint	JSHint, JSCS	Yes	GitHub
Moment	Yes	—	JSHint, JSCS	—	—	GitHub
MySQL	No Evidence	—	—	—	—	OpenHub
Neovim	Yes	—	Coverity Scan, Clang, Custom linter	—	—	GitHub
Node.js	Yes	—	JSLint, CPPLint, Closure Linter	—	—	GitHub

Table A.10: Full results for the prevalence analysis for ASATs (continued).

Tool	[R] ASATs	[Q] ASATs	[R] Tools Used	[Q] Tools Used	[Q] Enforced	Source
Normalize.css	No Evidence	—	—	—	—	GitHub
NotePad++ Plugin Manager	No Evidence	—	—	—	—	SourceForge
NW.js	No Evidence	—	—	—	—	GitHub
Oh My Zsh	No Evidence	—	—	—	—	GitHub
OpenOCD	Yes	Yes	Clang	Clang	No	Gitorious
OpenOffice	Yes	—	Coverity Scan	—	—	OpenHub
openSUSE Factory	Yes	Yes	RPMLint	RPMlint	Yes	Gitorious
openSUSE YaST	—	Yes	—	RuboCop	Yes	—
PDF.js	Yes	Yes	JSHint	JSHint	Yes	GitHub
PhantomJS	No Evidence	—	—	—	—	GitHub
PHP	Yes	—	Coverity Scan	—	—	OpenHub
pJax	No Evidence	—	—	—	—	GitHub
Pop	Yes	—	Linter (unspecified)	—	—	GitHub
PortableApps	No Evidence	—	—	—	—	SourceForge
Profanity	No Evidence	—	—	—	—	Gitorious
Pure	Yes	Yes	CSSLint	CSSLint	Yes	GitHub
Qt	Yes	Yes	Coverity Scan	Coverity Scan, Clang	No	Gitorious
Rails	Yes	—	W3C Validators	—	—	GitHub
Ratchet	Yes	—	JSHint, JSCS, CSSLint, HTML Validation	—	—	GitHub
React	Yes	Yes	JSHint	JSHint, Code Climate	No	GitHub
Redis	No Evidence	—	—	—	—	GitHub

Table A.11: Full results for the prevalence analysis for ASATs (continued).

Tool	[R] ASATs	[Q] ASATs	[R] Tools Used	[Q] Tools Used	[Q] Enforced	Source
Requests	Yes	—	Pylint, Pyflakes	—	—	GitHub
Reveal.js	Yes	—	JSHint	—	—	GitHub
Rust	Yes	—	Built—in Rust	—	—	GitHub
Sails	Yes	—	JSHint	—	—	GitHub
Select2	Yes	—	JSHint	—	—	GitHub
Semantic-UI	Yes	Yes	JSHint, CSSLint	JSHint	No	GitHub
Skrollr	Yes	—	JSHint	—	—	GitHub
SlapOS	No Evidence	—	—	—	—	Gitorious
Slick	No Evidence	No	—	—	—	GitHub
Snowdrift	Yes	Yes	Hlint	HLint	Yes	Gitorious
Socket.io	No Evidence	—	—	—	—	GitHub
Subversion	Yes	—	Coverity Scan	—	—	OpenHub
Sweetalert	Yes	—	JSHint	—	—	GitHub
Symfony	Yes	—	PHPCS, check_cs script	—	—	GitHub
Textmate	No Evidence	No	—	—	—	GitHub
Three.js	No Evidence	—	—	—	—	GitHub
TrueType core fonts	No Evidence	—	—	—	—	SourceForge
Typeahead.js	Yes	—	JSHint	—	—	GitHub
Ubuntu	Yes	—	Coverity	—	—	OpenHub
Underscore	Yes	—	ESLint	—	—	GitHub
VLC Media Player	Yes	Yes	Coverity Scan	Coverity Scan, Clang	No	SourceForge
Vuze	Yes	Yes	Coverity Scan	Coverity Scan	No	SourceForge

Table A.12: Full results for the prevalence analysis for ASATs (continued).

Appendix B

Full Results for the ASAT Configurations Analysis

This appendix presents the full data underlying the results of Chapter 4. Tables B.1 and B.2 show those results belonging to the Figures 4.3 to 4.6. Additionally, Tables B.3 to B.15 provide the underlying results for Tables 4.5 to 4.7. Finally, Figure B.1 and Table B.16 show the distribution of built-in warning rules per tool. Figures B.2 and B.3 then show the non-normalized data of Figures 4.3 and 4.4.

Category\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop	Median
Check	0	0	9.5	0	0	0	3.3	0	2	3.3
Concurrency	4.8	1.6	9.1	0	0	0	4.6	0	0	4.7
Error Handling	2.4	3.4	10.2	0	0	0	6.5	5.8	3.9	4.9
Interface	7.1	5.2	7.5	0	1.8	0	6.2	20.6	0.5	6.2
Logic	8.1	7	8.5	0	12	14.3	6.5	0	1.9	8.1
Migration	2.5	0	9	0	0	0	5.3	0	25.6	7.2
Resource	0	9.7	7.8	18.4	21	13.7	5.5	0	0.2	9.7
Best Practices	3.3	6.9	8.1	12.1	7.3	17.7	4.2	13.7	2	7.3
Code Structure	4.8	0	1	0	0	0	5.2	0	0	4.8
Documentation	4.5	3.2	0	3.2	0	21.3	1	22.5	1.4	3.2
Conventions										
Metric	5.4	7.2	0	11.2	3.9	0	17.7	16.8	54	11.2
Naming Conventions	12.7	18	0	18.5	20.8	0	6.9	11.7	2.5	12.7
Object Oriented Design	7.3	0	10.2	0	0	0	3.4	0	0	7.3
Redundancies	8.4	8.1	8.8	23.3	20.8	13.7	8.5	5.2	0.9	8.5
Simplifications	17.1	19.2	10.2	0	0	6.8	6.9	0	2.1	8.6
Style Conventions	11.6	10.3	0	13.4	12.4	12.4	8.3	3.7	3.1	11

Table B.1: The distribution of rules that are enabled by developers per tool, according to our classification. The last column displays the median value for every category. Normalized to the number of possible rules in a category. All numbers are percentages and all columns, except the last, sum up to 100 (approximately, due to rounding).

Category\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop	Median
Check	0	0	0.3	0	0	0	8	0	5.5	5.5
Concurrency	8.1	9.5	2.6	0	0	0	5.7	0	0	6.9
Error Handling	9.1	1.6	0.4	0	0	0	2.5	5.6	7.7	4.1
Interface	7.2	1.5	4.1	0	9.5	0	6.6	5.2	1.8	5.2
Logic	6.7	2.5	8.3	0	12.2	28.5	2.1	2.8	6.6	6.7
Migration	9	0	1.1	0	0	0	1.6	0.8	21.7	1.6
Resource	0	2.4	3.6	3.7	1.4	2	10.5	3.5	2.8	3.2
Best Practices	8.7	10	6.3	22.2	14.5	7.3	8	13	4.8	8.7
Code Structure	8.1	0	60.7	0	0	0	12.5	9.4	0	10.9
Documentation	8.2	16.5	0	3.7	0	0.3	2.1	24.7	13.4	8.2
Conventions										
Metric	7.9	9.8	0	7.4	11.4	0	5.9	12.5	16.6	9.8
Naming Conventions	4.8	33.2	0	29.6	22.1	0	16.2	7.6	4.7	16.2
Object Oriented Design	7.1	0	0	0	0	0	6.2	0	0	6.7
Redundancies	6.6	2.4	12.4	11.1	11.5	17.3	6.4	6.5	2.3	6.6
Simplifications	3.1	3.9	0.2	0	0	19.6	2.1	3.5	8.2	3.5
Style Conventions	5.3	6.6	0	22.2	17.5	24.9	3.9	4.8	4.1	6

Table B.2: The distribution of rules that are disabled by developers per tool, according to our classification. The last column displays the median value for every category. Normalized to the number of possible rules in a category. All numbers are percentages and all columns, except the last, sum up to 100 (approximately, due to rounding).

Default Rule	Disabled	Configured
no-underscore-dangle	59.5	0
strict	50.7	14.4
curly	25.2	34.7
no-constant-condition	23.2	0
space-infix-ops	22.8	8.6
new-parens	20.1	0
no-new-func	19.6	0
no-new-object	19.4	0
no-use-before-define	12.6	0
camelcase	11	10.8
quotes	8.4	72.2
new-cap	8.1	54.7
consistent-return	7.8	0
eol-last	6.2	0
no-console	5.8	0
no-new	4.8	0
no-shadow	4.6	0
no-alert	4.3	0
no-unused-expressions	3.4	0
eqeqeq	3.2	37.7
dot-notation	3	10.6
no-unused-vars	2.9	41.8
no-multi-spaces	2.8	3.6
no-process-exit	2.8	0
no-trailing-spaces	2.3	0
key-spacing	1.9	4.5
no-loop-func	1.2	0
semi	1.2	15.6
yoda	1.2	7.5
global-strict	1.2	4.3
no-extend-native	1.1	0
no-cond-assign	0.9	7.9
no-spaced-func	0.9	0
no-empty	0.9	0
no-mixed-spaces-and-tabs	0.8	10.8
no-undef	0.8	0
no-return-assign	0.8	0
no-new-wrappers	0.7	0
no-sparse-arrays	0.7	0
no-multi-str	0.7	0
no-catch-shadow	0.7	0
no-wrap-func	0.6	0
comma-spacing	0.5	3.6
no-extra-semi	0.5	0

Default Rule	Disabled	Configured
no-array-constructor	0.5	0
no-debugger	0.5	0
no-redeclare	0.5	0
valid-typeof	0.5	0
space-return-throw-case	0.5	0
no-caller	0.4	0
no-labels	0.4	0
no-control-regex	0.4	0
no-extra-strict	0.4	0
no-fallthrough	0.4	0
space-unary-ops	0.4	4.5
no-native-reassign	0.3	0
no-eval	0.3	0
no-script-url	0.3	0
no-extra-bind	0.3	0
no-irregular-whitespace	0.3	0
no-inner-declarations	0.2	0
no-undef-init	0.2	0
no-delete-var	0.2	0
no-octal-escape	0.2	0
no-proto	0.2	0
no-extra-boolean-cast	0.2	0
no-regex-spaces	0.2	0
no-func-assign	0.2	0
no-implied-eval	0.1	0
no-with	0.1	0
no-octal	0.1	0
no-obj-calls	0.1	0
no-sequences	0.1	0
no-dupe-keys	0.1	0
no-empty-class	0.1	0
no-unreachable	0.1	0
no-iterator	0.1	0
no-ex-assign	0.1	0
no-empty-label	0.1	0
no-lone-blocks	0.1	0
no-shadow-restricted-names	0.1	0
no-invalid-regexp	0.1	0
use-isnan	0.1	0
no-negated-in-lhs	0.1	0
no-label-var	0	0
comma-dangle	0	0
no-dupe-args	0	0
semi-spacing	0	0

Table B.3: How often a default rule of ESLint is either disabled or configured by developers. 100% represents 4407 configurations.

Default Disabled Rule	Enabled
brace-style	55.7
wrap-iife	32.9
space-after-keywords	32.2
radix	31.4
no-floating-decimal	31.3
consistent-this	31.1
no-nested-ternary	29.8
func-style	29.2
no-else-return	28.1
no-lonely-if	27.6
space-in-brackets	26.8
no-div-regex	25
no-extra-parens	24.5
max-depth	22.9
max-params	22.6
guard-for-in	12.5
no-self-compare	10.8
no-space-before-semi	9.4
no-comma-dangle	9.3
no-eq-null	7.4
block-scoped-var	7
valid-jsdoc	7
no-warning-comments	6.1
default-case	6
max-len	5.6
handle-callback-err	5.4
max-nested-callbacks	5.4
no-path-concat	5
comma-style	4.6
no-mixed-requires	4.4
no-bitwise	4.4
space-before-blocks	4.3
space-in-parens	4.1
no-undefined	3.8
spaced-line-comment	3.7
complexity	3.7
no-new-require	3.2
no-reserved-keys	3.1
no-multiple-empty-lines	3
quote-props	2.8
wrap-regex	2.4
no-void	2.4
no-sync	2.4
one-var	2.2
no-plusplus	2.2
func-names	2.2
max-statements	2
indent	1.9
space-after-function-name	1.8
vars-on-top	1.8
padded-blocks	1.5
operator-assignment	1.4
no-process-env	1.2
sort-vars	1.1
no-ternary	1
no-restricted-modules	0.8
no-inline-comments	0.7
space-before-function-parentheses	0.6
no-throw-literal	0.4
no-var	0.3
generator-star	0.2

Table B.4: How often a default disabled rule of ESLint is enabled by developers. 100% represents 4407 configurations.

Default Rule	Disabled
MutableStaticFields	11.8
FindUnsatisfiedObligation	10.6
FindReturnRef	9
SerializableIdiom	8.8
ComparatorIdiom	8.5
InfiniteLoop	7.7
FindSqlInjection	7.2
FindNakedNotify	3.8
SwitchFallthrough	3.7
WaitInLoop	3.6
CrossSiteScripting	3.6
CloneIdiom	3.5
FindBadCast2	3.1
FindUnconditionalWait	2.9
MethodReturnCheck	2.7
CheckTypeQualifiers	2.5
BadResultSetAccess	2.4
NumberConstructor	2.4
UnreadFields	2.3
StaticCalendarDetector	2.1
Naming	1.9
DontUseEnum	1.5
RuntimeExceptionCapture	1.4
DoInsideDoPrivileged	1.4
InvalidJUnitTest	1.4
AppendingToAnObjectOutputStream	1.2
HugeSharedStringConstants	1.2
BadlyOverriddenAdapter	1.2
MultithreadedInstanceAccess	1.1
FormatStringChecker	1
RedundantInterfaces	1
UncallableMethodOfAnonymousClass	1
CheckImmutableAnnotation	1
FindUncalledPrivateMethods	1
StringConcatenation	0.9
BooleanReturnNull	0.8
FindPuzzlers	0.8
DontIgnoreResultOfPutIfAbsent	0.8
ReadReturnShouldBeChecked	0.8
WrongMapIterator	0.7
LostLoggerDueToWeakReference	0.7
FindInconsistentSync2	0.5
FindDoubleCheck	0.4
FindJSR166LockMonitorenter	0.4
SynchronizeOnClassLiteralNotGetClass	0.4
SynchronizingOnContentsOfFieldToProtect-Field	0.4
FindMismatchedWaitOrNotify	0.4
FindUnreleasedLock	0.3
FindDeadLocalStores	0.3
IncompatMask	0.3
PreferZeroLengthArrays	0.3
LazyInit	0.3
InfiniteRecursiveLoop	0.3
RepeatedConditionals	0.2
InitializationChain	0.2
InheritanceUnsafeGetResource	0.2
VarArgsProblems	0.2
SuperfluousInstanceOf	0.2
OverridingEqualsNotSymmetrical	0.2
InconsistentAnnotations	0.2
XMLFactoryBypass	0.2

Default Rule	Disabled
FindNullDerefsInvolvingNonShortCircuitEvaluation	0.2
DefaultEncodingDetector	0.2
DumbMethods	0.2
NoteUnconditionalParamDerefs	0.2
FindSpinLoop	0.2
VolatileUsage	0.2
StartInConstructor	0.2
FindUnsyncGet	0.2
FindEmptySynchronizedBlock	0.2
MutableLock	0.2
SynchronizeAndNullCheckField	0.2
SynchronizationOnSharedBuiltinConstant	0.2
FindSleepWithLockHeld	0.2
FinalizerNullsFields	0.1
FindLocalSelfAssignment2	0.1
FindFinalizeInvocations	0.1
FindFieldSelfAssignment	0.1
IteratorIdioms	0.1
DontCatchIllegalMonitorStateException	0.1
ConfusionBetweenInheritedAndOuterMethod	0.1
FindNonShortCircuit	0.1
DroppedException	0.1
LoadOfKnownNullValue	0
FindOpenStream	0
EqualsOperandShouldHaveClassCompatibleWithThis	0
IntCast2LongAsInstant	0
BadUseOfReturnValue	0
SuspiciousThreadInterrupted	0
FindTwoLockWait	0
OptionalReturnNull	0
InitializeNonnullFieldsInConstructor	0
FindFloatEquality	0
FindUnrelatedTypesInGenericContainer	0
FindSelfComparison2	0
FindRefComparison	0
FindHEmismatch	0
FindUseOfNonSerializableValue	0
InefficientIndexOf	0
URLProblems	0
AtomicityProblem	0
FindNullDeref	0
IDivResultCastToDouble	0
ExplicitSerialization	0
FindMaskedFields	0
FindBadForLoop	0
ConfusedInheritance	0
DumbMethodInvocations	0
UnnecessaryMath	0
FindSelfComparison	0
FindUselessControlFlow	0
FindUninitializedGet	0
InstantiateStaticClass	0
FindRoughConstants	0
QuestionableBooleanAssignment	0
DuplicateBranches	0
ReadOfInstanceFieldInMethodInvokedByConstructorInSuperclass	0
InefficientToArray	0
FindRunInvocations	0
BadSyntaxForRegularExpression	0
CheckRelaxingNullnessAnnotation	0

Table B.5: How often a default rule of FindBugs is disabled by developers. 100% represents 2077 configurations.

Default Disabled Rule	Enabled
EmptyZipFileEntry	92.6
CallToUnsupportedMethod	38
UselessSubclassMethod	27.7
PublicSemaphores	26.8
InefficientMemberAccess	10.4
FindCircularDependencies	9.7
BadAppletConstructor	6.5
CheckExpectedWarnings	3

Table B.6: How often a default disabled rule of FindBugs is enabled by developers. 100% represents 2077 configurations.

Default Rule	Disabled	Configured
strict	13.6	0
curly	9.9	0
bitwise	7.2	0
forin	7.2	0
devel	6.9	0
browser	6.7	0
unused	6.6	58
noempty	4.8	0
eqeqeq	4.5	0
noarg	1.7	0
undef	1.4	0
indent	1	44.1
freeze	0.6	0
maxerr	0.3	11.9
mocha	0.2	0
nonbsp	0	0

Table B.7: How often a default rule of JSHint is either disabled or configured by developers. 100% represents 108604 configurations.

Default Disabled Rule	Enabled
immed	60.8
newcap	56.9
node	55.8
latedef	52.7
quotmark	50.2
camelcase	34.3
esnext	33.5
eqnull	31.6
boss	23.6
expr	22.3
sub	21
jquery	17.1
nonew	15.4
maxlen	10
evil	9
laxbreak	8.1
validthis	7.6
es5	7.3
maxdepth	7.2
globalstrict	6.8
laxcomma	6.8
loopfunc	6.4
asi	6.3
debug	6
maxparams	5.4
wsh	4.5
maxstatements	4.4
maxcomplexity	4.2
multistr	2.4
plusplus	2.4
funcscope	2.4
shadow	2.2
supernew	2.2
proto	1.9
scripturl	1.4
nonstandard	1.2
lastsemic	1
worker	0.8
noyield	0.8
jasmine	0.7
rhino	0.7
iterator	0.7
qunit	0.6
browserify	0.5
yui	0.2
moz	0.2
dojo	0.1
notypeof	0.1
couch	0.1
mootools	0.1
prototypejs	0.1
shelljs	0

Table B.8: How often a default disabled rule of JSHint is enabled by developers. 100% represents 108604 configurations.

Default Rule	Disabled
inc_dec_within_stmt	13.7
anon_no_return_value	11.4
legacy_control_comments	7.9
comparison_type_conv	7.7
missing_default_case	7.7
useless_void	7.1
equal_as_assign	6.6
comma_separated_stmts	6.5
no_return_value	5.5
empty_statement	4.1
missing_break_for_last_case	4.1
ambiguous_nested_stmt	3.9
useless_comparison	3.9
leading_decimal_point	3.9
trailing_decimal_point	3.8
ambiguous_newline	1.6
redeclared_var	1.2
var_hides_arg	0.7
octal_number	0.6
misplaced_regex	0.4
parseint_missing_radix	0.3
missing_break	0.2
missing_semicolon	0.1
use_of_label	0.1
default_not_at_end	0.1
useless_assign	0.1
nested_comment	0
unreachable_code	0
ambiguous_else_stmt	0
assign_to_function_call	0
context	0
dup_option_explicit	0
duplicate_case_in_switch	0
duplicate_formal	0
jsl_cc_not_understood	0
legacy_cc_not_understood	0
meaningless_block	0
multiple_plus_minus	0
partial_option_explicit	0
trailing_comma_in_array	0
with_statement	0
lambda_assign_requires_semicolon	0

Table B.9: How often a default rule of JSL is disabled by developers.
100% represents 5139 configurations.

Default Disabled Rule	Enabled
always_use_option_explicit	6.3
missing_option_explicit	4
block_without_braces	1.5
jscript_function_extensions	0

Table B.10: How often a default disabled rule of JSL is enabled by developers. 100% represents 5139 configurations.

Default Rule	Disabled
ShortVariable	7.4
LongVariable	6.2
LocalVariableCouldBeFinal	3
MethodArgumentCouldBeFinal	2.7
AvoidDuplicateLiterals	2.7
DataflowAnomalyAnalysis	2.2
AbstractNaming	2.1
TooManyMethods	1.9
JUnitTestsShouldIncludeAssert	1.8
AvoidInstantiatingObjectsInLoops	1.7
OnlyOneReturn	1.7
AtLeastOneConstructor	1.6
UnusedFormalParameter	1.5
ImmutableField	1.5
ConfusingTernary	1.5
BeanMembersShouldSerialize	1.3
TooManyStaticImports	1.3
ShortMethodName	1.2
AvoidFinalLocalVariable	1.2
EmptyCatchBlock	1.2
AvoidSynchronizedAtMethodLevel	1.1
EmptyMethodInAbstractClassShouldBeAbstract	1
JUnitTestContainsTooManyAsserts	1
UnusedModifier	1
CyclomaticComplexity	1
NullAssignment	1
JUnitAssertionsShouldIncludeMessage	0.9
CallSuperInConstructor	0.9
UseLocaleWithCaseConversions	0.9
DefaultPackage	0.9
AvoidThrowingRawExceptionTypes	0.8
LawOfDemeter	0.8
ExcessiveImports	0.8
TestClassWithoutTestCases	0.8
AssignmentInOperand	0.7
BooleanGetMethodName	0.7
BooleanInversion	0.7
ConstructorCallsOverridableMethod	0.7
AvoidUsingHardCodedIP	0.7
CloseResource	0.7
AbstractClassWithoutAbstractMethod	0.6
UseStringBufferForStringAppends	0.6
AvoidReassigningParameters	0.6
CollapsibleIfStatements	0.6
UncommentedEmptyMethod	0.6
AvoidThreadGroup	0.6
AvoidLiteralsInIfCondition	0.5
LoosePackageCoupling	0.5
UnnecessaryLocalBeforeReturn	0.5
UseAssertTrueInsteadOfAssertEquals	0.5
InsufficientStringBufferDeclaration	0.5
CheckResultSet	0.5
UseConcurrentHashMap	0.5
AvoidDeeplyNestedIfStmts	0.4
TooFewBranchesForASwitchStatement	0.4
AvoidFieldNameMatchingTypeName	0.4
IfStmtsMustUseBraces	0.4
SimpleDateFormatNeedsLocale	0.4
ShortClassName	0.4
UncommentedEmptyConstructor	0.4
UnusedImports	0.4
UselessParentheses	0.4
SimplifyStartsWith	0.4
JUnitSpelling	0.4
UseObjectForClearerAPI	0.4

Default Rule	Disabled
NPathComplexity	0.3
VariableNamingConventions	0.3
PreserveStackTrace	0.3
AvoidFieldNameMatchingMethodName	0.3
UnusedPrivateMethod	0.3
InefficientEmptyStringCheck	0.3
CommentSize	0.3
CouplingBetweenObjects	0.3
AvoidUsingVolatile	0.3
FinalFieldCouldBeStatic	0.3
TooManyFields	0.3
SwitchDensity	0.3
SignatureDeclareThrowsException	0.3
IfElseStmtsMustUseBraces	0.3
ReturnEmptyArrayRatherThanNull	0.3
ConsecutiveLiteralAppends	0.3
UseAssertEqualsInsteadOfAssertTrue	0.2
UseAssertSameInsteadOfAssertTrue	0.2
CommentRequired	0.2
ProperCloneImplementation	0.2
DuplicateImports	0.2
AvoidUsingShortType	0.2
MissingBreakInSwitch	0.2
JUnit4TestShouldUseTestAnnotation	0.2
UnusedImports	0.2
ExcessiveMethodLength	0.2
UnnecessaryConstructor	0.2
SignatureDeclareThrowsException	0.2
CompareObjectsWithEquals	0.2
ExcessivePublicCount	0.2
GodClass	0.2
UselessOverridingMethod	0.2
ClassWithOnlyPrivateConstructorsShouldBeFinal	0.2
StringInstantiation	0.2
LooseCoupling	0.2
AvoidCatchingThrowable	0.2
CloneThrowsCloneNotSupportedException	0.2
SwitchStmtsShouldHaveDefault	0.2
NonThreadSafeSingleton	0.2
AvoidPrefixingMethodParameters	0.2
UnnecessaryParentheses	0.2
OverrideBothEqualsAndHashcode	0.1
MissingStaticMethodInNonInstantiatableClass	0.1
ReplaceVectorWithList	0.1
MoreThanOneLogger	0.1
AddEmptyString	0.1
EmptyCatchBlock	0.1
RedundantFieldInitializer	0.1
UnusedLocalVariable	0.1
AvoidCatchingGenericException	0.1
AccessorClassGeneration	0.1
ExcessiveParameterList	0.1
ReplaceHashtableWithMap	0.1
AssignmentToNonFinalStatic	0.1
WhileLoopsMustUseBraces	0.1
EqualsNull	0.1
ExcessiveClassLength	0.1
UnusedPrivateField	0.1
FieldDeclarationsShouldBeAtStartOfClass	0.1
GuardLogStatementJavaUtil	0.1
AvoidRethrowingException	0.1
AvoidConstantsInterface	0.1
BooleanInstantiation	0.1
AvoidUsingOctalValues	0.1
AvoidUsingNativeCode	0.1

Table B.11: How often a rule of PMD is disabled by developers. 100% represents 7452 configurations. Because of size constraints, the table only displays the 130 rules that were disabled the most.

Default Rule	Disabled
W0142	36.9
C0111	34.8
W0511	25.7
C0103	23.5
R0903	21.9
R0904	18.7
R0201	18.3
W0622	17.2
R0913	16
R0801	15.8
R0914	15.5
W0141	15.5
R0902	15.3
R0912	15.3
W0232	14.9
E1101	14.7
C0301	14.7
W0212	14.6
R0915	14.4
C0302	13.7
W0603	13.6
E1103	12.8
W0201	12
W0703	11
W0614	10.9
W0613	10.4
W0402	9.8
W0401	9.8
W0404	9.6
R0911	8.9
W0702	8.6
W0403	8.1
W0105	7.7
W0221	7.1
W0621	7
W0122	7
E1002	6.9
R0922	6.9
R0901	6.8
R0921	6.6
W0612	6.4
C0321	6.2
W0223	6.2
W0231	6.1
W0102	6
W0611	6
E0102	5.7
W0631	5.7
W0602	5.6
E0602	5.6
W0311	5.6
W0104	5.5
C0322	5.4
C0324	5.4
C0323	5.3
W0108	5.3
C0112	5.2
W0301	5.1
E0101	5.1
W0601	5.1
W0701	5
W1201	4.9
E0611	2.8
missing-docstring	2.5
too-few-public-methods	1.9

Default Rule	Disabled
star-args	1.7
R0401	1.5
too-many-public-methods	1.5
E1120	1.4
invalid-name	1.4
C0325	1.1
C0330	1
no-member	1
no-init	0.9
C0326	0.9
no-self-use	0.8
W0312	0.8
fixme	0.8
E0202	0.7
R0924	0.7
C0303	0.7
no-name-in-module	0.6
R0923	0.6
W0110	0.6
bad-continuation	0.6
too-many-ancestors	0.6
too-many-arguments	0.5
E1102	0.5
line-too-long	0.5
too-many-instance-attributes	0.5
W1623	0.5
unused-argument	0.5
E1121	0.4
E1604	0.4
W1601	0.4
W1621	0.4
W1631	0.4
C0304	0.4
super-on-old-class	0.4
W1001	0.4
attribute-defined-outside-init	0.4
E1123	0.4
too-many-locals	0.4
protected-access	0.4
abstract-method	0.3
too-many-lines	0.3
bad-whitespace	0.3
global-statement	0.3
too-many-branches	0.3
old-style-class	0.3
duplicate-code	0.3
broad-except	0.3
superfluous-parens	0.3
W0710	0.2
E0211	0.2
useless-else-on-loop	0.2
E0603	0.2
trailing-whitespace	0.2
W0120	0.2
E0213	0.2
W1202	0.2
E1003	0.2
too-many-statements	0.2
C0203	0.2
unpacking-non-sequence	0.1
W0233	0.1
too-many-function-args	0.1
too-many-return-statements	0.1
E0203	0.1
unused-import	0.1

Table B.12: How often a rule of Pylint is disabled by developers. 100% represents 4065 configurations. Because of size constraints, the table only displays the 130 rules that were disabled the most.

Default Disabled Rule	Enabled
I0021	0
E1601	0
E1602	0
E1603	0
E1604	0
E1605	0
E1606	0
E1607	0
E1608	0
I0020	0
W0704	0
W1601	0
W1602	0
W1603	0
W1604	0
W1605	0
W1606	0
W1607	0
W1608	0
W1609	0
W1610	0
W1611	0
W1612	0
W1613	0
W1614	0
W1615	0
W1616	0
W1617	0
W1618	0
W1619	0
W1620	0
W1621	0
W1622	0
W1623	0
W1624	0
W1625	0
W1626	0
W1627	0
W1628	0
W1629	0
W1630	0
W1632	0
W1633	0
W1634	0
W1635	0
W1636	0
W1637	0
W1638	0
W1639	0

Table B.13: How often a default disabled rule of Pylint is enabled by developers. 100% represents 4065 configurations.

Default Rule	Disabled	Configured
Documentation	34.5	0
HashSyntax	22.4	21.2
LineLength	22.1	49.2
StringLiterals	21.5	6.2
IfUnlessModifier	17.8	0.9
AlignParameters	16.8	2.4
ClassLength	16.2	7.2
SignalException	11.9	3
WhileUntilModifier	10.5	0.6
MethodLength	10.3	36
SpaceAroundEqualsInParameterDefault	9.4	1.1
DeprecatedHashMethods	8.3	0
Lambda	7.1	0
CyclomaticComplexity	6.8	8.9
FileName	6.7	2.4
SingleSpaceBeforeFirstArg	6.5	0
TrailingComma	5.9	2.3
ClassAndModuleChildren	5.8	1.7
RegexpLiteral	5.7	3.3
AsciiComments	5.6	0
DoubleNegation	5.5	0
TrivialAccessors	4.9	2.2
AssignmentInCondition	4.9	0.9
NumericLiterals	4.9	2.9
HandleExceptions	4.4	0
CaseEquality	4.2	0
GuardClause	4.1	0.6
AccessorMethodName	4	0
Blocks	4	0
DotPosition	3.9	3.8
FormatString	3.7	0.7
WordArray	3.6	4.2
ClassVars	3.4	0
SingleLineBlockParams	3.4	0.6
SpecialGlobalVars	3.4	0
ParameterLists	3.3	6.8
AndOr	3.1	0.7
EachWithObject	3.1	0
RaiseArgs	3	1.8
PerlBackrefs	3	0
Next	3	0.2
UselessAssignment	2.9	0
PerceivedComplexity	2.8	2.9
Void	2.7	0
UnusedBlockArgument	2.6	0
CommentAnnotation	2.6	0.7
PercentLiteralDelimiters	2.6	3.2
AmbiguousRegexpLiteral	2.6	0
ColonMethodCall	2.5	0
RedundantSelf	2.5	0
AbcSize	2.4	3.8
AmbiguousOperator	2.4	0
BlockNesting	2.4	6.4
GlobalVars	2.4	0.8
ParenthesesAsGroupedExpression	2.4	0
NegatedIf	2.4	0
Not	2.4	0
RescueModifier	2.4	0
ConstantName	2.2	0
BracesAroundHashParameters	2.2	1.1
ModuleFunction	2.2	0
LineEndConcatenation	2.2	0
Proc	2.2	0
SingleLineMethods	2.2	0.9
NilComparison	2.1	0
TrailingBlankLines	2.1	0.8
EmptyLiteral	2.1	0
EmptyLines	2.1	0

Default Rule	Disabled	Configured
SelfAssignment	2	0
Alias	2	0
OneLineConditional	2	0
OpMethod	2	0
LeadingCommentSpace	2	0
Loop	2	0
RedundantBegin	1.9	0
UnusedMethodArgument	1.9	0
VariableInterpolation	1.9	0
DeprecatedClassMethods	1.9	0
SpaceInsideHashLiteral-Braces	1.9	3.9
UnderscorePrefixedVariable-Name	1.8	0
RedundantReturn	1.8	0.9
IndentationWidth	1.8	0.6
Attr	1.8	0
Eval	1.8	0
EvenOdd	1.8	0
NegatedWhile	1.8	0
PredicateName	1.7	2
LiteralInCondition	1.7	0
SpaceInsideBrackets	1.7	0
RequireParentheses	1.7	0
AccessModifierIndentation	1.7	3.1
ElseLayout	1.7	0
SpaceAroundOperators	1.6	0.4
ArrayJoin	1.6	0
Delegate	1.6	0
TrailingWhitespace	1.6	0
WhenThen	1.6	0
CharacterLiteral	1.5	0
LambdaCall	1.5	0.6
ConditionPosition	1.5	0
AsciiIdentifiers	1.5	0
CaseIndentation	1.5	3.4
IfWithSemicolon	1.5	0
LiteralInInterpolation	1.5	0
AlignHash	1.5	1.5
SpaceInsideParens	1.4	0
InvalidCharacterLiteral	1.4	0
FlipFlop	1.4	0
MultilineBlockChain	1.4	0
Semicolon	1.3	1.3
MethodName	1.3	0.7
RescueException	1.3	0
SpaceAfterComma	1.3	0
IndentHash	1.2	0.9
IndentationConsistency	1.2	0.4
SpaceBeforeBlockBraces	1.2	0.8
BlockAlignment	1.2	0
UnlessElse	1.2	0
EmptyLinesAroundAccess-Modifier	1.2	0
SpaceInsideBlockBraces	1.1	0.9
EmptyLinesAroundBlock-Body	1.1	0.2
VariableName	1.1	0.6
Tab	1.1	0
MethodCallParentheses	1	0
ShadowingOuterLocalVari-able	1	0
CommentIndentation	1	0
EndAlignment	1	3.1
ParenthesesAroundCondition	1	0.9
MethodDefParentheses	1	0.9
AlignArray	0.9	0

Table B.14: How often a default rule of RuboCop is either disabled or configured by developers. 100% represents 10036 configurations. Because of size constraints, the table only displays the 130 rules that were disabled the most.

Default Disabled Rule	Enabled
CollectionMethods	6.4
Encoding	3.3
SymbolArray	0.7
MethodCalledOnDoEndBlock	0.2
ExtraSpacing	0.1
InlineComment	0
AutoResourceCleanup	0
MissingElse	0

Table B.15: How often a default disabled rule of RuboCop is enabled by developers. 100% represents 10036 configurations.

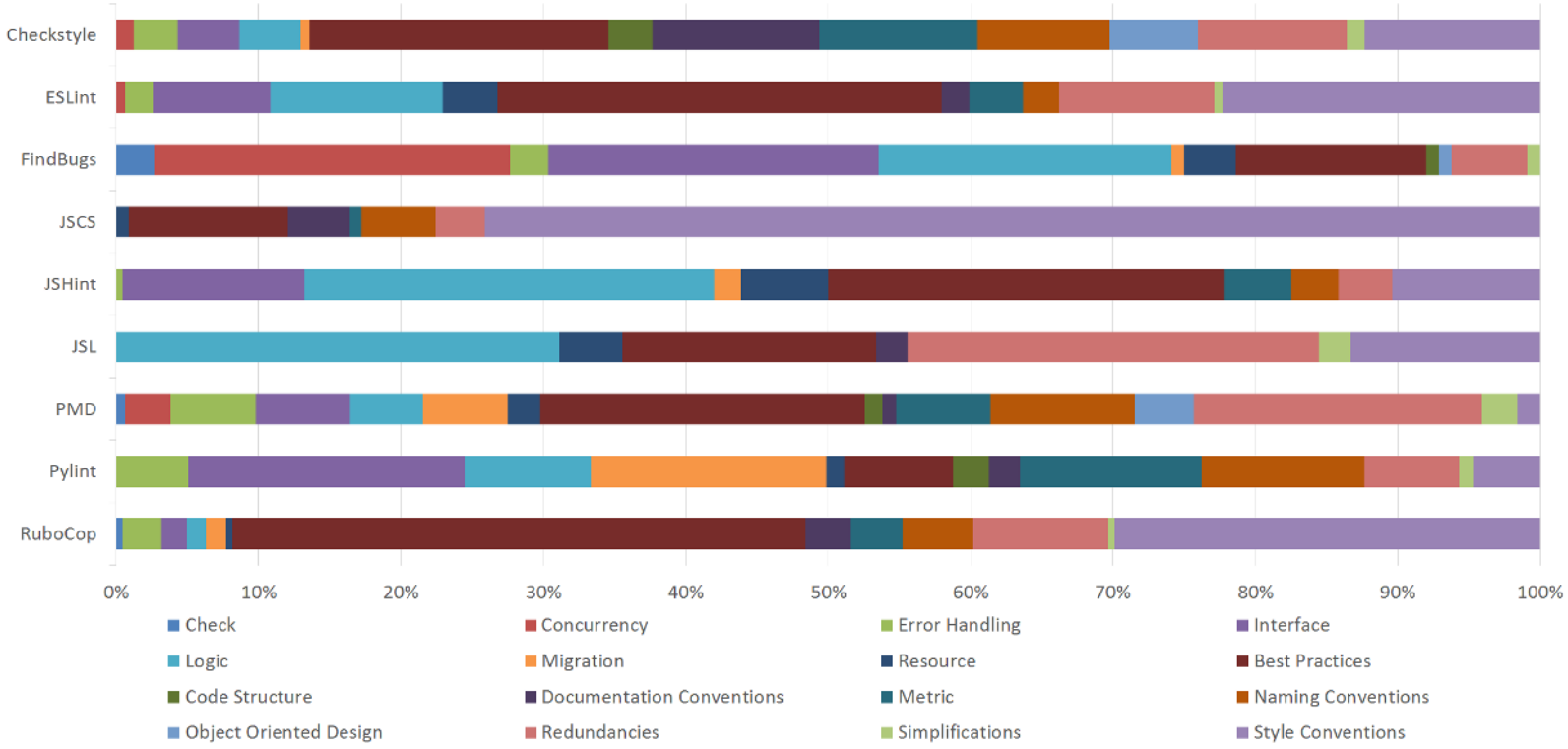

Figure B.1: The distribution of rules per tool, according to our classification.

Category\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop
Check	0	0	2.7	0	0	0	0.6	0	0.5
Concurrency	1.2	0.6	25	0	0	0	3.2	0	0
Error Handling	3.1	1.9	2.7	0	0.5	0	6	5.1	2.7
Interface	4.3	8.3	23.2	0	12.7	0	6.6	19.4	1.8
Logic	4.3	12.1	20.5	0	28.8	31.1	5.1	8.9	1.4
Migration	0.6	0	0.9	0	1.9	0	6	16.5	1.4
Resource	0	3.8	3.6	0.9	6.1	4.4	2.2	1.3	0.5
Best Practices	21	31.2	13.4	11.2	27.8	17.8	22.8	7.6	40.3
Code Structure	3.1	0	0.9	0	0	0	1.3	2.5	0
Documentation Conventions	11.7	1.9	0	4.3	0	2.2	0.9	2.2	3.2
Metric	11.1	3.8	0	0.9	4.7	0	6.6	12.7	3.6
Naming Conventions	9.3	2.5	0	5.2	3.3	0	10.1	11.4	5
Object Oriented Design	6.2	0	0.9	0	0	0	4.1	0	0
Redundancies	10.5	10.8	5.4	3.4	3.8	28.9	20.3	6.7	9.5
Simplifications	1.2	0.6	0.9	0	0	2.2	2.5	1	0.5
Style Conventions	12.3	22.3	0	74.1	10.4	13.3	1.6	4.8	29.9

Table B.16: The distribution of rules per tool, according to our classification. All numbers are percentages and all columns sum up to 100 (approximately, due to rounding).

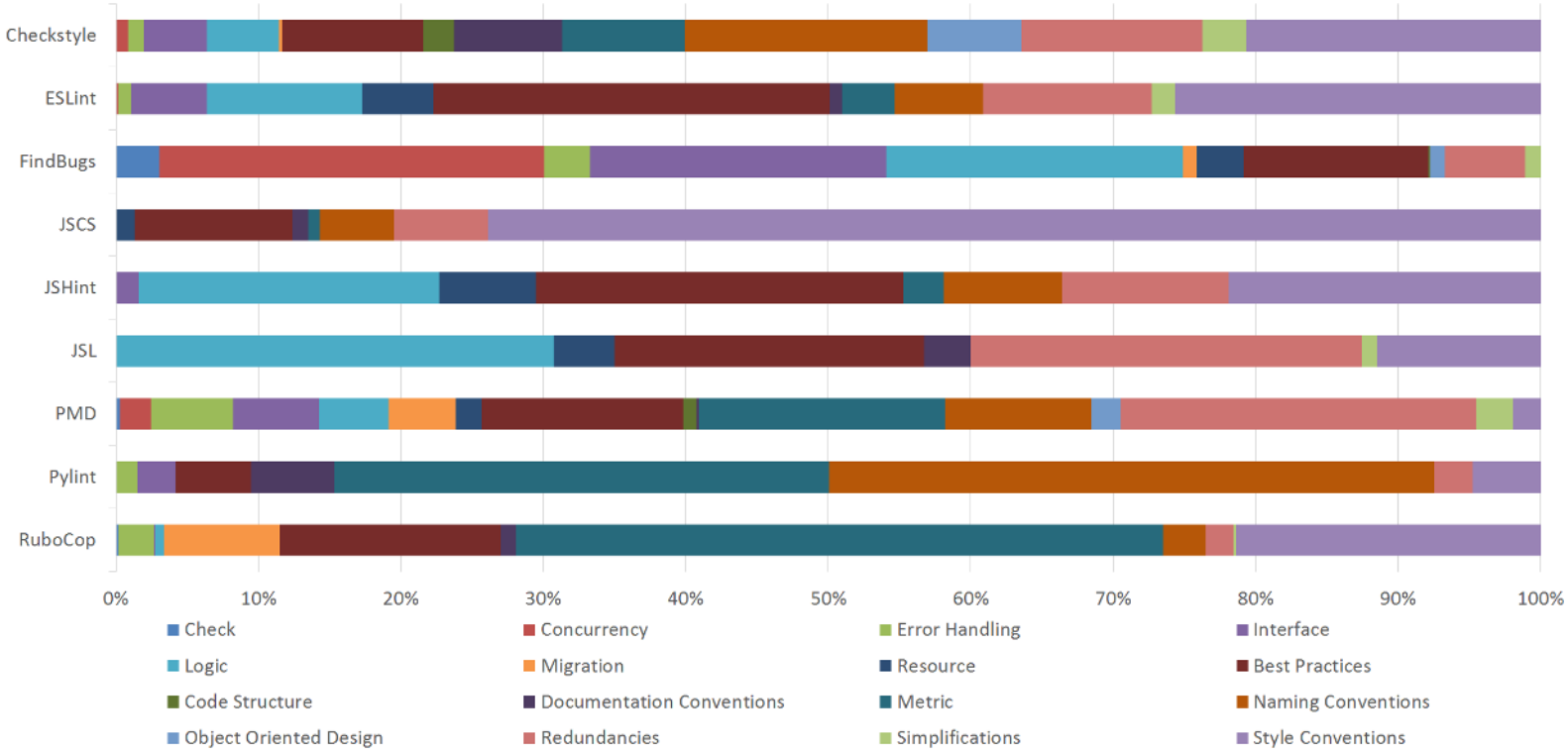

Figure B.2: The distribution of rules that are enabled by developers per tool, according to our classification.

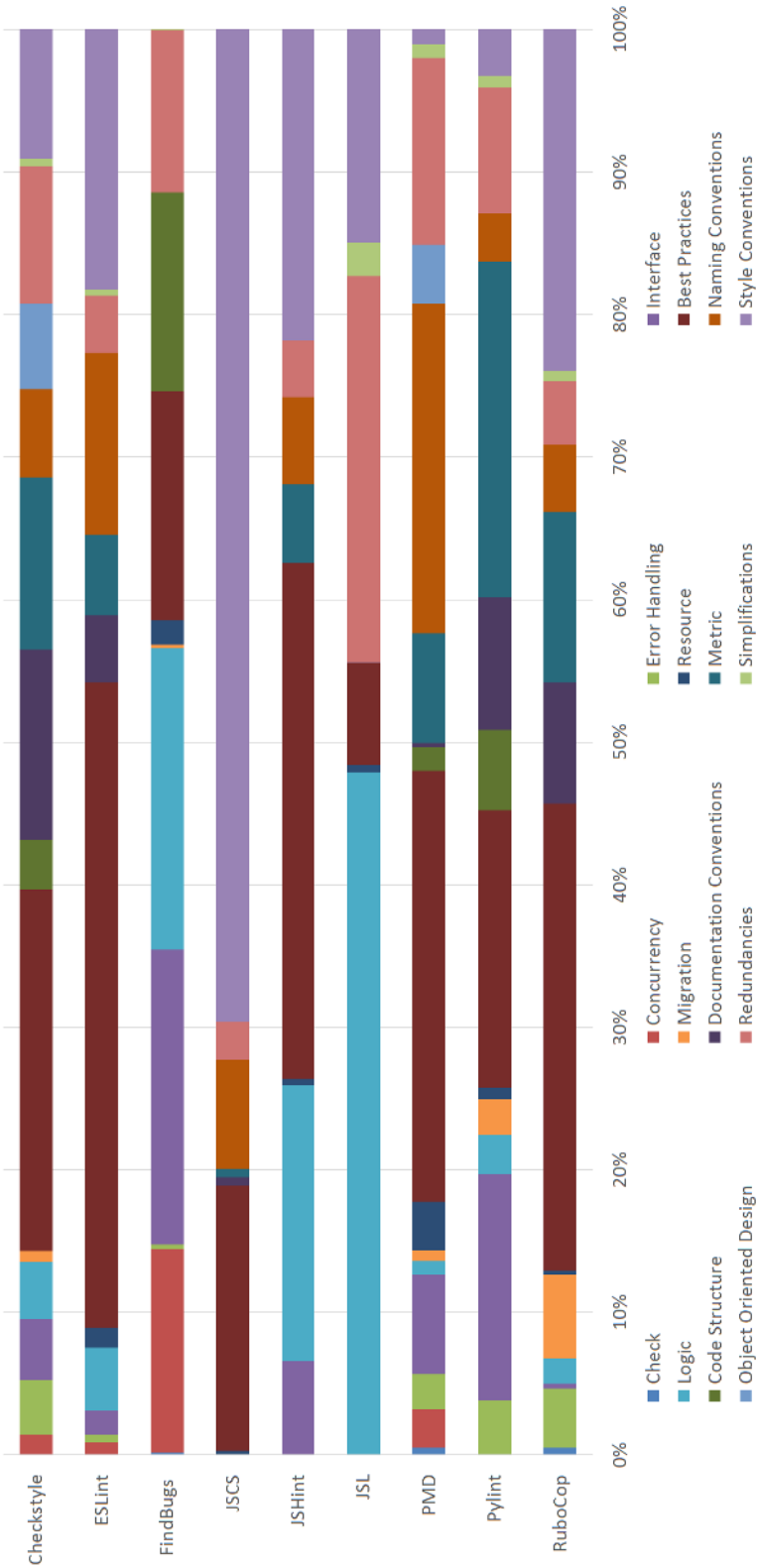

Figure B.3: The distribution of rules that are disabled by developers per tool, according to our classification.

Category\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop
Check	0	0	3	0	0	0	0.3	0	0.2
Concurrency	0.9	0.1	27	0	0	0	2.2	0	0
Error Handling	1.1	0.9	3.2	0	0	0	5.8	1.5	2.5
Interface	4.4	5.3	20.8	0	1.6	0	6	2.7	0.1
Logic	5.1	10.9	20.9	0	21.1	30.7	4.9	0	0.6
Migration	0.2	0	1	0	0	0	4.7	0	8.1
Resource	0	5	3.3	1.3	6.8	4.2	1.8	0	0
Best Practices	9.9	27.8	12.9	11.1	25.8	21.8	14.1	5.3	15.5
Code Structure	2.1	0	0.1	0	0	0	1	0	0
Documentation Conventions	7.6	0.8	0	1.1	0	3.3	0.1	5.8	1
Metric	8.6	3.7	0	0.8	2.8	0	17.3	34.7	45.5
Naming Conventions	17.1	6.2	0	5.2	8.4	0	10.3	42.5	2.9
Object Oriented Design	6.6	0	1.1	0	0	0	2.1	0	0
Redundancies	12.7	11.9	5.6	6.6	11.7	27.5	24.9	2.7	2
Simplifications	3	1.7	1.1	0	0	1	2.6	0	0.2
Style Conventions	20.7	25.6	0	73.9	21.9	11.5	1.9	4.8	21.4

Table B.17: The distribution of rules that are enabled by developers per tool, according to our classification. All numbers are percentages and all columns sum up to 100 (approximately, due to rounding).

Category\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop
Check	0	0	0.2	0	0	0	0.5	0	0.5
Concurrency	1.4	0.9	14.3	0	0	0	2.7	0	0
Error Handling	3.9	0.5	0.3	0	0	0	2.5	3.8	4.2
Interface	4.3	1.7	20.7	0	6.6	0	7	15.9	0.3
Logic	4	4.4	21.2	0	19.4	47.9	1	2.8	1.8
Migration	0.8	0	0.2	0	0	0	0.7	2.5	5.9
Resource	0	1.4	1.7	0.3	0.4	0.5	3.4	0.8	0.3
Best Practices	25.3	45.2	16.1	18.6	36.3	7.1	30.3	19.5	32.8
Code Structure	3.5	0	14	0	0	0	1.6	5.6	0
Documentation Conventions	13.4	4.8	0	0.6	0	0	0.3	9.3	8.5
Metric	12.1	5.7	0	0.6	5.5	0	7.7	23.5	12
Naming Conventions	6.2	12.8	0	7.7	6.1	0	23.1	3.4	4.7
Object Oriented Design	6	0	0	0	0	0	4.1	0	0
Redundancies	9.6	4	11.4	2.7	4	27.1	13.1	8.8	4.4
Simplifications	0.5	0.4	0	0	0	2.3	1	0.8	0.7
Style Conventions	9.1	18.3	0	69.6	21.8	15	1	3.3	23.9

Table B.18: The distribution of rules that are disabled by developers per tool, according to our classification. All numbers are percentages and all columns sum up to 100 (approximately, due to rounding).

Appendix C

Full Results for the Evolution Analysis

This appendix presents the full data underlying the results of Chapter 5. Tables C.1 to C.5 show those results belonging to the Figures 5.3 to 5.7. Additionally, Tables C.6 and C.7 show the distributions of additions per change and deletions per change respectively. These form the basis for Table C.2 and Figure 5.4. Because of space restrictions, Tables C.1 to C.3, C.6, and C.7 do not present all the data that we collected, but rather the most relevant parts.

Days\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop	Total	%
0	12304	3442	1123	9501	85085	725	4692	2527	7173	126572	81.2
1	1285	470	118	976	9801	50	381	513	1076	14670	9.4
2	800	150	235	390	4094	43	193	235	553	6693	4.3
3	366	98	13	140	1493	10	60	116	309	2605	1.7
4	257	60	25	67	933	6	68	74	175	1665	1.1
5	147	19	8	43	526	6	21	81	99	950	0.6
6	109	26	3	27	360	0	15	36	77	653	0.4
7	94	14	4	11	177	0	5	18	58	381	0.2
8	61	8	0	11	186	2	7	16	43	334	0.2
9	37	5	0	14	179	5	2	23	24	289	0.2
10	41	3	0	7	95	0	1	7	22	176	0.1
11	34	2	0	1	48	0	2	11	30	128	0.1
12	27	0	0	5	31	0	1	7	22	93	0.1
13	13	0	0	1	29	1	0	60	11	115	0.1
14	25	0	0	7	18	0	0	32	7	89	0.1
15	8	2	0	1	15	0	1	9	13	49	0
16	9	0	0	6	5	0	1	7	6	34	0
17	10	1	0	3	36	0	0	9	6	65	0
18	12	0	0	8	13	0	1	12	12	58	0
19	9	1	0	2	9	1	1	6	7	36	0
20	8	1	0	0	5	0	0	12	1	27	0
21	8	2	0	0	13	0	3	5	7	38	0
22	3	0	0	0	9	0	1	5	5	23	0
23	6	0	0	4	6	0	0	8	2	26	0
24	0	0	0	0	1	0	1	1	4	7	0
25	4	0	0	0	0	0	0	2	3	9	0
26	8	0	0	0	0	0	0	5	4	17	0
27	4	0	0	0	1	1	0	0	3	9	0
28	4	1	0	0	2	0	0	2	1	10	0
29	0	0	0	1	0	0	0	1	0	2	0
30	0	0	0	0	2	0	0	3	2	7	0
31	0	0	0	0	2	0	0	0	0	2	0
32	4	0	0	0	2	0	0	1	0	7	0
33	0	0	0	0	2	0	0	0	0	2	0
34	6	0	0	0	1	0	0	0	2	9	0
35	0	0	0	0	0	0	0	0	4	4	0
36	3	0	0	0	0	0	0	0	0	3	0
37	1	0	0	0	0	0	0	0	2	3	0
38	0	0	0	0	0	0	0	1	1	2	0
39	1	0	0	0	0	0	0	0	1	2	0
40	2	0	0	0	0	0	0	0	0	2	0
41	2	0	0	0	0	0	0	0	0	2	0
42	0	0	0	0	0	0	0	0	0	0	0
43	1	0	0	0	0	0	0	0	0	1	0
44	0	0	0	0	0	0	0	0	0	0	0
45	0	0	0	0	1	0	0	0	0	1	0
46	0	0	0	0	0	0	0	0	0	0	0
47	1	0	0	0	0	0	0	0	1	2	0
48	0	0	0	0	0	0	0	0	2	2	0
49	0	1	0	0	0	0	0	0	0	1	0
50	4	0	0	0	0	0	0	2	0	6	0
51	0	0	0	0	0	0	0	0	1	1	0
52	0	0	0	0	0	0	0	0	0	0	0
53	1	0	0	0	0	0	0	0	3	4	0
54	0	0	0	0	0	0	0	0	0	0	0
55	0	0	0	0	0	0	0	0	0	0	0
56	0	0	0	0	0	0	0	0	0	0	0
57	0	0	0	0	0	0	0	0	0	0	0
58	0	0	0	0	0	0	0	1	0	1	0
59	0	0	0	0	0	0	0	0	2	2	0
60	0	0	0	0	0	0	0	0	0	0	0
61	0	0	0	0	0	0	0	0	0	0	0
62	0	0	0	0	0	0	0	0	1	1	0
63	0	0	0	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0	0	0	0
65	0	0	0	0	0	0	0	0	0	0	0
Total	15722	4307	1529	11227	103180	850	5457	3849	9786	155907	100

Table C.1: The number of times a configuration file was changed. Limited to 65 changes because of size constraints. The full results go until 248 days. The totals in the last row are accumulated over all the results, not just the first 65.

Changes\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop	Total	%
-32	4	0	3	10	20	0	4	1	5	47	0.1
-31	8	0	0	2	16	0	2	1	3	32	0
-30	9	1	0	5	12	1	0	0	2	30	0
-29	2	0	0	2	56	0	1	1	3	65	0.1
-28	20	1	7	3	35	0	0	0	6	72	0.1
-27	3	1	0	8	30	0	2	1	8	53	0.1
-26	3	0	0	4	38	0	1	0	5	51	0.1
-25	9	2	0	3	31	0	2	3	7	57	0.1
-24	5	2	0	2	65	0	0	2	14	90	0.1
-23	2	1	0	5	137	0	0	0	9	154	0.2
-22	8	2	0	5	64	0	1	0	5	85	0.1
-21	5	1	0	2	70	0	2	0	11	91	0.1
-20	6	0	0	0	52	0	1	0	8	67	0.1
-19	6	2	0	1	42	0	1	0	7	59	0.1
-18	15	5	0	4	40	0	3	0	10	77	0.1
-17	7	2	0	8	73	0	0	1	10	101	0.1
-16	10	0	0	6	28	0	4	0	23	71	0.1
-15	10	4	0	16	134	0	0	3	20	187	0.3
-14	17	3	0	7	84	0	1	4	9	125	0.2
-13	20	1	0	7	70	0	2	2	9	111	0.1
-12	9	2	0	9	61	0	1	1	28	111	0.1
-11	10	1	0	7	64	0	2	2	47	133	0.2
-10	30	3	0	17	57	2	1	0	54	164	0.2
-9	39	3	0	12	72	0	2	3	49	180	0.2
-8	35	5	0	69	126	0	2	1	37	275	0.4
-7	92	5	0	8	159	0	19	6	34	323	0.4
-6	56	9	0	30	238	0	7	19	426	785	1.1
-5	107	8	0	30	210	1	6	44	693	1099	1.5
-4	135	21	0	45	278	0	11	11	392	893	1.2
-3	204	18	0	38	361	0	9	57	326	1013	1.4
-2	243	39	1	108	701	1	30	43	194	1360	1.8
-1	752	114	12	469	3700	12	66	658	234	6017	8.1
0	3363	492	486	833	10581	82	540	2609	1999	20985	28.2
1	1022	741	42	829	10671	140	192	892	799	15328	20.6
2	974	143	19	177	2761	34	94	321	756	5279	7.1
3	597	100	25	155	1262	3	76	308	1224	3750	5
4	425	66	5	58	883	6	43	276	658	2420	3.3
5	285	35	6	65	498	3	30	82	289	1293	1.7
6	263	37	4	44	375	1	25	146	246	1141	1.5
7	149	15	8	38	401	3	18	60	108	800	1.1
8	146	4	18	36	265	3	23	26	100	621	0.8
9	75	8	3	14	225	2	21	71	97	516	0.7
10	104	9	8	14	165	2	8	81	56	447	0.6
11	70	12	3	32	134	0	20	10	40	321	0.4
12	81	5	1	10	146	0	3	5	69	320	0.4
13	54	8	6	12	110	1	12	9	27	239	0.3
14	35	10	0	7	95	1	7	11	27	193	0.3
15	51	9	1	16	132	0	6	5	39	259	0.3
16	76	7	5	50	75	0	8	7	28	256	0.3
17	59	14	0	5	89	1	7	3	16	194	0.3
18	37	4	0	4	72	0	17	2	20	156	0.2
19	23	18	2	3	57	1	4	3	13	124	0.2
20	53	6	1	3	76	0	10	2	26	177	0.2
21	12	3	0	5	55	0	6	1	13	95	0.1
22	10	6	0	2	87	0	2	20	17	144	0.2
23	11	0	8	4	64	0	0	1	13	101	0.1
24	20	8	1	2	68	0	2	3	16	120	0.2
25	13	2	0	3	79	0	6	2	13	118	0.2
26	8	3	0	3	40	0	0	8	14	76	0.1
27	8	3	0	4	42	0	7	0	4	68	0.1
28	16	1	0	0	44	0	1	0	12	74	0.1
29	14	4	0	1	20	0	4	1	6	50	0.1
30	5	4	4	0	17	0	4	1	9	44	0.1
31	12	1	1	2	29	0	1	1	5	52	0.1
32	19	0	0	1	44	0	10	4	10	88	0.1
Total	11712	2191	764	3724	38317	310	1479	6019	9796	74312	100

Table C.2: The size of the change, defined as additions minus deletions. Limited to the range of 32 deletions to 32 additions because of size constraints. The full results go from 1126 deletions to 2055 additions. The totals in the last row are accumulated over all the results, not just over what is shown in the table.

Days\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop	Total	%
0	1715	382	11	718	7761	25	316	596	2603	14127	18
1	395	115	32	322	1775	6	81	217	390	3333	4.3
2	269	75	5	110	1138	7	47	167	212	2030	2.6
3	224	53	12	81	970	4	18	169	260	1791	2.3
4	174	40	5	62	755	1	23	99	160	1319	1.7
5	180	54	2	37	771	3	18	103	232	1400	1.8
6	172	50	1	40	645	4	12	44	138	1106	1.4
7	179	47	7	41	628	4	22	90	82	1100	1.4
8	130	28	2	105	613	1	7	38	124	1048	1.3
9	113	21	3	46	478	4	10	26	76	777	1
10	116	22	3	53	349	1	23	56	93	716	0.9
11	104	29	24	36	320	2	19	23	98	655	0.8
12	98	16	2	33	363	1	9	90	128	740	0.9
13	142	24	3	26	396	3	15	133	57	799	1
14	101	34	1	19	305	2	34	7	93	596	0.8
15	130	20	6	32	295	2	13	16	66	580	0.7
16	84	21	4	19	252	7	4	17	42	450	0.6
17	67	22	0	15	452	0	6	20	50	632	0.8
18	64	21	0	45	264	0	8	15	30	447	0.6
19	51	18	0	76	234	0	4	18	55	456	0.6
20	149	26	1	22	271	0	4	29	73	575	0.7
21	90	15	8	18	261	0	5	118	54	569	0.7
22	59	8	1	20	236	0	5	27	33	389	0.5
23	56	20	2	15	231	10	6	7	44	391	0.5
24	66	11	1	11	201	0	6	18	49	363	0.5
25	52	17	3	14	168	2	2	15	29	302	0.4
26	53	18	0	17	179	2	3	14	39	325	0.4
27	63	10	2	13	199	1	10	17	39	354	0.5
28	38	26	1	20	246	1	6	16	50	404	0.5
29	40	15	0	47	192	0	1	82	70	447	0.6
30	83	4	0	12	158	1	6	7	57	328	0.4
31	39	8	0	17	158	0	0	50	57	329	0.4
32	28	13	1	19	167	1	4	15	32	280	0.4
33	62	11	12	15	215	4	9	9	50	387	0.5
34	51	9	2	36	172	0	6	12	55	343	0.4
35	36	7	1	28	183	8	2	9	48	322	0.4
36	57	11	1	8	114	11	5	14	24	245	0.3
37	28	8	1	11	176	0	10	18	30	282	0.4
38	52	9	1	13	148	0	5	16	25	269	0.3
39	59	3	0	9	131	1	2	12	29	246	0.3
40	42	15	1	17	127	4	3	17	23	249	0.3
41	49	10	0	11	175	0	6	14	36	301	0.4
42	36	19	1	20	124	0	8	32	22	262	0.3
43	31	4	0	25	103	1	3	49	29	245	0.3
44	43	5	0	9	110	0	4	16	18	205	0.3
45	23	7	0	13	108	0	5	9	22	187	0.2
46	36	1	1	37	118	2	2	16	20	233	0.3
47	45	46	0	11	68	0	0	50	40	260	0.3
48	73	20	0	14	133	0	10	11	40	301	0.4
49	42	6	0	3	102	0	6	6	21	186	0.2
50	32	8	0	14	92	0	5	20	19	190	0.2
51	11	18	0	5	75	0	4	4	28	145	0.2
52	19	8	0	23	128	3	1	2	17	201	0.3
53	16	4	3	41	124	2	2	5	8	205	0.3
54	13	6	0	9	106	0	5	62	33	234	0.3
55	40	7	5	6	91	0	1	16	80	246	0.3
56	22	12	0	16	104	3	1	12	15	185	0.2
57	14	25	0	9	126	0	2	7	25	208	0.3
58	34	16	0	17	85	2	0	7	22	183	0.2
59	15	3	0	12	98	0	1	6	8	143	0.2
60	10	7	0	5	62	0	2	8	16	110	0.1
61	23	4	0	4	93	0	5	9	12	150	0.2
62	18	7	1	20	114	0	0	7	24	191	0.2
63	20	8	0	18	100	2	10	8	19	185	0.2
64	13	3	2	6	85	0	1	5	25	140	0.2
65	32	3	2	6	51	0	6	10	15	125	0.2
Total	12853	2241	763	3908	40165	339	1744	6241	10016	78270	100

Table C.3: The time between the creation of a file and the time when the file was changed. Limited to 65 days because of size constraints. The full results go until 4251 days. The totals in the last row are accumulated over all the results, not just the first 65.

Version	Days Since Previous Version	Changes To Next Version	Changes Per Day	Creations To Next Version	Creations Per Day	Changes Per Day, Per Creation
0.18.0	14	44	4	0	0	4
0.17.1	10	77	8	0	0	8
0.17.0	4	47	12	2	1	12
0.16.2	3	35	12	0	0	12
0.16.1	3	26	9	1	1	9
0.16.0	1	16	16	0	0	16
0.15.1	9	143	16	115	13	2
0.15.0	5	78	16	155	31	1
0.14.1	13	173	14	334	26	1
0.13.0	15	167	12	348	24	1
0.12.0	7	47	7	138	20	1
0.11.0	18	146	9	310	18	1
0.10.2	17	92	6	184	11	1
0.10.1	7	43	7	111	16	1
0.10.0	8	52	7	119	15	1
0.9.2	27	129	5	386	15	1
0.9.1	7	43	7	120	18	1
0.8.2	35	156	5	424	13	1
0.8.1	10	42	5	101	11	1
0.8.0	5	13	3	39	8	1
0.7.4	57	207	4	370	7	1
0.7.3	1	3	3	12	12	1
0.7.2	1	1	1	14	14	1
0.6.2	46	159	4	535	12	1
0.6.1	6	22	4	96	16	1
0.5.1	30	38	2	246	9	1
0.5.0	7	15	3	28	4	1
0.4.5	12	6	1	16	2	1
0.4.4	4	2	1	1	1	1
0.4.3	6	9	2	2	1	2
0.4.2	15	13	1	24	2	1
0.4.1	5	6	2	4	1	2
0.4.0	15	17	2	12	1	2
0.3.0	22	26	2	13	1	2
0.2.0	20	3	1	9	1	1
0.1.4	26	10	1	16	1	1
0.1.3	10	10	1	4	1	1
0.1.2	2	0	0	0	0	0
0.1.1	14	3	1	5	1	1
0.1.0	6	5	1	4	1	1
0.0.9	29	12	1	6	1	1
0.0.7	75	67	1	2	1	1
0.0.6	6	20	4	0	0	4
0.0.5	11	18	2	1	1	2
0.0.4	2	0	0	0	0	0
0.0.3	3	0	0	0	0	0
0.0.2	1	0	0	0	0	0

Table C.4: The number of changes per day per creation, for each version of ESLint. All fractions rounded to the next integer.

Version	Days Since Previous Version	Changes To Next Version	Changes Per Day	Creations To Next Version	Creations Per Day	Changes Per Day, Per Creation
1.4.3	19	46	3	7	1	3
1.4.2	3	3	1	0	0	1
1.4.1	54	275	6	281	6	1
1.4.0	54	411	8	244	5	2
1.3.1	91	746	9	388	5	2
1.3.0	29	87	3	99	4	1
1.2.0	95	349	4	327	4	1
1.1.0	120	457	4	352	3	2
1.0.0	139	443	4	316	3	2
0.28.0	103	619	7	231	3	3
0.27.0	58	94	2	106	2	1
0.26.0	144	286	2	243	2	1
0.25.2	80	171	3	78	1	3
0.25.1	222	308	2	409	2	1
0.25.0	62	215	4	79	2	2
0.24.0	79	68	1	80	2	1
0.23.0	190	295	2	123	1	2
0.22.0	57	84	2	21	1	2
0.21.4	19	61	4	18	1	4
0.21.3	29	64	3	16	1	3
0.21.2	33	90	3	16	1	3
0.21.1	83	507	7	87	2	4
0.21.0	24	6	1	27	2	1
0.20.0	49	46	1	19	1	1
0.19.0	95	135	2	52	1	2
0.18.1	113	50	1	25	1	1
0.18.0	155	85	1	44	1	1
0.17.0	6	0	0	0	0	0
0.16.0	50	11	1	11	1	1
0.15.2	107	11	1	17	1	1
0.15.1	28	32	2	4	1	2
0.15.0	5	2	1	0	0	1
0.14.0	240	56	1	21	1	1
0.13.2	221	43	1	17	1	1
0.13.1	97	9	1	20	1	1
0.13.0	2	0	0	0	0	0
0.12.2	97	16	1	3	1	1
0.12.1	59	0	0	14	1	0

Table C.5: The number of changes per day per creation, for each version of Pylint. All fractions rounded to the next integer.

Additions\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop	Total	%
0	1932	218	54	764	4803	15	199	735	2095	10815	14.6
1	2751	779	33	1092	11791	169	428	2226	2078	21347	28.7
2	938	393	82	547	7812	58	132	1065	1076	12103	16.3
3	1029	206	197	274	2526	14	90	487	1455	6278	8.4
4	624	100	11	164	1419	10	74	339	798	3539	4.8
5	490	60	44	99	1542	19	77	277	398	3006	4
6	402	47	13	55	695	1	47	152	400	1812	2.4
7	251	31	149	45	552	5	24	113	198	1368	1.8
8	217	20	13	33	539	4	27	31	175	1059	1.4
9	136	22	8	56	342	2	26	138	130	860	1.2
10	107	10	6	29	317	0	23	86	97	675	0.9
11	105	10	24	21	246	2	20	16	87	531	0.7
12	107	12	2	17	237	0	6	14	79	474	0.6
13	70	13	0	10	221	1	19	18	57	409	0.6
14	64	14	4	11	326	1	16	10	45	491	0.7
15	113	10	8	16	189	0	14	11	45	406	0.5
16	96	8	4	32	192	1	20	5	36	394	0.5
17	73	12	5	11	268	0	18	4	27	418	0.6
18	65	8	0	9	188	1	23	8	35	337	0.5
19	32	3	0	33	168	0	6	3	49	294	0.4
20	64	7	0	5	146	0	33	3	29	287	0.4
21	34	2	0	7	152	0	9	4	25	233	0.3
22	30	7	1	8	241	0	5	39	22	353	0.5
23	25	4	0	4	155	0	4	2	14	208	0.3
24	21	8	1	8	134	0	5	2	15	194	0.3
25	23	24	4	6	102	0	2	3	19	183	0.2
26	13	4	0	5	144	0	6	5	14	191	0.3
27	13	7	1	10	93	0	5	8	11	148	0.2
28	19	2	1	9	127	1	4	3	18	184	0.2
29	21	4	0	4	65	0	5	2	13	114	0.2
30	10	3	0	1	65	0	5	1	16	101	0.1
31	17	1	0	3	54	0	2	2	7	86	0.1
32	22	2	4	7	83	0	10	5	10	143	0.2
33	6	45	0	2	69	0	5	2	9	138	0.2
34	25	5	0	62	49	0	6	1	3	151	0.2
35	7	7	0	3	66	0	1	1	8	93	0.1
36	13	6	1	22	42	0	3	0	3	90	0.1
37	13	1	4	1	45	0	2	0	7	73	0.1
38	9	0	1	11	32	0	1	3	10	67	0.1
39	12	4	0	2	759	0	2	0	8	787	1.1
40	8	1	0	2	33	0	4	0	3	51	0.1
41	3	5	0	4	41	0	1	0	2	56	0.1
42	9	2	0	24	36	0	2	2	6	81	0.1
43	72	0	0	2	45	0	3	7	1	130	0.2
44	13	0	0	1	32	0	10	1	2	59	0.1
45	6	2	0	9	24	0	2	0	4	47	0.1
46	5	1	0	0	10	0	1	0	7	24	0
47	7	1	0	2	8	0	1	0	5	24	0
48	5	0	1	0	44	0	0	4	2	56	0.1
49	2	2	2	7	20	0	0	0	2	35	0
50	11	0	0	3	24	0	1	0	2	41	0.1
51	11	0	0	2	4	0	0	0	2	19	0
52	6	1	0	1	15	0	1	0	7	31	0
53	7	4	0	3	30	0	0	0	2	46	0.1
54	5	1	0	1	17	0	1	14	0	39	0.1
55	3	1	0	3	18	0	0	0	1	26	0
56	2	0	0	0	8	0	0	0	5	15	0
57	3	0	0	2	13	0	0	0	2	20	0
58	12	1	0	3	15	0	0	0	8	39	0.1
59	5	2	0	5	20	0	1	1	1	35	0
60	3	2	2	3	9	0	0	0	0	19	0
61	3	1	0	3	19	1	0	0	1	28	0
62	4	1	0	1	17	0	1	1	3	28	0
63	7	2	0	1	13	0	1	3	0	27	0
64	4	1	0	3	3	0	0	0	5	16	0
65	3	0	0	7	21	0	0	0	4	35	0
Total	11712	2191	764	3724	38317	310	1479	6019	9796	74312	100

Table C.6: The amount of additions per change. Limited to 65 additions due to size constraints. The full results go up to 2058 additions. The totals in the last row are accumulated over all the results, not just over what is shown in the table.

Deletions\Tool	Checkstyle	ESLint	FindBugs	JSCS	JSHint	JSL	PMD	Pylint	RuboCop	Total	%
0	2832	644	70	1066	8251	172	514	781	3412	17742	23.9
1	3748	811	66	1213	16623	67	381	3668	1935	28512	38.4
2	966	241	118	333	3527	23	154	685	734	6781	9.1
3	633	86	219	178	1173	19	80	191	801	3380	4.5
4	426	50	18	158	742	8	45	104	506	2057	2.8
5	271	25	21	51	1065	7	30	223	714	2407	3.2
6	207	50	13	46	402	2	24	42	532	1318	1.8
7	149	16	147	34	338	0	20	17	162	883	1.2
8	106	20	2	29	352	0	11	17	117	654	0.9
9	106	10	0	17	231	1	15	13	75	468	0.6
10	97	11	0	12	270	0	6	7	94	497	0.7
11	68	17	0	21	181	2	10	12	84	395	0.5
12	64	13	3	13	251	0	7	7	56	414	0.6
13	64	7	0	22	206	0	6	4	30	339	0.5
14	36	7	1	18	326	0	9	7	30	434	0.6
15	44	11	0	18	179	0	3	31	21	307	0.4
16	43	7	0	18	174	0	20	32	44	338	0.5
17	33	3	1	11	238	0	12	3	15	316	0.4
18	33	11	1	5	126	0	2	2	19	199	0.3
19	13	2	0	22	207	0	1	1	20	266	0.4
20	26	3	0	12	195	0	6	3	13	258	0.3
21	24	2	0	4	139	0	2	3	15	189	0.3
22	13	5	0	9	252	0	3	0	13	295	0.4
23	20	1	0	4	245	0	5	0	14	289	0.4
24	11	4	0	6	117	0	2	4	11	155	0.2
25	20	3	0	9	79	0	4	4	10	129	0.2
26	7	4	2	6	94	0	3	0	10	126	0.2
27	15	5	0	14	53	0	21	5	7	120	0.2
28	14	3	0	11	96	0	1	0	10	135	0.2
29	3	0	0	4	48	0	3	1	3	62	0.1
30	8	4	0	12	96	0	1	1	4	126	0.2
31	12	2	0	28	39	0	4	1	2	88	0.1
32	12	0	0	5	46	0	4	0	12	79	0.1
33	17	41	0	9	63	0	5	1	9	145	0.2
34	13	1	0	67	31	0	0	1	6	119	0.2
35	2	8	0	14	48	0	1	0	6	79	0.1
36	20	5	0	34	31	0	1	1	1	93	0.1
37	14	3	0	3	33	2	0	1	8	64	0.1
38	6	1	0	6	29	0	1	2	10	55	0.1
39	7	2	0	2	373	0	1	0	7	392	0.5
40	7	0	0	1	556	0	1	3	1	569	0.8
41	7	3	0	1	28	0	1	1	4	45	0.1
42	0	1	0	1	21	0	2	1	5	31	0
43	66	0	7	3	18	0	0	0	6	100	0.1
44	7	0	3	2	11	0	3	1	4	31	0
45	6	1	0	3	14	0	1	0	6	31	0
46	4	2	0	2	14	0	0	0	1	23	0
47	2	0	0	1	21	0	0	2	1	27	0
48	2	0	0	0	11	0	0	0	2	15	0
49	2	0	0	5	8	0	0	0	2	17	0
50	4	1	0	1	11	0	0	1	4	22	0
51	7	1	0	1	6	0	0	0	5	20	0
52	0	3	0	1	8	0	0	1	4	17	0
53	1	4	0	0	9	0	1	0	3	18	0
54	3	1	0	2	9	0	0	0	3	18	0
55	4	0	0	2	35	0	0	1	4	46	0.1
56	3	0	0	0	3	0	1	0	4	11	0
57	4	0	0	0	7	0	1	1	1	14	0
58	9	1	0	4	16	0	0	0	5	35	0
59	3	3	0	2	20	0	0	0	4	32	0
60	5	0	0	2	13	0	0	0	2	22	0
61	0	0	0	1	6	0	0	0	5	12	0
62	2	1	0	4	14	0	0	2	3	26	0
63	1	0	0	4	34	0	1	1	4	45	0.1
64	0	2	0	3	16	0	2	2	4	29	0
65	1	0	0	2	21	0	1	0	1	26	0
Total	11712	2191	764	3724	38317	310	1479	6019	9796	74312	100

Table C.7: The amount of deletions per change. Limited to 65 deletions due to size constraints. The full results go up to 1250 deletions. The totals in the last row are accumulated over all the results, not just over what is shown in the table.