A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools

Marcus Nachtigall
Heinz Nixdorf Institute, Paderborn University
Germany
marcus.nachtigall@uni-paderborn.de

Michael Schlichtig
Heinz Nixdorf Institute, Paderborn University
Germany
michael.schlichtig@uni-paderborn.de

Eric Bodden
Heinz Nixdorf Institute, Paderborn University & Fraunhofer IEM
Germany
eric.bodden@uni-paderborn.de

ABSTRACT

Static analysis tools support developers in detecting potential coding issues, such as bugs or vulnerabilities. Research on static analysis emphasizes its technical challenges but also mentions severe usability shortcomings. These shortcomings hinder the adoption of static analysis tools, and in some cases, user dissatisfaction even leads to tool abandonment.

To comprehensively assess the current state of the art, this paper presents the first systematic usability evaluation in a wide range of static analysis tools. We derived a set of 36 relevant criteria from the scientific literature and gathered a collection of 46 static analysis tools complying with our inclusion and exclusion criteria—a representative set of mainly non-proprietary tools. Then, we evaluated how well these tools fulfill the aforementioned criteria.

The evaluation shows that more than half of the considered tools offer poor warning messages, while about three-quarters of the tools provide hardly any fix support. Furthermore, the integration of user knowledge is strongly neglected, which could be used for improved handling of false positives and tuning the results for the corresponding developer. Finally, issues regarding workflow integration and specialized user interfaces are proved further.

These findings should prove useful in guiding and focusing further research and development in the area of user experience for static code analyses.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; **Software usability**.

KEYWORDS

static analysis, program analysis, explainability, user experience, tool support

ACM Reference Format:
Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. 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 (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534374

1 INTRODUCTION

Static analysis tools analyze program code without executing it. They are used in different contexts and for different purposes, ranging over different complexities. For instance, there are rather simple tools considering code quality [18, 73], more complex bug finders [47, 71], and even more sophisticated security scanners searching for exploitable vulnerabilities [23, 64]. Research in static analysis has made much progress, allowing a more diverse application of static analysis.

While the central task of static analysis is detecting respective code problems, the tool also has to inform the developer about these issues, support them in resolving issues, and offer all of its features in a usable way. If the tool is not able to support the developer in these aspects, they will not be able to solve or even understand the issue—they might not even believe the tool and regard correct warnings as a false positive [8]. If it is too difficult to access the tool's features in a user-friendly way, the developer will simply not use it. Hence, previous research has shown that it is insufficient to only study the technical perspective of static analyses and that research should rather also take into focus the users' perspective, and support users accordingly [12, 20, 34, 66]. Over the past decade, this perspective has been supported by several publications, pointing to specific flaws and weaknesses when it comes to the user experience with static analysis tools [25, 29, 61, 74, 78].

This paper is the first to provide a *comprehensive* view on the current state of Static Analysis Tools (SAT) from the user's perspective. While prior studies drew anecdotal evidence from surveys, interviews, or the authors' own experiences, this paper takes a complementary view on reported issues by instead considering a broad range of existing tools and evaluating them directly. This paper thus presents the first such direct assessment of user interactions offered by a large fraction of the SAT landscape. Importantly, it paints a *current* picture of this landscape—after all, some of the previous studies are several years old and many tools might well have evolved since then.

To evaluate the current state of SATs from this perspective, we executed an assessment of current tools. We searched the current related literature in detail and extracted 36 relevant criteria, which we used for the evaluation. We discussed and defined these criteria with their assessment grades in advance. Furthermore, we collected 243 tools and evaluated a representative set of 46 tools that met our inclusion and exclusion criteria. Our evaluation yields central open challenges in the current state of SATs: Overall, the tools present their warnings rather poorly, which causes a high risk that developers may not understand or even care for the warning. Furthermore, in most cases neither the warning messages, nor any other feature in the tool, supports the developer in fixing the issue. Next, current tools neglect the potential of asking for the developer's knowledge and integrating this information to provide a more tailored user experience. Last but not least, the tool integration into developers' workflow remains an open challenge in many tools.

To summarize, this paper makes these original contributions:

- It presents a catalog of 36 usability-evaluation criteria drawn from the scientific literature.
- It evaluates the current state of SATs by applying these criteria to 46 current SATs.
- For each criterion, the paper discusses why it is relevant, and what fraction of current tools offers features to fulfill the respective criterion.
- It reports open challenges and neglected aspects, which might be starting points for further research.

All raw data we collected is being made available online as a curated artifact at: https://sites.google.com/view/datatoolsurvey/

The remainder of the paper is structured as follows: In section 2 we describe our general methodology on how we collected tools, defined criteria, and executed the evaluation. Section 3 discusses our criteria in more detail and presents our evaluation results. Section 4 discusses potential threats to validity. Section 5 presents some related work. Finally, section 6 concludes with our main findings.

2 METHODOLOGY

We first explain the most important parameters to our usability survey of SATs: Which SATs to evaluate? What evaluation criteria to use? And what exact evaluation procedure to use for each tool?

Collection of Tools. While it is infeasible to compile a complete set of all existing SATs, we aimed at gathering a representative set of tools. Therefore, we searched several prominent websites giving recommendations for which SAT to use.[1] We found these lists from the scientific literature [66] and by snowballing from these lists, as they also recommend further lists. This process resulted in a very substantial set of tools, even after removing duplicates.

As it would not be realizable to evaluate all these tools, and neither give a realistic view on the current state of the art, we decided to apply reasonable inclusion and exclusion criteria to the collected set.

First, we included only tools that consider code of the languages C, C++, Java, JavaScript, C#, and Python, since these appear to be the most used languages [75], also being more relevant in the related literature than the excluded languages. After applying this criterion, we obtained a set of 243 tools in total, which were considered in more detail in the next steps. Second, we included only such SATs that are in some way giving warning messages and hinting at coding issues. This includes different analysis types, such as simple linters, bug finders, and more complex security checkers, but not libraries or model checkers. 51 tools do not fulfill this criterion. Third, we excluded four tools for which we were unable to find their executable (nor source code), or even a related website. Fourth, we excluded most proprietary tools as we were unable to access them, which accounts for 81 tools. As an exception, to broaden our evaluation results, we contacted the leading companies of proprietary tools mentioned in the Gartner Magic Quadrant [76]. In the end, though, only one of these tools was considered, as the remaining reported tools did not match our inclusion criteria or we were not able to access them after all. Fifth, we excluded 41 tools that were not maintained in the previous two years, as these tools would not represent the current state of the art. In this step, we excluded a tool when following the documentation did not lead to a successful installation and when we did not have any further ideas on how the installation might succeed. At last, we had to exclude 20 tools that we were unable to install as described below.

Table 1 gives an overview of the number of overall collected, excluded, and included tools. Full information on each considered tool is provided in our artifact.

Some of the considered lists recommending SATs contain a Multi-Language section. If tools from this section have been included in the evaluation, they are mapped to the language we evaluate it in. Accordingly, all tools were mapped to only one programming language even though they might support several languages. For excluded tools, we did not consider them further or map them to a specific language. The miscellaneous exclusions row includes libraries and frameworks, model checker, IDEs, tools without static analysis, and tools without any kind of warning messages. Overall, 46 tools are included in the evaluation. In this, Java and Python account for the biggest share. In comparison to the amount of included tools, the number of tools we have not been able to install appears somewhat high. This hints at first usability issues of poor documentation which is not considered further in this study, but still is a major perceived pain point in this study.

Evaluation Criteria. The main goal of our evaluation is to assess the usability of current SATs. To conduct such an assessment objectively, one requires clear criteria. We considered different aspects to derive such criteria: we considered specific features, how the user might interact with the tool, what information is given to the user, and how this information is presented to the user. While some criteria are strongly connected to specific features, other criteria are rather abstract and might be fulfilled in different ways. Regarding the former, our approach is somewhat related to a feature analysis as in DESMET [37], yet regarding the latter it extends further. We aim at defining clear criteria to objectively evaluate the tools' usability, but unlike DESMET are not constrained to the context of an organization and its culture. Furthermore, our main focus is not on the features itself and on the process of deciding what tool to use, but rather on using these features as a device to evaluate the overall usability of SATs.

Nachtigall et al. [42] have grouped recurrent usability issues into six categories, which are connected to understandable *Warning Messages*, *Fix Support*, *False Positives*, *Integration of User Feedback*, *Workflow Integration*, and a specialized *User interface*. To expand on

[1]We used the following websites:
https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html
https://security.web.cern.ch/security/recommendations/en/code_tools.shtml
http://projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList
https://github.com/mre/awesome-static-analysis#multiple-languages-1
https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
https://dwheeler.com/essays/static-analysis-tools.html
http://www.spinroot.com/static/

Table 1: Overview of tools considered, excluded, and included overall

	C and C++	Java	Python	C#	JavaScript	Multi-Language	Overall
All considered tools	69	52	28	18	23	53	243
Excluded: Proprietary	29	19	0	4	3	26	81
Excluded: Not maintained	15	8	2	4	7	5	41
Excluded: Unable to install	8	4	2	0	3	3	20
Excluded: Other reasons	12	8	7	4	5	19	55
Included	5	13	17	6	5	0	46

this categorization, we searched the related literature and compiled a set of usability issues related to SATs. For collecting criteria, we were inspired by the approach of systemic literature reviews [36] but did only adhere to in a lightweight form. Particularly, we started with central papers from this area and further used snowballing.

After collecting the usability issues reported by the related literature, we mapped the identified issues to these six categories. Furthermore, we discussed for each issue whether we find reasonable assessment grades. For instance, it would be desirable for warning messages to be clearly understandable [8, 20] or to use natural language [34]. However, it might be very difficult to objectively evaluate whether a tool fulfills these aspects, as this would require a dedicated user study with every single tool. If we could not find objective definitions for a criterion and its assessment grades, we excluded it from our study. For all remaining criteria, we defined in advance what exactly has to be evaluated, and we predefined assessment grades. In most cases, these assessment grades are composed of *yes*, *partially*, and *no*, indicating to what extent the corresponding criterion is fulfilled. The next section explains the criteria in detail.

Setup and Evaluation. Regarding the evaluation of one specific tool, we used the following protocol. First, we searched for the related website or github repository. The websites we collected our tools from usually refer to some of these. On the resulting site, we checked the documentation on the setup of the tool and also on how the tool should be used. We try to set up the tool on a standardized Windows or Linux VM, depending on which platform the tool supports. If the installation (following the documentation) did not succeed, we checked for alternative solutions until either the tool was successfully installed or we ran out of ideas on how to complete the installation. Due to the high workload involved, usually the installation was done by the first author, and only if this did not succeed the second author also tried to install it, again according to the documentation and potential solution ideas. If both failed, the tool was excluded from the study. Since both authors are experts regarding SATs, it is plausible that others would fail here, too, making this exclusion process appropriate. If the installation succeeded, the tool was evaluated using a multiple-raters approach [60]: The first and second author evaluated the tool independently from each other, according to the predefined evaluation criteria. For this purpose, we explored the available documentation and attempted to find relevant features and information in the installed tool. Afterwards, both evaluations were compared. For coinciding evaluations, we set this evaluation as the final evaluation. In case of disagreement, both authors discussed the source of disagreement until agreement on a final evaluation was reached. For every tool we used real world projects and took care to use a project where the respective tool detects many issues. We also used comparable input projects within the same programming languages and analysis scopes.

3 RESULTS

In this section, we discuss the used criteria, tool evaluations, and resulting implications in more detail. Table 2 provides an overview of all included tools and a summarized view of the evaluation for each tool in each category. The table is divided into command line interface (CLI) tools and graphical user interface (GUI) tools. For this rough evaluation, it is easier to receive partial fulfillment in some cases. For instance, some basic criteria from the workflow integration and user interface are fulfilled by many tools. Full results are available in our online artifact. Since the main differences in the evaluations are mainly related to whether the tool is a CLI or a GUI tool, we will repeatedly compare these types of tools. As we found that there is no significant influence by the analyzed programming language, we will not further discuss the language in the following. In the remainder, we will focus on the evaluation criteria, discuss their relevance, and then present our evaluation results.

3.1 Warning Messages

Warning messages are the main approach of SATs to point potential code issues to developers. Therefore, they have to direct the developer’s attention to the detected issue and give information on what might be wrong, why it should be fixed, and how it could be fixed. This information has to be presented in a clear way [8]. Johnson et al. show in their survey [34], that most of their participants criticize the poorly presented tool output and that it should be presented more user-friendly and intuitive. To analyze this area in more detail, we gathered criteria from the literature and evaluate collected tools.

Internal Reasoning. Barik et al. [6] show that warnings only present the final result of the analysis but do not give any insight into the underlying reasoning so that the developer has to mentally duplicate this process. Barik further argues that SATs are able to computationally expose the internal reasoning [4]. Rule graphs are one way to achieve this [19].

In our evaluation, only three tools give at least some insight into the reasoning process. This indicates that related concepts are not as highly prioritized by tool developers as it is by the scientific literature. While it might need quite some implementation effort and is not that relevant for simple linting tools, this aspect is quite relevant for tools that seek to signal complex findings in the code.

Table 2: Overview of all evaluated tools (CLI and GUI).

○ : virtually not fulfilled
◐ : somewhat fulfilled
● : clearly fulfilled

	Warning Msg.	Fix Support	False Positives	User Feedback	Workflow Integr.	User Interface
Bandit [3]	◐	○	◐	◐	◐	○
Cpplint [15]	○	○	○	○	○	○
CScout [16]	○	○	○	○	◐	◐
Dlint [18]	○	○	○	○	◐	○
Flawfinder [24]	◐	○	○	○	◐	◐
Hegel [28]	○	○	○	○	◐	○
InferSharp [30]	○	○	○	○	◐	◐
McCabe [39]	○	○	○	○	◐	○
NodeJSScan [45]	◐	○	○	○	◐	○
Dependency Check [10]	◐	○	●	○	◐	○
Pycodestyle [49]	○	○	○	○	◐	○
Pydocstyle [51]	○	○	○	◐	◐	○
Pyflakes [52]	○	○	○	○	◐	○
Pyre-Check [54]	○	○	○	○	◐	○
Pytype [56]	○	○	○	○	○	○
Radon [57]	○	○	○	○	◐	○
Semgrep [65]	○	○	○	◐	◐	○
Vulture [81]	○	○	●	○	◐	○
Wemake-Python-Style [82]	○	○	○	◐	◐	○
Wily [83]	○	○	○	○	◐	◐
Xenon [84]	○	○	○	○	◐	○
Xo [86]	○	○	○	○	◐	○
Checkstyle [11]	○	○	○	●	◐	◐
CogniCrypt [13]	○	◐	◐	○	◐	◐
Commercial Tool	●	○	○	●	●	●
CppCheck [14]	◐	○	○	●	○	◐
DevSkim [17]	○	○	○	◐	◐	○
ErrorProne [48]	○	◐	○	○	○	○
Eslint [21]	○	○	○	◐	◐	◐
Fb-Contrib [22]	◐	○	◐	○	◐	◐
FindSecurityBugs [23]	◐	○	◐	○	◐	◐
Standard [73]	○	○	○	○	◐	◐
Mypy [41]	○	○	○	○	◐	◐
PMD [47]	○	○	○	●	◐	◐
Puma Scan [63]	◐	◐	◐	●	○	◐
PyDev [50]	○	●	○	○	◐	◐
Pylint [53]	○	○	◐	◐	◐	◐
Pyright [55]	○	○	○	○	◐	◐
Reshift [58]	◐	◐	○	○	◐	●
Roslynator [59]	◐	●	◐	●	◐	◐
Security Code Scan [64]	◐	◐	◐	●	○	◐
SonarLint [69]	●	●	◐	●	○	◐
SonarQube [70]	●	○	◐	◐	●	●
SpotBugs [71]	◐	○	◐	○	◐	◐
VisualCodeGrepper [79]	○	○	◐	◐	◐	◐
VSDiagnostics [80]	◐	◐	◐	◐	○	◐

Clickable Trace or Path Projection. Johnson et al. mention the participants' demand for visual outputs supporting the warning [34]. Approaches for this would be clickable traces of the relevant intermediate steps or path projections. Similar to internal reasoning, these approaches hint at the main points of the issue and how they are connected with each other. One general approach is presented by Phang et al. [35]. In their study with FindBugs and Fortify SCA, Ayewah et al. [1] point to the benefits of path projection for the understanding and evaluation of a reported bug but also point to its shortcomings in potentially confusing the users.

Our evaluation shows that most of the considered tools do not support any related features. Two tools fulfill this criterion and three tools give at least little support in this regard, while 41 tools do not point to any intermediate steps in their warning messages. All CLI tools are evaluated negatively in this criterion. Again, this feature is not required for simple linting tools but is of high value in the evaluation of warnings of complex issues.

Explaining Consequences of an Issue. Intuitively, if developers are not informed about the consequences of a detected issue, they will probably care less about it, as the importance of the problem may remain unclear. In their interview and observation-based study, Thomas et al. [74] evaluate their interactive analysis tool ASIDE and discuss the necessity of explaining the issue's consequences. Before approaching a detected issue, it appears to be highly relevant to evaluate whether a found issue is indeed causing problems or exploitable. If this appears not to be the case, developers tend to ignore the issue. Therefore, explaining the consequences of an issue is not only important for understanding the issue, but also for the evaluation of whether the developer considers fixing it.

In our evaluation, the kind and degree of the expected explanation highly depend on the analysis type. For linters considering code style, the consequences are rather trivial. Bug finders or security scanners require more explanation since the context usually is more complex. While the majority of the considered tools (33 of 46) still does not explain the consequences of detected issues, 13 tools mention potential consequences (five *partially*, eight *yes* evaluations). Comparing CLI tools with the GUI tools shows that the fraction of tools not explaining the consequences is higher in CLI tools (86.3% compared to 71.7%).

Explaining by Example. Another approach in explaining a problem concept lies in illustrating it with examples. These examples might consider the formation of the issue, its type, consequences, or fix. Explaining the underlying issue with relevant background information using the example helps the developer to get a better understanding of the issue and how to fix it [34]. In a next step, Smith et al. [66] argue that providing any example is better than not doing it at all, but also that mismatched examples are another usability issue. When the hard-coded example differs too much from the original code, the developer has to figure out the transferable similarities between them.

In our evaluation, we mainly consider whether there is any example given for the explanation of the issue and the fix. We find that five tools offer reasonable and complete examples, two tools give at least some kind of example for illustration, whereas 39 tools do not use any kind of supporting example. This is worse for CLI tools, where only one tool offers a partial solution. While we agree with Smith et al. [66] that tool developers should aim for matching examples, our evaluation shows that as first step even more examples should be included in warning messages.

Offer Further Information. One pain point of insufficient warning messages often lies in the issue of presenting insufficient information [34, 66, 68]. Smith et al. [66] consider missing information regarding vulnerability prioritization and fix information, but also mention that the solution would not be to overload the notification with too much information. Johnson [31] argues that offering insufficient information leads to the developer searching other resources, while processing too much information is too time-consuming. Since the context and the developer's expertise are relevant as well [33], solution approaches might need to offer additional information to more details e.g. via links [34] or else adapt to the corresponding developer, e.g. using machine learning [32].

Almost one-third of the considered tools offer more information in warning messages (three *partially*, eleven *yes* evaluations). In many cases, a link to their website offers explanations about their different warning categories, more detailed information about the detected issue. Still, the ratio of tools offering more information is a little worse in CLI tools (one *partially*, three *yes* evaluations).

Warning Connection to Code. As mentioned above, Smith et al. [66] report issues related to reports that are disconnected from the code. Warnings often consist of mismatched examples or a main template text, where only names of the variables, method, etc. are changed. As errors from the same warning category with the same warning message still might be unique, the developer still has to process the message further, in addition to the other challenges the developer faces while understanding the warning. Thomas et al. [74] find that developers wish to integrate the warnings within the code and contextualize them further. Other developers desire for the use of more natural language [34]. To pursue natural explanations, Barik et al. [5] analyzed the communication on StackOverflow to find out how human developers explain code issues.

We consider a message as generic if the tool always states the same error message for a specific error or only adapts the names of parameters or methods. Accordingly, we only found three tools to really fulfilling a closer connection to the code base, while 43 tools are more generic. In many cases, the content of such messages is also very short, which makes it even more complicated to understand and apply the warning message to the code base. In these cases, the developer is roughly informed about the issue and knows where to find the issue. However, such generic messages often lack information and context and hinder the developer in understanding what exactly is wrong with the code.

Unique Weakness Identifier. If the developer would like to learn more about a specific code issue outside the SAT, a unique weakness identifier is valuable as it helps the developer recognize it or search for information. Security vulnerabilities are usually categorized according to the Common Weakness Enumeration (CWE).[2] In other contexts such categorization is not that clear, for instance in code style linters or non-security bug finders. Still, such categorization according to globally unique identifiers would improve the search for further information for specific issues outside the used tool.

[2]http://cwe.mitre.org

Our evaluation validates this description on unique identifiers: security SATs (ten tools) categorize issues according to the CWE, while this is difficult to achieve in other contexts. Three tools find partial solutions for unique identifiers, which are not as well-defined as CWEs but still helped with clear identifiers. Again, this aspect is not as often implemented in CLI tools as in other tools. This is probably connected to the higher amount of simpler tools (e.g., code style linter) in CLI tools as in more powerful and complex tools.

Error Information. In our last aspect related to warning messages, we evaluated whether the tools answer some of the central information categories that are relevant to understand, evaluate, and fix the issue. Therefore, we consider the categories of the description, existence of a general warning categorization, localization of the found issue, and whether there is a severity evaluation. The latter belongs to the most relevant factors while selecting warnings [78].

Though the implementation of an error description in the warning message might appear trivial, we find that there are quite some tools using very unclear warning messages. In these cases, there is a general warning text, but no description of what is wrong with the code or what the developer might want to change. Sometimes there is a description of a part, but no explanation of what is wrong about it (e.g., code style linting tools describing the length of a code line or outputting a number related to code complexity). In ten tools we did not find a description of a code issue and in eight cases the description is insufficient. Again, CLI tools stand out with a higher fraction of no descriptions or partial descriptions. Only about 40% give a satisfying error description in *CLI* tools, while this fraction lies around 60% for *GUI* tools.

Next, we evaluate whether the tool associates a unique warning category with every warning message. Compared to the unique identifier, no global unique identifier is required, but a rough classification of the issue types the SAT considers. Such categorization helps in our evaluation as it roughly hints to the error type and helps the developer to recognize and connect experienced warning messages over time. 32 of the considered tools apply clear error categories, as opposed to twelve tools not categorizing errors accordingly (two partially evaluations). The share of CLI tools fulfilling this criterion is somewhat smaller, but still more than half of the CLI tools use error categories (about 59% compared to about 69%).

In the next criterion, we evaluate whether the warning messages hint at the place of the issue. In case of issues covering several intermediate steps, we ignore intermediate steps as this is already covered above. Instead, we only consider the main location of the error and evaluate its granularity, using the evaluation grades *line of code*, *method*, *class*, and *project*. 40 tools hint at the exact *line of code*. Since there are some issue types that are not related to one specific line but rather to methods or classes (e.g., dependency checker), there are good reasons for a rougher localization.

Last, we check severity evaluations of warning messages. Severity supports the developer in the evaluation of a notification and for the prioritization of fixing errors. The distributions of tools fulfilling this criterion is somewhat balanced since 21 tools do not use severity classifications whereas 20 tools do (five *partially* evaluations). In CLI tools, the distribution is very different, where four tools fulfill this criterion and 18 do not.

Summary. Overall, our evaluation of the SATs' warning messages very much confirms the usability issues presented in the literature. For most of the categories, the majority of tools do not support the required features. Warning messages in CLI tools tend to be worse compared to GUI tools. We found too many tools with very short warning messages and hardly any documentation about detected issues and what might be wrong with the code. Then again, there are also good examples for all considered aspects. This leads to the assumption that there are known approaches to address most usability issues but only a few implementations of these approaches.

Main Findings: (1) More than half of the tools have too poor warning messages (30 of 46, see Table 2). (2) Three tools give good warning messages, which might inspire more tool developers. (3) Hardly any tools give information about why they evaluate something as an issue (giving traces or paths, internal reasoning). (4) Explaining the code issue with details exceeding the most basic aspects also remains a common problem.

3.2 Fix Support

The fix support category is related to warning messages as a good explanation already leads to potential fix ideas. Heckman et al. [26] try to classify alert messages as actionable and unactionable using machine learning techniques and continue to synthesize available research results about main approaches for actionable alert identification techniques in a systematic literature review [27]. Accordingly, SATs should not only inform the user about potential issues but also add further information on how to fix the issue. Sadowski et al. [61] affirm the challenge of unactionable alerts: One of their main lessons learned is that tools should not just find bugs, but also fix them. Smith et al. [66] evaluate four security-oriented SATs and conclude that SATs do not support the developer well enough to fix the issues, which even might lead to the abundance of SATs. Therefore, we sought to analyze the fix support features of the considered tools.

Quick Fix. Regarding quick fixes, the scientific literature points to two main issues: the lack of implemented quick fixes and the lack of trust in existing quick fixes. Nguyen Quang Do et al. [20] state that quick fixes are one of the most popular features of SATs, based on their survey of developers. Johnson et al. [34] find the lack of or ineffectively implemented quick fixes as the most frequently mentioned difficulty in their study. This lack of effectively implemented quick fixes may diminish trust: some developers trust their own solutions more than solutions offered by the tool [74].

We considered it too difficult in our context to evaluate whether offered quick fixes are implemented too ineffectively. Hence, we only check whether the tool offers quick fixes at all. We find that 36 tools do not offer any fixes to the developer (see again Table 2). Only one of 22 CLI tools offers quick fixes. While this shows that the overall number of tools not offering quick fixes is too high, almost half of GUI tools offer some partially or fully satisfying quick fixes. While it is technically challenging to offer quick fixes, especially for more complex issues, these numbers show that it is still possible to offer some help to the developer, but also that we can still do better by offering more fix support.

Alternative Solutions. In many cases, there is not only one possible solution for a detected issue. Depending on the developer's preferences, the development contexts, and other aspects, one solution might be better fitting than other solutions. Therefore, understanding alternative fixes and approaches is one of the main questions, developers ask while diagnosing potential security vulnerabilities [67, 74]. Instead of quick fixes, Barik et al. [7] propose the idea of slow fixes, in which the developers are supported in exploring different solutions and balancing the benefits of manual and automated fixing.

In our evaluation, we find that only four tools consider alternative solutions. Two of these tools are evaluated as partially fulfilling as they offer alternative solutions in some cases and only one option for other issues. None of the CLI tools consider alternative solutions. This shows that even though there is an interest from the developers to consider alternative solution approaches, this is not implemented by hardly any tool developers.

Fix Preview. Connected to the need of exploring the space of possible solutions is the question of whether offered fixes are applicable in the given code context. Due to mentioned trust issues towards quick fixes, developers might be afraid that the fix might break the code in other parts [74]. Therefore, a fix preview helps evaluate whether the fix causes undesired side-effects and to assess the application of the fix in the given context [67]. Johnson et al. [34] also support that developers would prefer fix previews instead of directly applying the fix.

Just like the previous criterion, none of the CLI tools offers a fix preview. While there are few other tools implementing this feature (one *partially*, four *yes* evaluations), this only constitutes a small fraction of the overall considered tools (about 11%).

Fix Example. Corresponding to the explanation of the issue in warning messages using examples, examples also might be used to explain and illustrate solution approaches. Similar to fix previews, Johnson et al. [34] report that some participants of their study would prefer the use of examples to get a better understanding of how to fix the problem. Hartmann et al. [25] state the hypothesis that relevant solution examples help novices to interpret and correct error messages.

In our evaluation, we find that 21 of 22 CLI tools do not present any fix examples to the developer. Similar to *Fix Preview*, five of the remaining GUI tools provide fix examples to the developer, but this represents a small fraction as well.

Fix Tutorial. From the perspective of a warning message, explaining all necessary steps the developer has to take to fix the issue appears to be relevant and useful. Nachtigall et al. [42] present the idea of an interactive system between the SAT and the user, in which the tool takes the role of an assistant and of a teacher. The teacher guides the developer through fixing all detected issues and gives all necessary information at every state of the fix. Johnson et al. [32] aim to model the user's knowledge and adapt the tool. The tool would present the required knowledge accordingly and might support the user with the next steps to fix an issue.

As before, only one CLI tool gives partially satisfying instructions on how to fix the reported issue. With regard to the remaining GUI tools, there are five more tools (one further *partially*, four *yes* evaluations) giving more detailed guidance on how to fix the issue.

However, those tutorials are not comparable to for instance the assistant discussed before.

Summary. The *Fix Support* category appears to be a much neglected area. This may be due to technical challenges in the implementation of corresponding features. Yet, the literature shows that giving more fix support is a highly relevant task for tool developers. Only reporting potential issues is insufficient, if the developer is not able to fix them afterwards. Throughout our considered criteria, CLI tools offer almost no support. GUI tools reach somewhat better results. Nonetheless, the numbers of tools fulfilling these criteria are very low. Not every tool needs to fulfill every aspect since these approaches aim for the same goal. Overall, our evaluation exposes definitive challenges for future work.

Main Findings: (1) With 37 tools giving almost no fix support, this appears to be the most neglected category. (2) Three tools give sufficient fix support. (3) Only ten (partially) offer quick fixes. Even fewer tools support this with alternative solutions or previews.

3.3 False Positives

One major reason why developers do not use SATs is the high amount of false positives. Johnson et al. [34] discuss the consequences of when there are too many false positives. Accordingly, false positives outweigh true positives and lead to dissatisfaction. Christakis et al. [12] stress that high false positives rates lead to disuse of the analysis. Furthermore, an analysis of static analysis tool alerts related questions on StackOverflow reports that false positives belong to the most prevalent topics [29]. As the existence and occurrence of false positives mainly is a technical problem related to the implementation and we focus on usability aspects, we consider how developers might deal with false positives.

Suppression of False Positives. One mechanism to avoid some false positives lies in suppressing false positives. By telling the tool that a specific warning is a false positive, the tool might ignore reporting the same issue again or even use this information for a user feedback-based ranking of warnings [40]. The high relevance of suppression mechanisms for false positives is stressed in several publications [2, 12, 34].

In the evaluation of this criterion, we only consider explicit ways of reporting false positives to the tool. Using implicit mechanisms like code annotations are valid as well as desired by developers, but will be considered in the temporal suppression criterion as code annotation is used in more contexts than for false positives. We find that overall twelve of the considered tools offer the possibility to explicitly report false negatives, see again Table 2. Only two of these twelve tools are CLI tools.

Confidence Evaluation. SATs often add further metrics to the warning message for the evaluation whether it is a false positive, one of which is a confidence evaluation [20]. Tools such as FindBugs associate a confidence factor to each warning [2], which allows the developer to filter by confidence and hence support the developer in focusing on relevant warnings.

In our evaluation, we find that six tools offer such confidence evaluation, including three CLI tools. Three of these six tools are extensions of FindBugs, which already offers confidence evaluation before extending it. This emphasizes how rarely this mechanism is considered overall.

Ask for User Knowledge. Sometimes the analysis has incomplete knowledge about the context, in which the analyzed code is executed and erroneously reports an issue due to the analysis' overapproximation. To avoid this, Zhu et al [87] propose to explicitly ask for the developer's knowledge in the context of access control. This approach might also be transferable to different contexts. Furthermore, asking for user knowledge might help in modeling the user's knowledge and use this information to adapt the tool to the developer's preferences [32]. Tripp et al. [77] present a similar approach, where the tool applies feedback and statistical learning to improve reports. As mentioned above, a SAT also might collect the user's feedback on whether a report is a false or true positive for future reports [40].

We evaluate eight tools to partially fulfill this criterion, while all remaining 38 tools offer no related features. Again, the evaluation is worse for CLI tools, where no tool asks for the user's knowledge. While this feature might be less prioritized than other features, our results indicate that this concept idea is rather in early steps and is hardly used in practice.

Summary. Our observations in the category of false positives indicate that usability-related mechanisms are hardly used to avoid false positives. For each of the considered criteria, there are tools considering them, but overall these mechanisms are implemented too rarely. Considering the high impact of false positives on the developers' satisfaction, this poses many tasks for future work.

Main Findings: (1) Only twelve tools let the developers explicitly report a warning as a false positive. (2) Overall, 15 tools fulfill the considered criteria partially or better, while 31 are weak in this aspect.

3.4 User Feedback

The category of how false positives might be handled from the perspective of usability is related to the perspective of the user's feedback. Hence, some examples of how user feedback might be integrated in SATs are mentioned above. In this section, we consider some further, more general use cases, in which the integration of the user's feedback is relevant. The more the analysis adapts to specific users, the more useful it will appear to them [31, 32].

Customizability of Analysis Rules. Johnson et al. [34] point out that customizability, in general, is a problem why developers do not use SATs, including analysis rules but not restricted to them. Nguyen Quang Do et al. [20] confirm this demand for the customization of analysis rules. Not all rules should be turned on by default and the process of selecting rules should be made easy [12].

In our evaluation, we check whether the tools allow the developer to *turn specific analysis rules off*, *modify existing rules*, or *introduce own rules*. These options are not mutually exclusive and allow for multiple selections. By doing this, we find that 19 tools do not offer any customization of the analysis rules. With 14 tools the majority of these tools are CLI tools. The remaining 27 of all considered tools offer the option to switch off specific rules. 19 of these 27 tools also allow to modify existing rules and 17 of them allow to introduce own rules. These fractions indicate a positive tendency towards allowing customizability of analysis rules.

Validation of True Positives. As seen above, it is not only useful to report false positive to the tool, but also the validation of true positives for user feedback-based self-adaptive ranking [40]. Accordingly, warning types with a higher fraction of true positives and fewer false positives might be higher prioritized.

Regarding the 46 selected tools, only two tools offer explicit validation of true positives, none of which are CLI tools. In other cases, this might only be done implicitly by fixing the error or by assigning a responsible developer for fixing it. However, this information is not used for validation purposes in any further tools, according to the corresponding tools' documentation.

Temporal Suppression. The relevance of suppression mechanisms for false positives is described above. Many developers prefer to do this via code annotations [12, 34]. Code annotations allow for temporal suppression, which is relevant when the corresponding code part is still under development and warnings would rather be distracting than supporting [34]. The implemented code annotations usually do not refer to false positives, but just generally ignore a line of code or a whole code section.

In our evaluation, we find 19 tools to find described features, while 27 tools do not offer temporal suppression, see again Table 2. Eight of these 19 tools are CLI tools, which makes up for a comparable share between CLI tools and GUI tools.

Filter Alerts. An evaluation of Stack Overflow questions about SAT alerts shows that ignoring / filtering alerts is the most discussed issue [29]. This shows that developers try to use these features but also that there are challenges in using these features.

In ten tools we find the option to filter alerts. Four further tools partially support this feature, which means that these tools generally offer filters but are limited to very few filter options. In contrast to this overall result, 21 of 22 CLI tools do not offer filtering alerts.

Summary. Overall, we find different results in this category. On the one hand, there are hardly any tools offering implicit validation of true positives, which would allow for adapted error reports. On the other hand, the majority of tools allows customization of analysis rules, with different options. The fraction of tools supporting alert filters and temporal suppression is somewhere in the middle, neither too small nor high enough.

Main Findings: (1) Eight tools strongly support the integration of user knowledge, while ten tools partially fulfill it. (2) The majority of tools allows to switch off rules, while 19 tools also allow further modification of the analysis rules. (3) 19 tools support temporal suppression of warnings. (4) Filtering alerts and validating true positives are implemented in a clear minority of tools.

3.5 Workflow Integration

The integration of the SAT's support into the developer's natural workflow is one of the main challenges of usable tool support and one of the main reasons why many users are dissatisfied [9, 34, 46]. As workflow integration is a key aspect for the use of SATs [62], Sadowski et al. [61] conclude that workflow integration should be pushed as early as possible.

Warning Prioritization. The larger the analyzed code base, the more warnings might be reported. In real-world projects, the amount of warnings often is too high to process for a human developer. Hence, developers demand automatic warning prioritization by the tool [12, 20], so that they know which bug to select. Many tools fail in providing enough information in this aspect [66], which again leads to the user's dissatisfaction.

Twelve tools report the detected issues in a prioritized order. Two more tools also consider prioritization but are rather rough by just offering a few priorities. The remaining 32 do not prioritize their warnings. Regarding CLI tools, three are tools fulfilling this criterion, while 19 do not. As warning prioritization requires some kind of evaluation of priorities, this requires more complex processing. Still, this should be worth the effort as it makes the SAT effective [34].

IDE Integration. The IDE or more powerful editors are the standard development environment. As this might be the closest connection to the developer's workflow, SATs should be integrated in IDEs to offer better workflow integration [20, 61].

Almost half of the considered tools (20 of 46) offer plugins for one or more IDEs in the respective programming language, see again Table 2. The one reported CLI fulfilling this criterion offers some plugins, which we were unable to install. Some IDEs offer unified tool integration (like IntelliJ), while other IDEs allow more options in offering and accessing the tool's features (like Eclipse).

Standalone Tool. In this criterion, we evaluate another type of tool's accessibility and check whether it is possible to use it as a standalone tool. Therefore, we inspect whether it is possible to install, configure, and execute the analysis on its own. We find that nearly all tools work as standalone tools as well. Only six tools do not fulfill this criterion, for instance, tools that can only be integrated with maven or other build functions.

Browser Access. Furthermore, there are tools accessible in typical browsers, which includes only services as well as locally hosted analysis servers, accessible via browser. Four of the evaluated tools offer this access type. While this option is not directly integrated into the workflow, this setting usually allows for more presentation possibilities of the analysis.

Disjoint Process of Understanding and Fixing. Offering different access types is of advantage as the developer might choose according to their own preferences. However, options that are not directly available in the developers' coding environment lead to a disjoint process of understanding and fixing reported issues [34]. This means that the developer has to open tool reports to see and understand the warnings, while they have to switch to the coding environment afterwards to see and work on the corresponding code. As this process is ineffective and disruptive, developers tend to not use such tools [34].

Overall, the evaluation of this criterion completely overlaps with the question of whether the tool supports IDE integration, including respective editors. Since all evaluated IDE integrations allow to display the warning message and the corresponding code part at the same time, the process of understanding and fixing the issue is not disjoint. In all remaining tools, this process is disjoint as the warnings are reported outside the coding environment.

Runtime of the Analysis. The longer the runtime of the analysis the more difficult it is to integrate the analysis into the developer's workflow. Johnson et al. [34] report in their study that developers complain about analyses taking too much time, which disrupts the flow. Accordingly, analyzing a larger code base is even more problematic as it requires more time. These problems might be slightly avoided by running the analysis at different stages of the software development lifecycle, like during nightly builds or at major project milestones [20]. While these options are valid, they are also not integrated into the developer's workflow as well as running the analysis at coding time.

In our evaluation, we tried to use comparable projects within each considered language. For some tools, we were not able to run the analysis on the complete project but only on one directory or even one file, because of the otherwise too high runtime. While this makes comparison harder, it points to another usability issue. Independent from the exact size of the analyzed code, the resulting runtime still allows for an evaluation of whether a corresponding tool might be useful during coding time or only during nightly builds or at major project milestones. Around half of the tools required less than a *minute* for their analysis (22 tools). This time most likely makes sense to be executed during coding time. Twelve tools require *more than one minute*, nine tools *more than five minutes*, and one tool each for *more than ten*, *15*, and *30 minutes*. Five tools were not evaluated in this criterion as we were unable to execute the analysis on the whole project.

Feedback on Fixed Issue. After spending some time fixing reported issues, the developer might require feedback about whether these issues are really solved. Giving specific feedback about fixed issues gives confirming and motivating information to the developer, which might keep them engaged and more satisfied with tool's overall experience [44].

In our evaluation we only find two tools giving explicit feedback about fixed issues. For all remaining tools, the developer has to remember the warning, rerun the analysis, and check whether the issue is still reported. Hence, these results show that this idea is hardly used in practice and rather less prioritized by tool developers, even though such feedback would support and motivate the developer's work.

Summary. Concluding the category of the SAT's *Workflow Integration*, our evaluation confirms several challenges from the literature. Offering different ways to access a tool is a good way to adapt to the developer's preferences. Our results show that a high amount of tools is only available as CLI tool. As shown through our whole evaluation, CLI tools are weaker than the other options in most usability-related aspects. It should be the goal to get as close to the developer's natural workflow as possible, which is reached through IDE integration. This is done by roughly half of the considered tools. Also, the aspects of warning prioritization and feedback on fixed issues pose open challenges for SAT developers.

Main Findings: (1) 22 of 46 tools are only available via CLI, which is highly problematic for the workflow integration. (2) 22 of the tools also need more than one minute for the analysis, which might already be too long for the integration into the normal coding process. (3) Overall, we evaluate 36 tools to partially allow workflow integration, while only two tools stand out positively.

3.6 User Interface

In this final main category, we evaluate how the tool generally presents all of its interaction options to the developer. While the user interface is not the core of static analysis, it might still distract or support the developer. To support developers in their work with SATs, there are different approaches for improving the user interface [40]. In the following, we evaluate central aspects of supportive user interfaces.

Highlighting in Code and Warning Icons. Highlighting the error in the code view and adding warning icons are commonly known mechanisms to draw the developer's attention to code issues. While only highlighting code would not be enough, it is still useful for many developers as are warning icons [43]. Apart from error locations, highlighting and icons might also be used for other related code locations, such as intermediate steps of how the error evolves through the code, locations where the analysis might require the developer's knowledge or wherever the tool might want to guide the developer's attention [38, 85].

16 of the considered tools highlight errors in the code and twelve tools integrate warning icons, see again Table 2. At first sight, these fractions might appear small, but they are downgraded by the observation that no CLI tool supports either code highlighting or warning icons. Hence, the fraction of GUI tools supporting code highlighting is two-thirds, while this fraction is little lower for warning icons.

IDE or CLI Conformity. In their handling of SATs, developers have specific expectations based on experience and on other tools communicating in similar contexts [33]. Accordingly, developers expect the tool to behave similarly to how they know it. Therefore, we evaluate whether the considered tools report their finding in a similar way like other tools or compilers. While doing this, we also try to maintain the context of IDE or CLI presentation of warnings. We evaluate that overall eight tools deviate from the typical communication patterns, five of which are CLI tools.

Overview of Warnings. In huge code bases with many detected issues, it is difficult for the developer to get an overview of main results of the analysis. To support the developer in doing this, several SATs process the overall results in dashboards or similar warning overviews [61]. Nguyen Quang Do et al. [20] support the demand and value of dashboards and report that the developers often look at it to understand the project's health before considering the single warnings.

Eight of the evaluated tools offer a dashboard or a comparable overview. The remaining tools only present the list of all warnings to the developer, sometimes adding a few simple metrics. Though CLI tools are limited in their graphical expressiveness, still, five CLI tools offer comparable overviews.

Tracks Progress. Many tools only report currently existing issues found in the code base, but do not refer to older analysis runs to report fixed issues [66]. Hence, developers demand features keeping track of the progress, which also give a clear view of what is achieved [20].

In our study, we only find two tools offering such features, while this is not the case for the remaining 44 tools. In the latter case, the developer needs to manually keep track of the progress.

Search through Warnings. When the list of reported issues is too long to be overlooked by developers, they might want to search through the warnings. Using different metrics, the search might support the developer in finding relevant issues to fix, for instance issues the developer has specific knowledge about. Based on comparable use cases, developers confirm the demand of features searching through warnings. We find ten tools offering searching features, none of which are CLI tools.

Summary. Overall, our evaluation of user interface-related criteria yields mediocre to weak results. Most of the GUI tools guide the developer's attention using code highlighting and warning icons, while this is ignored by CLI tools. Most tools also present their results in typically used patterns. Still, the last three criteria also pose several tasks for future work since only few tools offer sufficient warning overviews, progress tracking, and search functions.

Main Findings: (1) Three tools stand out with good and supportive user interfaces. (2) Code highlighting and warning icons are completely neglected in CLI tools. (3) Searching through warnings and tracking progress are highly neglected over all tools (ten and two tools fulfill them, respectively). (4) 26 tools support at least few of the considered criteria, while three tools get a good overall evaluation in this category.

4 THREATS TO VALIDITY

We are aware, and sought to mitigate, the following threats to the validity of our study.

For tools supporting multiple operating systems, there is a bias towards the Windows version. Only if we were unable to install a tool on Windows did we try to install it on the Linux VM (where Linux versions were available). We see this bias as unproblematic, however, because Windows still has, by a large margin, the largest market share on desktop systems [72].

Overall, the evaluation process of all tools took more than one year. We always tried to evaluate the most current version of the tool, depending on the time of when we considered respective tools. This might lead to the issue that some tools we considered earlier might have been updated in the meantime. If these updates affected features that have been evaluated in an earlier version, our evaluation might slightly become obsolete in respective criteria. We see no way to avoid this.

Although the evaluation criteria and grades were established in advance of the study, the exact delineation between grade levels can be debatable in individual cases. Hence, it might happen that we assigned different grades to comparable fulfillment of a respective criterion in some cases. To mitigate this problem to the largest extent possible, we used a multiple-raters approach (see section 2).

While we try to evaluate the current state of SATs, it appears infeasible to collect a complete set of all relevant tools. As described above, we attempted to collect a representative set of tools in order to be able to obtain the most meaningful results possible. Nevertheless, our collection might miss a few relevant tools.

The same also holds for our selection of evaluated criteria. Literature surveys with a different methodology might result in further relevant aspects related to usability. When determining the list of criteria, we sought to find a good trade-off between feasibility and a meaningful representation of relevant usability aspects.

As described above, several tools have been excluded from the evaluation as we were unable to install the respective tool. We tried to install the tools with reasonable effort, depending on the documentation and our own knowledge. While we were unable to install these tools this does not generally mean that it is impossible to install them.

5 RELATED WORK

Much related work was already discussed above. Overall, many publications mention usability issues with SATs, yet some seek to give a wider and more complete view on the usability of SATs. We focus on such studies here.

Johnson et al. [34] research why SATs are not widely used by developers and how this might be improved, also discussing further implications. Their results are based on interviews with developers.

Christakis et al. [12] discuss what developers want and need from program analysis by surveying developers across Microsoft. They mainly focus on the following three aspects: 1) barriers and reasons, why developers stop using analysis, 2) functionality the developers want in analysis, and 3) non-characteristic functional characteristics a program analyzer should have.

Based on a survey within Software AG, a leading international software company, Nguyen Quang Do et al. [20] analyze the analysis tools integration in the development environment, the usage context of analysis tools, developers strategies in working with analysis tools, and features the analysis tool should provide.

These publications are related in that they seek to find usability issues and reasons why developers would use or reject SATs. This paper, however, rather seeks to use criteria mentioned in these papers to evaluate the current state of usability in SATs.

Nachtigall et al. [42] summarize the main design challenges for building usable SATs and very roughly evaluate 14 tools on the introduced six main challenges. While we revolve around their six main challenges, we split these categories up into more detailed criteria and also execute a more thorough evaluation of 46 tools.

Smith et al. [66] execute a heuristic walkthrough evaluation of four tools. In this approach, they familiarize themselves with the tools, like in a cognitive walkthrough. In the next step, they explore the system using a set of usability heuristics, like in heuristic evaluations. They reveal several usability issues and group them into six themes. While their work is restricted to four tools, we evaluate a larger set of tools.

6 CONCLUSION

This paper presented a large-scale assessment of user interactions offered by SATs. As the scientific literature points to many usability issues related to SATs, we sought to evaluate to what extent these aspects generally apply to the current state of the art. Therefore, we defined a set of relevant criteria, collected relevant SATs, and applied the criteria to the tools. Our tool-based perspective is complementary to the mainly survey- and interview-based view from previous work, yet empirically confirms many previous findings and suspicions.

Therefore, our results state a huge future work for SAT builders but also reveals the need for further research in the area of static analysis. Overall, the areas of explaining warning messages, fix support, handling of false positives, consideration of user feedback, integration into the developer's workflow, and supporting user interfaces are still much neglected. For CLI tools these challenges are even more problematic in all categories. This leads to the developer's dissatisfaction and might make them abandon support from static analysis.

ACKNOWLEDGMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297

REFERENCES

[1] Nathaniel Ayewah and William Pugh. 2008. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*. 1–5.

[2] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[3] Bandit. 2021. https://github.com/PyCQA/bandit.

[4] Titus Barik. 2016. How should static analysis tools explain anomalies to developers?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 1118–1120.

[5] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How should compilers explain problems to developers?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 633–643.

[6] Titus Barik, Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill. 2014. How developers visualize compiler messages: A foundational approach to notification construction. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 87–96.

[7] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. 2016. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 211–221.

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

[9] Eric Bodden and Lisa Nguyen Quang Do. 2018. Explainable static analysis. *Software Engineering und Software Management 2018* (2018).

[10] OWASP Dependency Check. 2021. https://owasp.org/www-project-dependency-check/.

[11] Checkstyle. 2021. https://checkstyle.sourceforge.io.

[12] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 332–343.

[13] CogniCrypt. 2021. https://www.eclipse.org/cognicrypt/.

[14] Cppcheck. 2021. https://cppcheck.sourceforge.io.

[15] Cpplint. 2021. https://github.com/cpplint/cpplint.

[16] CScout. 2021. https://www.spinellis.gr/cscout/.

[17] DevSkim. 2021. https://github.com/microsoft/DevSkim.

[18] Dlint. 2021. https://github.com/dlint-py/dlint.

[19] Lisa Nguyen Quang Do and Eric Bodden. 2020. Explaining static analysis with rule graphs. *IEEE Transactions on Software Engineering* (2020).

[20] Lisa Nguyen Quang Do, James Wright, and Karim Ali. 2020. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering* (2020).

[21] ESLint. 2021. https://eslint.org.

[22] Fb-contrib. 2021. http://fb-contrib.sourceforge.net.

[23] FindSecurityBugs. 2021. https://find-sec-bugs.github.io.

[24] Flawfinder. 2021. https://dwheeler.com/flawfinder/.

[25] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1019–1028.

[26] Sarah Heckman and Laurie Williams. 2009. A model building process for identifying actionable static analysis alerts. In *2009 International Conference on Software Testing Verification and Validation*. IEEE, 161–170.

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

[28] Hegel. 2021. https://hegel.js.org.

[29] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. 2019. Challenges with responding to static analysis tool alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 245–249.

[30] InferSharp. 2021. https://github.com/microsoft/infersharp.

[31] Brittany Johnson. 2015. Adapting program analysis tool notifications to the individual developer. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 291–292.

[32] Brittany Johnson, Rahul Pandita, Emerson Murphy-Hill, and Sarah Heckman. 2015. Bespoke tools: adapted to the concepts developers know. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 878–881.

[33] Brittany Johnson, Rahul Pandita, Justin Smith, Denae Ford, Sarah Elder, Emerson Murphy-Hill, Sarah Heckman, and Caitlin Sadowski. 2016. 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*. 73–84.

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

[35] Yit Phang Khoo, Jeffrey S Foster, Michael Hicks, and Vibha Sazawal. 2008. Path projection for user-centered static analysis tools. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 57–63.

[36] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology* 51, 1 (2009), 7–15.

[37] Barbara Ann Kitchenham. 1996. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes* 21, 1 (1996), 11–14.

[38] Andrew J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.

[39] McCabe. 2021. https://github.com/PyCQA/mccabe.

[40] Tukaram Muske and Alexander Serebrenik. 2016. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 157–166.

[41] Mypy. 2021. http://mypy-lang.org.

[42] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Explaining Static Analysis-A Perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 29–32.

[43] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. 2017. A stitch in time: Supporting android developers in writingsecure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1065–1077.

[44] Lisa Nguyen Quang Do and Eric Bodden. 2018. Gamifying static analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 714–718.

[45] NodeJSScan. 2021. https://github.com/ajinabraham/nodejsscan.

[46] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. 2018. Myths and facts about static application security testing tools: an action research at Telenor digital. In *International Conference on Agile Software Development*. Springer, Cham, 86–103.

[47] PMD. 2021. https://pmd.github.io.

[48] Error Prone. 2021. https://errorprone.info.

[49] Pycodestyle. 2021. https://pycodestyle.pycqa.org/en/latest/.

[50] PyDev. 2021. https://www.pydev.org.

[51] Pydocstyle. 2021. http://www.pydocstyle.org/en/stable/.

[52] Pyflakes. 2021. https://pypi.org/project/pyflakes/.

[53] Pylint. 2021. https://pylint.org.

[54] Pyre-Check. 2021. https://pyre-check.org.

[55] Pyright. 2021. https://github.com/microsoft/pyright.

[56] Pytype. 2021. https://github.com/google/pytype.

[57] Radon. 2021. https://pypi.org/project/radon/.

[58] Reshift. 2021. https://www.reshiftsecurity.com.

[59] Roslynator. 2021. https://github.com/JosefPihrt/Roslynator.

[60] Frank E Saal, Ronald G Downey, and Mary A Lahey. 1980. Rating the ratings: Assessing the psychometric quality of rating data. *Psychological bulletin* 88, 2 (1980), 413.

[61] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66.

[62] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 598–608.

[63] Puma Scan. 2021. https://www.pumascan.com/.

[64] Security Code Scan. 2021. https://security-code-scan.github.io.

[65] Semgrep. 2021. https://semgrep.dev.

[66] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. 2020. 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)*. 221–238.

[67] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 248–259.

[68] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2018. How developers diagnose potential security vulnerabilities with a static analysis tool. *IEEE Transactions on Software Engineering* 45, 9 (2018), 877–897.

[69] Sonarlint. 2021. https://www.sonarlint.org.

[70] SonarQube. 2021. https://www.sonarqube.org.

[71] SpotBugs. 2021. https://spotbugs.github.io.

[72] statcounter. 2021. https://gs.statcounter.com/os-market-share/desktop/worldwide.

[73] JavaScript Standard Style. 2021. https://standardjs.com.

[74] Tyler W Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson Murphy-Hill. 2016. What questions remain? an examination of how developers understand an interactive static analysis tool. In *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*.

[75] TIOBE. 2021. https://www.tiobe.com/tiobe-index/.

[76] Ayal Tirosh, Mark Horvarth, and Dionisio Zumerle. 2019. *Magic Quadrant for Application Security Testing*. Technical Report. Gartner, Inc.

[77] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 762–774.

[78] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 38–49.

[79] VisualCodeGrepper. 2021. https://github.com/nccgroup/VCG.

[80] VSDiagnostics. 2021. https://github.com/Vannevelj/VSDiagnostics.

[81] Vulture. 2021. https://pypi.org/project/vulture/.

[82] wemake-python styleguide. 2021. https://wemake-python-stylegui.de/en/latest/.

[83] Wily. 2021. https://github.com/tonybaloney/wily.

[84] Xenon. 2021. https://xenon.readthedocs.io/en/latest/.

[85] Jing Xie, Bill Chu, Heather Richter Lipford, and John T Melton. 2011. Aside: Ide support for web application security. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 267–276.

[86] xo. 2021. https://github.com/xojs/xo.

[87] Jun Zhu, Bill Chu, Heather Lipford, and Tyler Thomas. 2015. Mitigating access control vulnerabilities through interactive static analysis. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*. 199–209.