What the Fix? A Study of ASATs Rule Documentation

Anonymous Author(s)

ABSTRACT

Automatic Static Analysis Tools (ASATs) are widely used by software developers to diffuse and enforce coding practices. Yet, we know little about the documentation of ASATs, despite it being critical to learn about the coding practices in the first place. We shed light on this through a two-phase study. First, we analyze the documentation of more than 100 rules of 16 ASATs for multiple programming languages, and distill a taxonomy of the goals of the documentation and its types of contents. Then, we conduct a survey to assess the effectiveness of the documentation in terms of its goals and types of contents. We highlight opportunities for improvement in ASAT documentation.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Software maintenance tools**; *Software design engineering*.

KEYWORDS

software quality, automatic static analysis tools, linters, documentation

ACM Reference Format:

Anonymous Author(s). 2024. What the Fix? A Study of ASATs Rule Documentation. In *Proceedings of 46th International Conference on Program Comprehension (ICPC 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXX.XXX

1 INTRODUCTION

Automatic Static Analysis Tools (ASATs, sometimes called linters) [1] are very popular quality insurance tools used in a quarter [2] to half of software projects [3]. According to the https://analysis-tools.dev website, there are more than 600 ASATs. The core principle behind these tools is to scan a code base looking for evidence of potential issues with the source code, such as not following the best practices, using error-prone constructs, and even security or performance problems [1, 4–7]. When such an issue is detected, a warning is issued to the developers of the project so that they can inspect the incriminated code and fix the issue if necessary. Issues are generally defined by *rules*, an infamous example being the `eqeqeq` rule in ESLint JavaScript's ASAT that advises developers to use `===` instead of `==` to avoid type coercion errors.

Even though these tools are prone to issue false-positive warnings [8–10] and their effect on software quality is still a matter of debate [11–13], one positive aspect raised by the use of ASATs is their ability to improve the developer's knowledge and skills [14, 15]. Following this idea, our industrial partner created a fully customizable ASAT aimed at centralizing and diffusing the knowledge of development teams, notably for the onboarding of new developers. In this article, we focus on the following scenario of ASATs: a developer comes across a warning for the first time and wants to learn more about it. Usually, warnings raised by ASATs within the IDE or within the terminal as a result of the use of its command line interface only contain limited information [7, 16]. To further explain a given rule, ASATs generally provide more extensive documentation via formatted documents available online.

We argue that this documentation is of crucial importance in order to improve the knowledge and skills of developers as well as increase their engagement in avoiding breaking the rule. Even though warning notifications have already been studied [7, 16], there is, to the best of our knowledge, no comprehensive study about the reference documentation of ASATs rules. Moreover, existing empirical evidence shows that existing ASAT rules documentations are low quality [15–17]. We believe that reference documentation of ASATs rules is not a classical piece of software documentation, such as an API documentation [18]. Indeed, ASATs rule documentation is challenging as it needs to explain issues regardless of their particular context in such a manner that developers are able later to understand them in their own context. Another difficult point is that ASATs rules are sometimes subjective such as the infamous use of goto [19] or the classical dilemma of choosing between snake and camel case [20]. Thus, little is known about how to best document ASAT rules. This lack of knowledge is damaging when ASATs such as Semgrep allow developers around the world to effortlessly create and share hundreds of rules.

In this article, we explore the current state of the ASATs' rules documentation and contrast it with developers' expectations, in order to extract best practices with the goal to increase the quality of this documentation, via a two-phase study. We start with an empirical study of how real-world ASAT rules are documented. Section 2 details our study of more than 100 rules across 16 ASATs, covering 7 programming languages (plus two polyglot ASATs). Through an iterative analysis of these rules, we derive a nomenclature covering 15 documentation attributes in three themes. In Section 3 we distill our nomenclature in a simple taxonomy of documentation purposes (*What* triggers the rule, *Why* it is important, and how to *Fix* the issue) and content types (*Text*, *Code*, and *Hyperlinks*). Only half of the rules we analyze have a *Why* purpose.

In the second step, we validate our taxonomy and use it to contrast the documentation of 12 real-world rules with developer expectations via a survey. Section 4 presents the survey in which 85 respondents evaluated the rules 298 times. Among other findings, developers highlight quality issues with the documentation of the *Why* purpose, and emphasize the pedagogical aspects and the need for conciseness in ASAT documentation. To close the paper, we discuss the limitations of this study (Section 5), other studies of ASATs and documentation (Section 6), before concluding (Section 7).

ICPC 2024, April 2024, Lisbon, Portugal

https://doi.org/XXX.XXX

2 A NOMENCLATURE FOR RULE DOCUMENTATION

The first step of our study is to create a nomenclature based on the documentation of the rules provided by ASATs. A nomenclature is a classification tool that provides a structured and systematic way of naming and referring to a wide range of objects. This nomenclature will help us define the types of information we find in a rule's documentation and group them by purpose. To build it, we use a three-step process described next: (1) we select a total of 16 diverse ASATs that provide documentation for their rules; (2) we iteratively build a corpus of 119 rules, for which we code the documentation concepts we encounter to support their comparison; and (3) we compare and analyze the rule documentations to provide the expected nomenclature, which covers 15 different concepts.

2.1 Selecting ASATs

A GitHub project listing ASATs[1] references over 600 different tools; choosing a reasonable subset is both necessary and challenging. A first hard requirement is that we only include ASATs focusing on code rule enforcement, leaving out for instance code beautifiers. A second hard requirement is that we only include ASATs that provide a website presenting the rules documentation. Finally, we select ASATs emphasizing both *diversity* and *popularity*. Since we want our nomenclature to be used beyond our study, regardless of ASAT, we include a diverse set of ASATs targetting different programming languages. Finally, we want to bias our corpus toward popular ASATs as we postulate that popular ASATs have more odds of including well-thought-out and comprehensive documentation.

We define two criteria to judge diversity and one criterion to judge popularity. The two diversity criteria we propose are: that the ASATs covers most of the popular programming languages[2], and to have at least two ASATs per programming language covered; we also include ASAT that supports multiple languages to account for this category. The popularity criterion we propose is that ASATs must have a GitHub repository with at least 1000 stars.

Starting with the ASAT list, we select the 16 ASATs in Table 1 using our criteria and the domain expertise of our industrial partner. The only exception to the popularity criterion is Gendarme, which we pick to increase diversity: it is the official ASAT for the alternative Mono C# implementation. We are aware that this selection is somewhat arbitrary and that another selection could have been made with the same criteria; we discuss this issue in Section 5.

2.2 Coding Documentation Concepts

Our selection of ASATs offer rule documentation that differs in structure and in content, and this can even happen between two rules from the same ASAT. To compare ASAT documentations, we identify and align all the concepts used in the documentation.

For example, Figure 1 displays the documentation for the rule *pointless-statement* provided by `Pylint` . Using visual and graphical cues on this documentation (headings, line breaks, boxes, images, etc.), we extract and code four concepts: an *emitted message* (error message when the rule is violated), a *description*, a *problematic code*, and a *correct code*. If we now want to compare this documentation with another rule, we need to align their documentation concepts. This alignment will, e.g., reveal which concepts are present in both documentations, or which ones are present in only one rule.

[1]https://github.com/analysis-tools-dev/static-analysis, 12K stars
[2]https://madnight.github.io/githut/#/pull_requests/2021/4

Table 1: Languages and ASAT selected

Languages	ASATs	Languages	ASATs
C / C++	OCLint Cppcheck	PHP	PHP CS Fixer Psalm
C#	Gendarme Roslynator	Python	Pylint Flake8
Java	Checkstyle SpotBugs	Ruby	RuboCop Brakeman
JS / TS	ESLint RSLint	Multi	Semgrep SonarLint

pointless-statement / W0104

Message emitted:

`Statement seems to have no effect`

Description:

Used when a statement doesn't have (or at least seems to) any effect.

Problematic code:

```
[1, 2, 3]  # [pointless-statement]
```

Correct code:

```
NUMBERS = [1, 2, 3]

print(NUMBERS)
```

Figure 1: Documentation of the rule *pointless-statement* from `Pylint`

Each ASATs provides many rules. We use a two-step process to first code and then reconcile the documentation concepts. More precisely, we build an independent coding for each ASAT iteratively and then reconcile them into a single and global coding for rules documentation. For each ASAT, we pick up rules at random, look at the documentation for visual and graphical cues, and add documentation concepts if they were not identified before. We repeat this until saturation—analyzing 5 successive rules without discovering a new concept. Using this process, we analyze at least 6 rules per ASAT; Table 2, column 3, counts the rules we inspect per ASAT.

The reconciliation of the documentation concepts identified in each ASAT was done manually by the authors during a harmonization session. For instance, we consider that the **Message emitted** concept in `Pylint` rule of Figure 1 is highly similar to the **Message output** concept identified in `Cppcheck` rules. We therefore reconcile these two concepts and code them with **Error Output**.

As a result, we build a corpus of 119 rules, with which we identify 15 documentation concepts across ASATs (see Table 2). Documentation consistency vary: depending on the tool, we need between 6 and 10 rules to reach saturation.

Table 2: Percentage of presence of all elements for each ASAT

			Comprehension				Usage							Metadata			
Language	**ASAT**	**# Rules**	*Code Example*	*Description*	*Further Information*	*When Not To Use It*	*Auto Fix*	*Compatibility*	*Configurations*	*Error Output*	*IDE Fix*	*Since*	*Usage Example*	*Related Rules*	*Rule Definition*	*Rule Set*	*Severity*
C++	OCLint	6	100	100								100			100		
	Cppcheck	6	100	100	83					67				100			100
C#	Gendarme	9	89	100				22	11			33					
	Roslynator	10	100		10				10								100
Java	Checkstyle	6	100	100	67				83	100		100	100			100	
	SpotBugs	6		100	33												
JS	ESLint	10	70	100	40	50	40	20	80		20	100		30	100		
	RSLint	8	100	100					38						100		
PHP	PHP CS Fixer	7	100	100		29	100		29							100	29
	Psalm	7	100	100	14				14								
Python	Pylint	10	100	100	10					100							100
	Flake8	6	100	100	83												
Ruby	RuboCop	9	89	100	56		78		56			100					
	Brakeman	7	71	100	57					43							
Multi	Semgrep	6	100	100	50										100		100
	SonarLint	6	100	100	17												100
	Total	119	104	109	36	7	18	4	26	23	2	34	6	9	30	13	40

2.3 The Nomenclature

After identifying documentation elements in the previous phase, to finalize our nomenclature, we grouped the documentation concepts we encountered into different themes to clarify their role. This classification was made by the authors following an approach similar to open card sorting [21]. We obtained the following three themes:

- **Comprehension**, which contains all the concepts identifying parts of the documentation that help to understand the rule.
- **Usage**, which contains all concepts identifying parts explaining how to correctly configure the rule for a given project.
- **Metadata**, which contains all the concepts identifying ASAT-specific information (such as organizational scheme or rules source code).

Our nomenclature is presented in Table 2 with 15 documentation concepts grouped into three themes. For the sake of readability, we present the results by ASAT, even if there are differences between rules within the same ASAT. For each ASAT and concept Table 2 shows the percentage of the concept's presence across all analyzed rules (the full per-rule version is available in our replication kit[3]).

The first point of interest is that *there is no single documentation concept that can be found in all ASATs*. The two most common concepts are *description* (100% in all but one ASAT, `Roslynator`) and *code example* (70–100% in all but one ASAT, `SpotBugs`). This presence is mirrored in the rules: 109 out of 119 rules have descriptions, and 104 out of 119 have code examples. We were surprised by the lack of descriptions for `Roslynator`, especially since every other ASAT had a description: for this ASAT, the title of the rule acts as a description [4]. Other concepts are sparser, with 40 rules out of 119 (6 ASATs) that include a *severity*, and 36 out of 119 that include *further information* (12 ASATs). Some concepts are very sparse, such as *IDE Fix* (2 rules out of 119) and *compatibility* (4 out of 119).

The second point of interest is that, apart from the description, the code example, and the further information concepts, *few ASATs share the same concepts*. We note that the average number of concepts per ASAT is 5.1, with a median of 4.5. `ESLint` is the exception, with 11 concepts used, albeit with only three concepts used consistently (*description*, *since*, and *rule definition*).

The final point of interest is that few ASATs are consistent in their use of the documentation elements. Only `OCLint` is completely consistent, while `ESLint` is the most inconsistent.

> **Finding #1:** Projecting our nomenclature onto the rules of the 16 ASATs we selected clearly reveals the differences in terms of rule documentation. This reinforces the need to define a more abstract taxonomy and to carry out a survey of developers to better understand their expectations and calls for action, which is the purpose of the following sections.

[3] https://icpc2024-asats.github.io?page=analysis&tab=nomenclature

[4] e.g., https://josefpihrt.github.io/docs/roslynator/analyzers/RCS0033

3 TAXONOMY OF CONTENT PURPOSES AND TYPES

In this section, we dive deeper into the actual content of rules documentation to better understand their purpose. We focus on the nomenclature's **Comprehension** theme as, we are interested in the developer's comprehension of ASAT rules and the best practices they describe. In contrast, the nomenclature's **Usage** theme relates to the usage of the ASAT (.e.g, how to configure a rule), while **Metadata** mainly contains ASAT-specific information (such as categories of rules or source code).

The **Comprehension** theme consists of four terms: *Code Example*, *Description*, *Further Information*, and *When Not To Use It* (Table 2); we focus on this theme in the following. When looking at the data, one quickly realizes that these umbrella terms conceal a rich underlying diversity of purposes and content types. For instance, *Description* content sometimes highlights the rationale for a particular rule; at other times, it describes how to identify code that breaches the rule. Similarly, *Code Example* snippets may present compliant code, non-compliant code, or a mix thereof. The content also uses a combination of text, hyperlinks, and source code.

Section 3.1 presents the methodology we follow to extract and consolidate this information; Section 3.2 describe how we validate the resulting taxonomy internally (the survey in Section 4 provides additional validation). Finally, Section 3.3 presents the results of applying our taxonomy to the rules of Table 2.

3.1 Extraction

To better understand what is the purpose and the types of content used in ASAT rule documentation, we employ an informal open card-sorting [21] methodology coupled with a saturation process similar to the one used in Section 2. The objective is twofold: identify *how* the content is materialized in the documentation (*e.g.*, text, source code, images, *etc.*), and what is the *purpose* of the content.

Traditional card sorting requires printing fragments of the documentation content on cards and physically regrouping them by common themes. Unfortunately, the large amount of content present in our sample (more than a hundred rules) makes this methodology impractical.

To simplify the process, we go over rules incrementally in an arbitrary order, limiting the open card sorting process to the content of these rules relevant to the **Comprehension** theme. We end the process when no new theme emerges after five successive rules.

Two of the authors conducted the open card sorting during a collaborative session, in which they reviewed a dozen rules. At the end of the session, we obtained the following taxonomy:

- **Purpose**
  - *What*: What triggers the activation of this rule and how to recognize violating code?
  - *Why*: Why does this rule matter, and why should it be enforced?
  - *Fix*: How should violating code be fixed to comply with this rule?
- **Content type**
  - *Text*: Free-form prose text
  - *Code*: Source code written in (one of) the programming language targeted by the ASAT, possibly including some prose embedded as comments
  - *Hyperlink*: Hyperlinks to other documentation, web pages, PDFs, *etc.*

Figure 2 shows an example rule from `Checkstyle` with its purposes and content types. Its description employs a mix of *Text* and *Hyperlink* to document the *What* and *Why* purposes. The *Code*, on the other hand, documents the *What* and *Fix* purposes via examples of compliant and non-compliant code.

3.2 Validation

We validate the completeness and objectivity of our taxonomy by calculating the agreement of independently annotating a set of rules. Note that this is an internal validation of our taxonomy; our survey (Section 4) validates it with external respondents.

The first author selects 12 rules from the considered ASATs, making sure to select non-trivial[5] rules covering the main categories of ASAT rules identified by Vassolo et al. [22]: naming and style, correctness, performance, and security. The first author then manually removes (when present) from their documentation the content related to **Usage** and **Metadata**—irrelevant to our taxonomy—keeping only the content related to **Comprehension**. Finally, the first author annotates the rules by highlighting the elements that address the *What*, *Why*, and *Fix* purposes, as well as the content types used, as shown in Figure 2.

The twelve rules are split randomly among three of the remaining authors, who follow the same rating process on four rules each, ensuring that each rule is annotated by two independent raters. We evaluate our taxonomy on these rules by assessing i) to what extent the *What*, *Why*, and *Fix* purposes are necessary and sufficient to rate all the documentation content (completeness), and ii) to what extent independent raters reliably agree on the rating of documentation content (objectivity).

Completeness. All raters used all purposes on all the rules. All raters used at least one purpose on all *Text* content, most *Hyperlinks*, and the majority of *Code* snippet. Some parts of the *Code* snippets were not rated as they do not relate to any purpose but rather serve as boilerplate code to ensure that the snippets are syntactically valid (for instance the class declaration in Figure 2). One rater found that some *Hyperlinks* were too general to be actionable and that it was not clear how they relate to the rule documentation (*e.g.*, a link to a list of the ten most critical vulnerabilities in web applications).[6]

Objectivity. We measure the agreement among raters using Cohen's kappa for the textual content, at the word level with regard to the purposes. (the content type being completely objective). Cohen's kappa is suitable for this situation as there are two raters per word, and each word is rated with a single purpose. For the *Code* and *Hyperlinks*, however, the same link or snippet may be associated with multiple labels. We use the better-suited weighted Fleiss' kappa with a weight computed using the MASI distance between sets of labels [23].

[5] An example of a trivial rule is https://rslint.org/no-await-in-loop/
[6] https://owasp.org/www-project-top-ten/

Table 3: Percentage of presence of each taxonomy purposes regarding the type of content for each ASAT

			Text			Code			Link		
Language	**ASAT**	**# Rules**	*What* (%)	*Why* (%)	*Fix* (%)	*What* (%)	*Why* (%)	*Fix* (%)	*What* (%)	*Why* (%)	*Fix* (%)
C / C++	OCLint	6	100	33		100			100		
	Cppcheck	6	100	83		100		17	83	83	
C#	Gendarme	9	100	67	78	89	22	89			
	Roslynator	10	10			100		90	10		
Java	Checkstyle	6	100	50	50	100		100	100		
	SpotBugs	6	100	83	100				33	17	
JS / TS	ESLint	10	100	80	90	100	10	100	100		
	RSLint	8	100	100	50	100	36	75	100		
PHP	PHP CS Fixer	7	100		57	100		100			
	Psalm	7	100	14	14	100		29			14
Python	Pylint	10	100	10	40	80		80	100		
	Flake8	6	100	17	83	100		100	83		
Ruby	RuboCop	9	100	44	33	100		89	44		
	Brakeman	7	100	86	57	71		14	71	57	43
Multi	Semgrep	6	100	67	67	100		67	100	50	50
	SonarLint	6	100	100	83	100		100	17	17	17
Total		119	110	60	59	108	6	82	69	14	8

- *Text*: we obtain kappa values of 0.3, 1, and 0.741 between the first author and the three other raters, indicating a rather strong agreement on textual content, except with one rater. The rater with the lowest agreement value performed its rating at the sentence level, while the other raters coded at the finer-grained level of individual words.
- *Code*: the kappa values are 1, 0.71, and 1, indicating a very strong agreement on this type of content.
- *Hyperlinks*: we obtain kappa values of 0.04, 0.33, and 0.14, suggesting a low agreement between the raters for this type of content. The disagreements are due to: i) some raters only rated using the context in which the link was used, without looking at its content, to perform the rating ii) some raters did not assign any rating to very general links, while others assigned all the possible ratings.

Since the agreement is overall high, with clear reasons for the disagreements we observe, our results indicate that our taxonomy is suitable for classifying ASAT documentation content.

3.3 Results

The first author applies the taxonomy to the 119 rules of Table 2 to highlight the content types and purposes for all documentation content pertaining to the **Comprehension** category. With regard to the objectivity issues identified in Section 3.2, the first author applies a fine-grained strategy to rate the *Text* content (rating at the word level), and an optimistic strategy to rate *Hyperlinks* (assigning multiple purposes to general documents using their content). Table 3 shows the percentage of rules, for each ASAT, that document the *What*/*Why*/*Fix* purposes for *Text*, *Code*, and *Hyperlinks*.

Purposes. Out of the 119 analyzed rules, 119 (100%) document the *What* purpose, 60 (50%) document the *Why* purpose, and 92 (77%) document the *Fix* purpose, regardless of the content type. Breaking down by content type, we see:

- *What*: 110 rules document it with *Text* (92%), 108 with *Code* (91%), and 69 with *Hyperlinks* (58%).
- *Why*: 60 rules document it with *Text* (50% overall, 100% when present), 6 with *Code* (5% overall, 10% when present), and 14 with *Hyperlinks* (12% overall, 23% when present).
- *Fix*: 59 rules document it with *Text* (50% overall, 64% when present), 82 with *Code* (69% overall, 90% when present), and 8 with *Hyperlinks* (7% overall, 9% when present).

Content types. *Text* is present in 110 of 119 rules (92%), *Code* in 108 (91%), and *Hyperlinks* in 70 (59%). Breaking down by purpose, we see:

- *Text* documents the *What* purpose in 100% of cases, the *Why* in 55% of cases, and the *Fix* in 54% of cases.
- *Code* documents the *What* purpose in 100% of cases, the *Why* in 6% of cases, and the *Fix* purpose in 76% of cases.
- *Hyperlinks* document the *What* purpose in 99% of cases, the *Why* in 20% of cases, and the *Fix* in 11% of cases.

By tool. As seen in the nomenclature, there is some variability. While All ASATs document the *What* purpose, one ASAT does not document the *Fix* purpose (OCLint) and some rarely document it (Cppcheck, Psalm). Finally, some ASATs do not document the *Why* purpose (Roslynator, PHP CS Fixer) and some rarely document it (Psalm, Pylint, Flake8, OCLint).

Finding #2: ASAT documentation has three main purposes: while the *What* is systematically documented, the *Fix* is often documented (77%), and the *Why* is documented only half of the time (50%). The *What* is documented with *Text* (92%) and *Code* (91%), the *Why* with *Code* (100%), and the *Fix* with *Code* (90%) and *Text* (69%).

4 QUESTIONNAIRE SURVEY

In this section, we evaluate whether the documentation of ASAT rules meet the expectations of their users, as well as validating the taxonomy with said users. Specifically, we assess whether the purposes documented in the rules and their incarnation as text, source code, and hyperlinks satisfy the developers facing them. To answer these questions, we design an anonymous questionnaire survey with a mix of open-ended and closed-ended questions shared with industrial partners and fellow researchers. Section 4.1 presents the design of our survey and Section 4.2 gives an overview of the participants and the analysis methodology. Section 4.3 details the quantitative results obtained for closed-ended questions and Section 4.4 the qualitative results emerging from the open-ended questions. The survey, responses, and plots we discuss in this section are available on an interactive companion webpage.[7]

4.1 Survey Design

The survey opens with a welcome message detailing its goals, authors, estimated completion time, and data policy. The remainder of the survey revolves around four parts: developer profile, taxonomy evaluation, rules analysis, and general feedback. Table 4 shows the questions in the survey.

Developer profile. To establish the profile of our participants, we ask about their experience as developers and which programming languages they use regularly. Participants may pick between four groups: *Novice* (0 to 4 years of experience), *Junior* (5–9), *Confirmed* (10–19), and *Senior* (20+). They select or enter their preferred programming languages from an open-ended list. As our study focuses explicitly on ASATs, we ask the participants about their experience with linters.[8] The first question asks whether they know what a linter is, the second whether they use linters on some of their projects, and the last one which linters they use (picked from the 16 ASATs we study in this paper or entered manually).

Taxonomy evaluation. In this part, participants are shown a screenshot of the `FetchEnvVar` rule from RuboCop[9] which serves as an illustration to the terminology we use in the survey (linter, rule, compliant code, non-compliant code). Then, we introduce the participants to the terms of our taxonomy and their definition: purposes (*What*/*Why*/*Fix*) and content types (*Text*, *Code*, *Hyperlinks*). We then ask the participants to evaluate the importance of each purpose in the documentation of ASAT rules. We employ an asymmetric survey response scale inspired by Kano *et al.* [24] and adapted by Begel *et al.* for software engineering [25]: *Essential*, *Worthwhile*, *Unimportant*, *Unwise*, *I don't understand*. For each purpose, we include an additional open-ended question asking why its presence in the documentation is or is not important. The final open-ended question asks the participants whether ASATs should document additional purposes, which we may have missed in our taxonomy.

Rules analysis. In this part, participants are tasked to evaluate the documentation of concrete ASAT rules. For each of the 12 rules used to validate our taxonomy in Section 3.2, we create a bespoke page in the survey. The page includes a screenshot of the rule's documentation, a link to its official documentation, and a series of questions regarding its quality. As our goal is not to evaluate the ability of participants to annotate the rule with our taxonomy, the screenshot includes information regarding its purposes and content types (as shown in Figure 2), agreed upon by four authors.

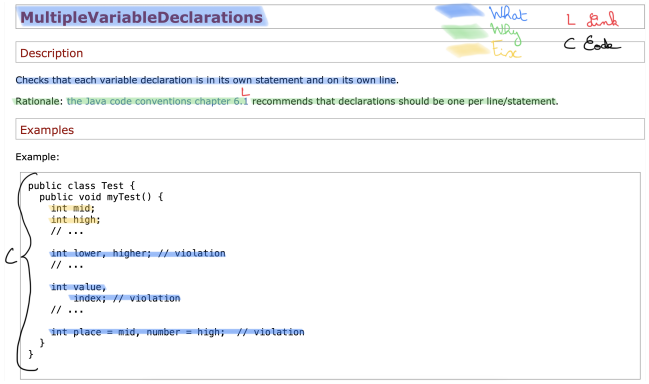


Figure 2: Taxonomy applied to rule *MultipleVariableDeclarations* from `Checkstyle`

The first question asks whether the participant already knows the rule. Then, for each purpose and each content type, a question asks to evaluate the importance of this content type to document this specific purpose using the asymmetric scale introduced above. Finally, the last series of questions asks the participant to judge the overall quality of the documentation for each purpose, using a symmetric scale to measure satisfaction: *Very satisfied*, *Satisfied*, *Neither satisfied nor dissatisfied*, *Dissatisfied*, *Very dissatisfied*. Participants may answer *Not present* to any question, indicating that there is no content of the appropriate type or that the rule does not document the purpose of interest. The *Not present* answer also serves as a quality check, as shown later in Section 4.2.

When participants complete the first two parts of the survey, one rule out of the 12 is drawn randomly and displayed. They can then opt-in to analyze another rule, drawn randomly from the remaining ones until there is no more rule to examine. When a participant evaluates all the rules or opts out, he is redirected to the last part of the survey.

General feedback. This final part consists of a single open-ended question asking the participants to comment on the documentation of the rules they evaluated. This question is designed to put our taxonomy aside and invite the participants to share their opinions more freely: what they liked and disliked about the rules and their documentation, and how it compared with their expectations.

4.2 Participants and Methodology

We primarily shared the survey with industrial partners and fellow researchers through direct contact and mailing lists. We also publicized it on social and professional networks (Twitter, LinkedIn, Slack). The survey was available online on a self-hosted LimeSurvey instance from July 4 to October 19, 2023. Overall, we received a total of 179 anonymous answers. The participants left at different stages: 179 entered their developer profile, 119 evaluated our

[7] https://icpc2024-asats.github.io?page=survey
[8] The survey uses the term linter rather than ASAT, as it is much more popular.
[9] https://docs.rubocop.org/rubocop/cops_style.html#stylefetchenvvar

Table 4: Our questionnaire survey's questions. We used *linter* instead of ASAT as it is more popular among developers.

	Question	Type	Mandatory
Developer profile	What is your experience as a developer?	Single choice	✓
	Which of the following languages do you use regularly?	Multiple choices	✗
	Do you know what a linter is?	Yes/No	✓
	Do you use a linter on some of your projects?	Yes/No	✗
	Which of the following linters were used in those projects?	Multiple choices	✗
Taxonomy evaluation	Rate the usefulness of each purpose in the documentation of a linter	Single choice for each purpose	✓
	For the *What* purpose, why do you think it is (not) important to be present in the documentation?	Open-ended	✗
	For the *Why* purpose, why do you think it is (not) important to be present in the documentation?	Open-ended	✗
	For the *Fix* purpose, why do you think it is (not) important to be present in the documentation?	Open-ended	✗
	Do you think that there are other purposes that a linter documentation should have?	Open-ended	✗
Rules analysis	Have you ever seen this rule?	Yes/No	✓
	For the rule and taxonomy provided, evaluate for each type of content its importance to explain the What purpose	Single choice for each type	✓
	For the rule and taxonomy provided, evaluate for each type of content its importance to explain the Why purpose	Single choice for each type	✓
	For the rule and taxonomy provided, evaluate for each type of content its importance to explain the Fix purpose	Single choice for each type	✓
	For the rule and taxonomy provided, indicate your satisfaction level on the quality of the documentation for each purpose	Single choice for each purpose	✓
General feedback	Please comment freely on the linters documentation you saw: what you appreciated, disliked, and how it compared with your expectations.	Open	✗

taxonomy, 85 evaluated at least one rule (for a total of 289 rule evaluations), and 26 answered the last open-ended question.

Our methodology to sanitize the data and analyze the responses is as follows. First, we clean up the responses and attempt to remove noise. As mentioned earlier, the participants can answer *Not present* when a type of content or a given purpose is missing from the documentation of the rule they are evaluating. We observe that, in some cases, participants marked some type of content or purpose as *Not present* although it was present and marked as such in the screenshot. For instance, some participants answered that the *Fix* purpose for the *Code* was not present in the screenshot of Figure 2. In this case, as a sanity measure, we discard the participant's answers related to this purpose for the given rule, for all content types. We apply the same filter when a participant provides an evaluation for a purpose that is not present in the rule presented to them. We obtained 225 evaluations for the *What* purpose, 91 for the *Why* purpose, and 161 for the *Fix* purpose.

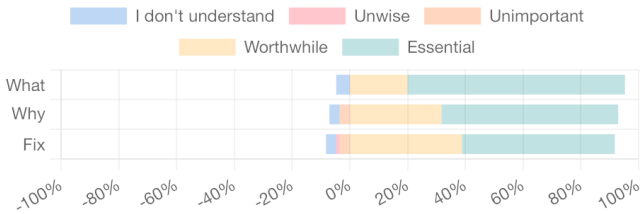
Second, we use thematic analysis [26] to extract codes from the answers to open-ended questions. Two authors read these answers and assigned codes. Then, the four first authors gather to harmonize the codes under higher-level themes, discussed in Section 4.4. Overall, we obtained 33 answers regarding whether ASAT rules should document other purposes, 56 responses evaluating the importance of each purpose in the documentation (168 in total), and 26 responses to the *General feedback* question.

4.3 Quantitative analysis

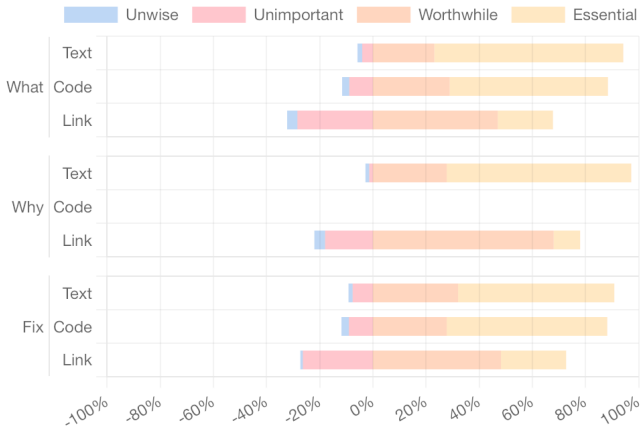
In this section, we review the responses to closed-ended questions. We only include the responses of the 85 participants that have evaluated at least one rule.

Developer profile. The distribution of participants in terms of experience is fairly even. Of the 85 participants, 37 have less than 5 years of experience as a developer (novices), 22 have between 5 and 9 years (juniors), and 26 have more than 10 years (seniors). The most used programming language is C/C++ (55%), closely followed by Python (54%), JavaScript and TypeScript (44%), and Java (42%). The remaining languages are used by less than 15% of respondents.

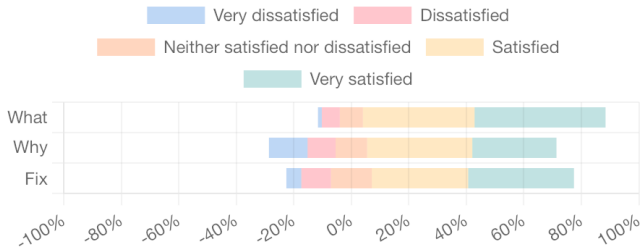
A large majority (81%) of participants do know what an ASAT is; the proportion grows with experience (73% of novices, 82% of juniors, and 92% of seniors). The same trend is found for ASAT usage: 65% of the participants use or used ASATs in their projects (46% for novices, 68% for juniors, and 88% for seniors). There is also a noticeable imbalance in ASAT use depending on the programming languages they use: 53% of C/C++ developers use an ASAT, 61% of Python developers, 64% of Java developers, and 81% of JavaScript and TypeScript developers. A plausible explanation is that ESLint is often bundled by default when initializing JavaScript and TypeScript projects. When participants use ASATs in their projects, the most popular one is indeed ESLint (41%), followed by Pylint (21%), and SonarLint (19%). The remaining tools are used by less than 10% of participants; three respondents mention additional ASATs they use—clang tidy, Fortify and OCaml platform—indicating that our selection of ASATs reflects the ones used in practice. Overall ASAT



(a) Usefulness of each purpose in the documentation



(b) Importance of each type of content to document each purpose



(c) Quality of the documentation for each purpose

Figure 3: Participants' evaluation of the usefulness, importance, and quality of the content types and purposes

usage reflects language use, with the exception of C/C++: since developers use more than one language, C/C++ users tend to use ASATs with other programming languages.

Taxonomy evaluation. In this part, we ask participants to judge the usefulness of the *What*, *Why*, and *Fix* purposes in the documentation of ASATs, independent of any particular rule. Figure 3a shows the results: the participants strongly expect each of the purposes to be present and documented. While the *What* and *Why* purposes are largely judged as essential, the *Fix* purpose appears slightly less essential to participants, but still worthwhile. In particular, we note the importance of the *Why* purpose to participants, which indicates that rule documentation should not only document the problem and its solution but also the rationale motivating the rule; in contrast, *only half of the rules* we analyzed had a rationale documented (Table 3). When grouping answers by developer experience, programming language, or other profile criteria we do not observe any major differences.

Rules analysis. The 85 participants analyzed a total of 289 rules, with a mean of 3.4 rules per participant. Each of the 12 rules has been evaluated by 19 to 29 different participants. Figure 3b shows how the participants rate the importance of each type of content to document each purpose in the rules they examined. If a rule does not include a particular type of content to document a given purpose, and the participant marks it as *Not present*, we omit this data point as it does not convey any positive or negative judgment. This explains why there is no evaluation of *Code* to document the *Why* purpose, as none of the 12 rules document it with code.

Participants evaluate the importance of the *What* purpose positively (*Essential* or *Worthwhile*), regardless of its incarnation (94% for *Text*, 88% for *Code*, and 68% for *Hyperlinks*). While text and source code are seen as essential, the *Hyperlinks* are mainly judged as worthwhile. For the *Why* purpose, participants mostly evaluate *Text* as essential (69%) and *Hyperlinks* as worthwhile (68%). For the *Fix* purpose, participants mostly evaluate *Text* (59%) and *Code* (60%) as essential, and *Hyperlinks* as worthwhile (48%).

Participants evaluate *Text* and *Code* very positively to explain the three purposes. This suggests that a combination of text and source code might be the best choice to document ASAT rules. Participants also evaluate *Hyperlinks* positively, with a minimum of 68% of positive evaluations for each purpose. Yet, most participants evaluate them as *Worthwhile* rather than *Essential*. Moreover, a significant portion judges them as *Unimportant*: three times or more than *Text* or *Code*—and more than they are judged *Essential*. A possible reason is that visiting external resources disrupts the reading flow and that important information may be lost among other resources, particularly if linking to larger documents. A solution could be to extract and distill the important information from external resources into the documentation, and cite it as a source.

> **Finding #3:** Text and source code are best suited to document the *What* and *Fix* purposes, and good documentation for an ASAT rule should include both. *Text* is the best-suited medium to document the *Why* purpose. *Hyperlinks* are rarely seen as essential and should be used sparingly.

Finally, Figure 3c displays the satisfaction of participants regarding the quality of the documentation in the rules they evaluated, for each purpose. We observe that the participants are mostly satisfied with the quality of the documentation for the *What* (84%) and *Fix* purposes (70%). Yet, we observe that almost a quarter of the participants are not satisfied with the quality of the documentation for the *Why* purpose. Worse, in 13% of cases, participants are very dissatisfied with its quality. This strongly contrasts with the usefulness evaluation emitted by participants in Figure 3a.

> **Finding #4:** Participants express a strong interest in understanding the rationale behind ASAT rules when reading their documentation (*Why*). Yet, **Finding #1** indicates that only 50% of ASAT rules document the *Why* purpose; when present, participants are dissatisfied with how it is documented in close to 25% of the cases. Clearly, *documentations should consistently include and explain the* Why *aspect.*

4.4 Qualitative analysis

In the following, we summarize free-form comments based on our coding. Codes mentioned for the first time include their frequency **like this (0)**. Code referenced after being introduced once is *like this*. We do not report on "obvious" codes (e.g., **understanding the rationale (29)** for the why purpose).

Transversal theme: learning. One of the most salient themes, across all three purposes (**learning-what (11)**, particularly **learning-why (21)**, **learning-fix (13)**), is *learning*: 45 comments touched on that theme in one way or another. This is sometimes phrased as self-improvement of the developer's skills, particularly for beginners or (more occasionally) for onboarding new team members. Comments on the *What* aspect mention the need to understand the error to not repeat it (thus improving skills), as well as team aspects (*"because it facilitates the integration of new members"*). Comments on the *Why* are the most prevalent. Explaining the rationale for an error is important to understand its purpose and importance, and from that, remembering it: *"If I don't know why, then I don't know why it's a bad thing and I cannot improve as a developer"*. Finally, the *Fix* is more immediate. Once the problem is known and understood, learning of possible solutions is valuable: *"It is likely the coder introduced a bad pattern/error due to lack of expertise; as such, it would be wrong to assume he/she will know how to fix it"*.

Transversal theme: saving time. Respondents emphasized efficiency in all three aspects (**saving time-what (2)**, **saving time-why (3)**, particularly **saving time-fix (13)**, **automated fixes (4)**). For instance, one respondent wants to *"understand in seconds what a lint is about"*, which requires clear and concise explanations. Missing information in the *Why* prevents one from deciding whether to act on a warning (*"I will probably lose time looking it up on the internet. Also I will be less motivated to fix it"*). Since fixes are the most actionable, there is more demand to have standard solutions available to solve the issue quickly, rather than searching for information on the web or asking teammates. Having to search for the information is perceived to increase the chance a warning is ignored. Going further, the logical end is automation: *"ideal thing is to just click a button to 'autofix' the issue, when available"*.

Aspects specific to the What. The *What* provides understanding of what triggers a rule. It should be written clearly and concisely to this as easy as possible (thus saving time). The *What* helps finding **where the warning is (8)**, which is not always obvious as *"many different things may be discussed/considered on a single piece of code"*. A key step in finding where the error is, is to **relate the error to one's code (12)**: rules are either described in the abstract, or, at best, with an **example (3)**. Examples are preferred for this: *"simpler the exemple [sic], the easier it is to relate to one's code"*. Another theme relates to **false positives and negatives (5)**. Rules are implemented by heuristics that may be imperfect especially for complex cases (e.g. regular expressions). Describing what triggers a rule in details is useful to disambiguate between true positives and false positives, as well as knowing cases that the rule can miss (false negatives).

Aspects specific to the Why. The *Why* is key to deciding whether to act on the warning, or not. Developers **analyse the tradeoffs and risks involved (11)**: the *Why* should in particular explain the severity of the warning: *"I can judge whether the criticality justify modifying this piece of code. I may choose to disregard the rule if I judge it not worthwhile"*. In some cases, the rule's **relevance (9)** will be questioned (such as when it is a *false positives and negatives*, or when it is a matter of personal or team **preferences (5)**, rather than a real issue. Thus the *Why* should **motivate and justify the effort (10)** that will be invested in fixing the warning; needless to say, if said effort is low (*saving time* via good *examples* or *automated fixes*), it will be easier to act on it.

Aspects related to the Fix. There is more debate as to the importance of the *Fix*, compared to the *What* and the *Why*. Some respondents state **fixes are less or not important (7)** for several reasons: either because the rules are simple, the fixes would be too basic, or because the *Why* and *What* are sufficient (*"in most cases previous information should be enough to infer how to fix"*. For other respondents, **fixes are essential or very important (10)**. For them, documentation without a fix is not actionable: *"If the Fix is not here, understanding the what and the why lead us to nowhere"*. Fixes **increase ASAT friendliness (5)**, and help *saving time*. Fixes are particularly **useful when they are not obvious (4)**, either due to lack of knowledge (*learning*) or due to their difficulty. Providing **examples (10)** is useful to clarify complex rules. Some respondents mention that **fixes may not be optimal (7)** given the context, at worse they can *"lead beginners to apply a cascade of bad decisions made to satisfy the linter"*.

Other purposes for documentation. Most respondents mentioned **no additional purpose (14)** than the *What*, *Why*, and *Fix*. A few respondents did mention other documentation purposes, such as **exceptions, alternatives, and limitations (3)**, **risks (3)**, or **configuration (2)** of the ASAT warnings. Indeed, we encountered these elements and included them in our nomenclature, but decided to exclude them from the remainder of our analysis as they dealt with more operational aspects.

Additional comments on the documentation. Several participants emphasized in their free-form comments some topics that emerged earlier. Several respondents highlighted the **need for a summary (7)** (*"Sometimes too much text which does not encourage taking the time to read and therefore deal with the error"* or a **structured template (6)** where *"the what, the why and the fix are clearly separated"*. Others reiterated the need to **use code examples (4)**, including adding examples of both compliant and non-compliant code, or to **avoid links (2)** (*"external links are almost never useful"*).

> **Finding #5:** ASAT documentation has important learning purposes (particularly the *Why*), and should be written with efficiency in mind. Respondents read the *What* to understand the error, and the *Why* to decide whether to *Fix* the warning; missing elements slow down the process, making fixes less likely. Respondents recommend to use summaries, examples, and to avoid links.

5 THREATS TO VALIDITY

External validity. Our study bears several threats with regard to external validity. The first threat is that our corpus of ASATs and

ASATs rules documentation is not representative, and is even biased toward popular ASATs. Therefore, we have no guarantee that our results would generalize to the actual population, especially the percentages displayed in Table 2 and Table 3. Another threat is that the respondents of our survey are not a random sample of the population of ASATs users. As a consequence, the results we obtained with our participants might not generalize, especially the percentages displayed in Figures 3a to 3c.

Internal validity. We extensively use qualitative methods to build our nomenclature and taxonomy (open card sorting [21]) and analyze the open answers to the survey (thematic analysis [26]). It is well known that these methods can be affected by subjectivity [27]. As a consequence, different researchers might have obtained a different nomenclature, taxonomy, and other themes from the survey answers. As a mitigation measure for the taxonomy, we performed an internal validation and a reality check in the survey where we observed that it was well understood by the respondents. For the other results obtained from qualitative analysis, we systematically used harmonization sessions to reduce the subjectivity of our findings.

With regard to the answers of the participants to the survey, there is a chance that they did not fully understand the questions we asked. This threat affects mostly the closed-ended questions where we cannot do any sanity check by looking at the answer. The most difficult questions concerned the analysis of the usefulness of each type of content to document each purpose, as the participants needed to carefully analyze how we rated the content of the rules. As a sanity check, we included a specific scale item (Not present) to double-check that the participants did understand our rating. We used this sanity check to filter out incoherent answers. However, this sanity check is not perfect and it is possible that participants answered these questions without understanding our rating.

6 RELATED WORK

To the best of our knowledge, there is no prior work having studied the rules documentation of ASATs. In the remainder of the section, we discuss the related work according to two topics: studies of ASATs and studies of software documentation.

6.1 ASATs studies

Researchers have emphasized various benefits associated with the use of ASATs. Tómasdóttir *et al.* highlight eight such advantages, such as error prevention, keeping the code simple and consistent, or improving the efficiency of code review and discussions [14]. ASATs can also automate compliance to a standard coding style [28].

Several studies explore the reasons inhibiting the use of ASATs by developers. Tómasdóttir *et al.* highlight that the agreement on rules to enable, especially for pre-existing projects, is an important issue and makes it difficult for developers to follow the rules when they disagree [14]. Other challenges were also raised: dealing with false positives [29], having better integration into the development process [17] and missing quick fixes when rules are detected [30].

The study from Novak *et al.* [1] produces a taxonomy of ASATs across 10 categories (such as rule domains or configurability). However, ASAT documentation has not been investigated in their study.

The studies on the warnings content of ASATs are closely related to ours, as warnings are the first piece of rule documentation that developers see when using an ASAT. An important result is that their content might not be sufficiently helpful to developers [7, 31]. Another finding is the need to have clear and concise warnings, especially when developers use ASATs in CLI [32]. It aligns with our findings for ASAT rule documentation: the main information should be quickly available and understandable. Buckers *et al.* present a tool and its evaluation to help developers treat more efficiently the numerous notifications and their contents [33].

6.2 Software documentation

Another related topic is the study of documentation of Application Programming Interfaces (APIs). API documentation shares a common goal with ASAT documentation as they are meant to teach developers how to correctly use an API as quickly as possible [18]. On the other hand, they focus on a single software component while ASAT documentation describes rules that apply in a wide range of contexts. We also find similar expected criteria with the writing of the rules documentation, as the need to include short code snippets demonstrating API usage in context or code examples illustrating the best practices [34, 35].

Aghajani *et al.* performed an empirical study on software documentation resulting in a taxonomy of documentation issues [36]. An interesting outcome is the identification of themes relatively close to ours, especially *Information Content* similar to our *Comprehension*. They pursue their work in another article [37] by realizing two surveys, first to evaluate the relevance of issues they had found, then to explore the needs on type of documentation. It results in guidelines to improve the state of the art around documentation.

Regarding software documentation in general, one major flaw in software documentation is the difficulty of keeping it updated through its lifetime [38]. Some developers simply assume it is outdated and do not rely on it [39]. At first glance, we did not see this problem in the documentation of ASAT rules maybe because they evolve less often, but it would be an interesting issue to study.

7 CONCLUSION

In this article, we explored how ASATs rules are documented, and contrasted them with developers' expectations via a two-phase study. We first studied how more than 100 rules—spanning 16 ASATs in multiple programming languages—are documented, leading to a nomenclature of documentation elements refined in a taxonomy of documentation purposes and content types. We then use this taxonomy to contrast the documentation of 12 real-world rules with developer expectations via a survey involving 85 respondents who evaluated the rules 289 times. Among other findings, we highlight issues with the *Why* purpose: despite being considered essential by developers to decide whether to act on a warning, half of the rules miss it; of the rest, a quarter of survey responses point at quality issues. We also find out that code examples in addition to text are attractive for documenting the *What* and *Fix* purposes. Finally, some developers expressed concern about saving time, leading to the recommendation to include summaries, and reducing the use of external links that disrupt the reading flow. In future work, we plan to extend our study to more ASATs and to more quality issues inside the documentation, such as outdatedness, as has been done for API documentation [36, 37].

REFERENCES

[1] J. Novak, A. Krajnc, and R. Žontar, "Taxonomy of static code analysis tools," in *The 33rd International Convention MIPRO*, May 2010, pp. 418–422.

[2] K. F. Tómasdóttir, M. Aniche, and A. Van Deursen, "The Adoption of JavaScript Linters in Practice: A Case Study on ESLint," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 863–891, Aug. 2020, conference Name: IEEE Transactions on Software Engineering.

[3] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Suita: IEEE, Mar. 2016, pp. 470–481. [Online]. Available: http://ieeexplore.ieee.org/document/7476667/

[4] R. K. McLean, "Comparing Static Security Analysis Tools Using Open Source Software," in *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, Jun. 2012, pp. 68–74.

[5] Q. Ashfaq, R. Khan, and S. Farooq, "A Comparative Analysis of Static Code Analysis Tools that check Java Code Adherence to Java Coding Standards," in *2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE)*, Mar. 2019, pp. 98–103.

[6] S. Habchi, X. Blanc, and R. Rouvoy, "On adopting linters to deal with performance concerns in Android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, Sep. 2018, pp. 6–16. [Online]. Available: https://doi.org/10.1145/3238147.3238197

[7] M. Tahaei, K. Vaniea, K. K. Beznosov, and M. K. Wolters, "Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Yokohama Japan: ACM, May 2021, pp. 1–17. [Online]. Available: https://dl.acm.org/doi/10.1145/3411764.3445616

[8] H. J. Kang, K. L. Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: how far are we?" in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 698–709. [Online]. Available: https://doi.org/10.1145/3510003.3510214

[9] Y. Kim, J. Lee, H. Han, and K.-M. Choe, "Filtering false alarms of buffer overflow analysis using SMT solvers," *Information and Software Technology*, vol. 52, no. 2, pp. 210–219, Feb. 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058490900175X

[10] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis," in *Static Analysis*, ser. Lecture Notes in Computer Science, C. Hankin and I. Siveroni, Eds. Berlin, Heidelberg: Springer, 2005, pp. 203–217.

[11] D. A. Tomassi, "Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 980–982. [Online]. Available: https://dl.acm.org/doi/10.1145/3236024.3275439

[12] D. A. Tomassi and C. Rubio-González, "On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 292–303, iSSN: 2643-1572.

[13] D. Kavaler, A. Trockman, B. Vasilescu, and V. Filkov, "Tool Choice Matters: JavaScript Quality Assurance Tools and Usage Outcomes in GitHub Projects," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 476–487, iSSN: 1558-1225.

[14] K. F. Tomasdottir, M. Aniche, and A. van Deursen, "Why and how JavaScript developers use linters," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana, IL: IEEE, Oct. 2017, pp. 578–589. [Online]. Available: http://ieeexplore.ieee.org/document/8115668/

[15] L. N. Q. Do, J. R. Wright, and K. Ali, "Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 835–847, Mar. 2022, conference Name: IEEE Transactions on Software Engineering.

[16] J. Smith, L. N. Q. Do, and E. Murphy-Hill, "Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security," in *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, 2020, pp. 221–238. [Online]. Available: https://www.usenix.org/conference/soups2020/presentation/smith

[17] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 672–681, iSSN: 1558-1225.

[18] R. Watson, M. Stamnes, J. Jeannot-Schroeder, and J. H. Spyridakis, "API documentation and software community values: a survey of open-source API documentation," in *Proceedings of the 31st ACM international conference on Design of communication*. Greenville North Carolina USA: ACM, Sep. 2013, pp. 165–174. [Online]. Available: https://dl.acm.org/doi/10.1145/2507065.2507076

[19] M. Nagappan, R. Robbes, Y. Kamei, E. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan, "An empirical study of goto in C code from GitHub repositories," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo Italy: ACM, Aug. 2015, pp. 404–414. [Online]. Available: https://dl.acm.org/doi/10.1145/2786805.2786834

[20] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in *2009 IEEE 17th International Conference on Program Comprehension*, May 2009, pp. 158–167, iSSN: 1092-8138.

[21] T. Zimmermann, "Card-sorting: From text to themes," in *Perspectives on Data Science for Software Engineering*, T. Menzies, L. Williams, and T. Zimmermann, Eds. Boston: Morgan Kaufmann, Jan. 2016, pp. 137–141. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128042069000271

[22] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, Mar. 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09750-5

[23] R. Passonneau, "Measuring Agreement on Set-valued Items (MASI) for Semantic and Pragmatic Annotation," in *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*. Genoa, Italy: European Language Resources Association (ELRA), May 2006. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2006/pdf/636_pdf.pdf

[24] K. N, "Attractive Quality and Must-Be Quality," *Journal of the Japanese Society for Quality Control*, vol. 31, no. 4, pp. 147–156, 1984. [Online]. Available: https://cir.nii.ac.jp/crid/1572261550744179968

[25] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India: ACM, May 2014, pp. 12–23. [Online]. Available: https://dl.acm.org/doi/10.1145/2568225.2568233

[26] G. Guest, K. M. MacQueen, and E. E. Namey, "Applied Thematic Analysis," Oct. 2023. [Online]. Available: https://uk.sagepub.com/en-gb/eur/applied-thematic-analysis/book233379

[27] R. Dowling, "Power, subjectivity and ethics in qualitative research," in *Qualitative research methods in human geography*, I. Hay, Ed. South Melbourne, Vic.: Oxford University Press, 2005, pp. 19–29.

[28] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Sep. 2008, conference Name: IEEE Software. [Online]. Available: https://ieeexplore.ieee.org/document/4602670

[29] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, Apr. 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584910002235

[30] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual South Korea: ACM, Jul. 2022, pp. 532–543. [Online]. Available: https://dl.acm.org/doi/10.1145/3533767.3534374

[31] B. Johnson, R. Pandita, J. Smith, D. Ford, S. Elder, E. Murphy-Hill, S. Heckman, and C. Sadowski, "A cross-tool communication study on program analysis tool notifications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Seattle WA USA: ACM, Nov. 2016, pp. 73–84. [Online]. Available: https://dl.acm.org/doi/10.1145/2950290.2950304

[32] P. L. Gorski, Y. Acar, L. Lo Iacono, and S. Fahl, "Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. Honolulu HI USA: ACM, Apr. 2020, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3313831.3376142

[33] T. Buckers, C. Cao, M. Doesburg, B. Gong, S. Wang, M. Beller, and A. Zaidman, "UAV: Warnings from multiple Automated Static Analysis Tools at a glance," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2017, pp. 472–476.

[34] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What Should Developers Be Aware Of? An Empirical Study on the Directives of API Documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012. [Online]. Available: http://www.monperrus.net/martin/An-Empirical-Study-On-the-Directives-of-API-Documentation.pdf

[35] A. Cummaudo, R. Vasa, and J. Grundy, "What should I document? A preliminary systematic mapping study into API documentation knowledge," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Sep. 2019, pp. 1–6, iSSN: 1949-3789.

[36] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software Documentation Issues Unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 1199–1210, iSSN: 1558-1225. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8811931

[37] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: the practitioners' perspective," in *Proceedings of the ACM/IEEE 42nd International Conference on Software*

Engineering, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 590–601. [Online]. Available: https://doi.org/10.1145/3377811.3380405

[38] A. Forward and T. C. Lethbridge, "The Relevance of Software Documentation, Tools and Technologies: A Survey," in *Proceedings of the 2002 ACM Symposium on Document Engineering*, ser. DocEng '02. New York, NY, USA: ACM, 2002, pp. 26–33. [Online]. Available: http://doi.acm.org/10.1145/585058.585065

[39] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 255–265.