Explaining Static Analysis – A Perspective

Marcus Nachtigall
Paderborn University
Germany
marcus.nachtigall@upb.de

Lisa Nguyen Quang Do
Paderborn University
Germany
lisa.nguyen@upb.de

Eric Bodden
Paderborn University and Fraunhofer IEM
Germany
eric.bodden@upb.de

***Abstract*—Static code analysis is widely used to support the development of high-quality software. It helps developers detect potential bugs and security vulnerabilities in a program's source code without executing it. While the potential benefits of static analysis tools are beyond question, their usability is often criticised and prevents software developers from using static analysis to its full potential. In the past decade, researchers have studied developer needs and contrasted them to available static analysis tool functionalities. In this paper, we summarize the main design challenges for building usable static analysis tools, and show that they revolve around the notion of *explainability*, which is a subarea of usability. We present existing analysis tools and current research in static analysis usability, and detail how they approach those challenges. This leads us to proposing potential lines of future work in explainability for static analysis, namely turning static analysis tools into *assistants* and *teachers*.**

***Index Terms*—Program analysis, Static analysis, Explainability, User experience, Program understanding**

I. Introduction

In practice, static analysis tools are often used to support software developers to detect and resolve software bugs and security vulnerabilities. In recent years, static analysis research has introduced increasingly complex analysis methods and tools that support a growing number of programming languages, libraries, and coding concepts, returning results faster and with better precision. After analyzing the code, warnings are reported during coding, nightly builds, or as automatically generated comments in the code review phase, pointing out potential weak spots to the developer. Such results are promising for potential uses in industry. However, end-user experience has shown a constant set of usability issues throughout the last decade, such as ill-explained warning messages, motivating the need for understanding how to design tools that satisfy the developers, and overcome the gap between the academically perceived potential of static analysis and its use in practice.

In this paper, we briefly summarize the usability issues encountered by static-analysis users identified by recent research. We group them into six main challenges that revolve around more understandable static analysis warnings, actionable messages to fix those warnings, and the integration of static analysis tools in the developer's workflow. Those challenges point towards the direction of a better explainability, by explaining why the warnings are reported and what should be done to fix them, in a way that is as effortless as possible to the developer.

We then turn to the current state-of-the-art with seven static analysis tools used in industry and seven research prototypes, and show how they address those challenges. We thus highlight the limitations of current tools (e.g., non-responsive tools, or generic warning messages) and potential avenues for research in the area of explainable static analysis. We take the last point further by proposing several directions for future research, in particular through lifting the analysis into a real-time assistant–teacher system that learns from the developer and the code base, and provides appropriate explanations about analysis warnings and how to fix them, when needed.

II. Explainability Challenges

Recent research in static-analysis usability indicates common usability issues that cause warning misunderstandings and mishandlings and bad user-experience of current static analysis tools [1]. We have surveyed research publications in the past decade on the topic, and summarize the usability issues they report, grouped into six challenges: **C1**–**C6**.

***C1** Understandable Warning Messages:* The goal of warning messages is to provide the developer with enough information to judge whether or not they care about the warning, and if yes, to determine how to fix it. As the analysis rules can be complex and sometimes counter-intuitive, warning messages should explain what the bug or vulnerability is and why the tool reports it [2]–[4]. Many tools present generic messages, e.g., "SQL injection from line 326 to line 459". Other tools provide more detailed explanations, giving more information of the relevant analysis steps, e.g., "SQL injection from line 326 to line 459 along path X if the value of t is 3". Research advocates for more descriptive warning messages, in particular, information adapted to the coding context and to the analyzed code [1].

***C2** Fix Support:* Analysis warnings often involve security-specific knowledge developers may not have. As a result, after explaining the warning, a useful static analysis tool should support the developer in resolving it [4], [5]. In many cases, not knowing how to fix a warning leads to wrong fixes and developer frustration. A commonly requested solution lies in offering quick fixes [1]. Alternative approaches consist in finding fitting solutions for the specific context of each warning, for example, with slow fixes [6], which apply the fix step-by-step and enable customization.

***C3** False Positives:* A major reason of tool abandonment lies in the large amount of false positives generated by static analyses [1], [3], [5], which require intensive manual work from the developer. Potential solutions include suppression

978-1-7281-4136-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ASEW.2019.00023

mechanisms [5] or confidence indications [7]. In many cases, false positives are due to missing knowledge about the coding context [8]. Integrating specific context information could reduce the amount of false positives.

***C4** User Feedback:* User knowledge is valuable to provide correct analysis results. Processing and integrating user feedback can thus improve user-experience [1], [9], by, for example, learning false positive patterns and removing potential false positives from the reports. User feedback can be gathered through warning rejection (suppression), ignorance, or confirmation (fixing), or more explicitly through dialogues [8]. Learning more from the user and the coding context can lead to more fitted analysis results, less false positives, and a better overall user-experience.

***C5** Workflow Integration:* If the analysis is a disjoint process from the developer's main tasks, they tend to not use it [1]. Therefore, it is crucial to integrate the analysis into the development process and the developer's working environment [1], [2], [10]. Workflow integration raises further challenges about when analyses should be executed, where and how the results should be, or tool responsiveness [5], [7].

***C6** Specialized User Interface:* User interfaces support users in communicating with the tools. Determining when and how to display or query information from the user without overwhelming or boring them is a major challenge [4].

We see that the six challenges revolve around user-tool communication, with the analysis explaining warnings and fixes to the user (**C1**–**C2**), and the user explaining coding context and providing feedback to the analysis (**C3**–**C4**), in a non-disruptive, useful way (**C5**–**C6**).

III. State-of-the-Art Static Analysis Tools

In this section, we present different types of static-analysis tools: command-line tools (CLI), standalone tools, IDE tools, and tools with multiple interfaces. We present examples of such tools used in industry, which we refer to as **T1**–**T7**, and in research, referred to as **T8**–**T14**. Table I lists the tools, their types, and how they address the six challenges from Section II.

Command-line tools: give the least support with respect to the challenges, since they do not provide a graphical user interface (GUI). The only way to communicate with those tools lies in basic commands, and their output tends to be sparse, for readability purposes. Developers thus need to rely on external sources to understand the tool's output and the potential options for action. Therefore, command-line tools are more suited to reporting simple rules (e.g., linters), which warnings and fixes can be explained easily. While this design does not weaken the theoretical value of the analysis, it definitely restricts its practical use.

Standalone tools: overcome few of these boundaries because they provide a GUI, offering more options for action and providing a clearer representation of the warnings and the developer's current working status. Warning messages can be longer and more easily illustrated, and the tool can provide fix support. Still, the fixing process is separated from the coding process as the standalone is outside the IDE. Custom visualizations such as dashboards and statistics provide an overview over the current state of the code base, and illustrate progress. However, in industry tools, warning messages tend to be generic. The user interface also allows more interaction between user and the tool, but user interaction is generally limited to disabling warnings.

IDE tools: improve the workflow integration, since the analysis becomes part of the developer's coding environment. FindBugs in particular introduces a confidence factor. In conjunction with the IDE integration, this mechanism introduces a direct interaction with the developer, who can directly see that their feedback is taken into account. Providing more specialized views directly in the coding environment upgrades the workflow integration.

Tools with multiple interfaces: combine the mentioned forms (standalone, IDE, CLI). This group of tools is mainly comprised of commercial tools, which often provide advanced visualizations, statistics, navigation options, and dashboards. In particular, CodeSonar can provide non-generic warning messages, which efficiently supports the developer and helps them resolve warnings (e.g., "The issue occurs if the highlighted code executes. See related events 2 and 4."). Similarly to standalone tools, some of these tools limit user interaction to disabling warnings and customizing analysis rules, which can only be done by dedicated teams who have the required rule knowledge.

Overall, static-analysis tools used in industry show three main weak points:

1) Warning messages are generic and do not provide adequate fix support (in the case of complex warnings).
2) Mechanisms against false positives and user feedback are mainly limited to disabling warnings and customization of the analysis rules.
3) Many tools are not completely integrated into the IDE, which interrupts the developer's workflow.

IV. Explainability in Static Analysis Research

In our recent research, we have explored different ways of designing usable static analysis tools, to address the shortcomings of current industry tools. In particular, we explored two directions for explaining analysis warnings and security concepts to software developers. In one direction (**T9** Cheetah, **T11** Mudarri), we elected to expose the inner workings of the analysis to the developer, to enhance their understanding of how the analysis works, and why the analysis reports certain warnings [7], [20]. In the case of **T9**, we ensured that the warnings were updated immediately after a change in the code editor, adding to the responsiveness of the tool. In addition, the tools also allow the developer to edit the analysis rules, to a certain extent. In our user studies, we showed that this system was particularly efficient in helping developers understand and fix analysis warnings, and gave them a better experience of the tool.

In the opposite direction (**T8** CogniCrypt), we sought to hide the complexity of the warnings by simplifying the explanations and generating secure code for specific security

Table I: Static-analysis tools used in industry (**T1**–**T7**) and research (**T8**–**T14**), and how they address the six challenges.

	Tool	Type	C1	C2	C3	C4	C5	C6	Target user
T1	Codacy [11]	Standalone	Generic msg.	Generic msg.	User feedback	Disable warning/rule	CI integration	Custom interface	Developer
T2	cppcheck [12]	Standalone	Generic msg.	Generic msg.	Simple rules	—	IDE plugin	Custom interface	Developer
T3	RIPS [13]	Standalone	Generic msg.	Generic msg.	—	—	IDE plugin	Custom interface	Developer
T4	FindBugs [14]	IDE	Generic msg.	Generic msg.	User feedback	Disable warning, confidence factor	IDE integration, instant updates	Simple interface	Developer
T5	Fortify [15]	Multi-Interface	Generic msg.	Generic msg.	Customizable rules	—	IDE plugin	Custom interface	Developer
T6	Checkmarx [16]	Multi-Interface	Generic msg.	Generic msg.	User feedback, Customizable rules	Disable warning	IDE plugin	Custom interface	Developer
T7	CodeSonar [17]	Multi-Interface	Specific msg.	Specific msg.	User feedback, Customizable rules	Disable warning	IDE plugin	Custom interface	Developer
T8	CogniCrypt [18]	IDE	Specific msg.	Generic msg.	Performant solver	—	IDE integration, instant updates	Custom interface	Developer
T9	Cheetah [7]	IDE	Generic msg.	—	Customizable rules	—	IDE integration, instant updates	Simple interface	Developer
T10	VisuFlow [19]	IDE	—	—	—	Live debugging	IDE integration	Custom interface	Analysis dev.
T11	Mudarri [20]	IDE	Specific msg.	—	—	—	IDE integration	Simple interface	Configurer
T12	SWAN [21]	IDE	—	—	User feedback	Disable warning	IDE integration	Custom interface	Configurer
T13	Soot-based [22]	CLI	Generic msg.	—	—	—	—	CLI options	Developer
T14	FlowDroid [23]	CLI	Generic msg.	—	Performant solver	—	—	CLI options	Developer

tasks, such as to spare the developer the overhead of learning about the analysis tool and making common mistakes [24]. Our preliminary user study also showed a significant improvement in the coding experience. The first tool aims at giving the developer control over the analysis by *teaching* them about the analysis and its security rules, while the second one goes into the opposite direction of *assisting* the developer in their coding task, and hides the analysis' details.

We also extended our research to include different types of users. While a majority of static-analysis research focuses on software developers [7], [24], we also research how to help analysis developers [19], [25], or security teams that configure static analysis tools [20], [21]. The complexity of understanding why static analysis warnings are (or are not) reported, and whether or not they are relevant to the end-user are consistent problematics across all three user groups. Through our prototypes and user studies, we have observed the importance of the six explainability challenges detailed in Section II, in particular, providing understandable warning messages, integrating our tooling into the user's workflow, and designing dedicated functionalities (e.g., graph visualizations for analysis developers) [7], [19]–[21].

While those results are promising, they also open new areas for research, especially in the domain of helping the user understand the analysis, and helping the analysis understand how to properly assist the user, following the six explainability challenges from Section II.

V. Future Lines of Research

In Section II, we saw that the main challenges with the usability of static analysis tools revolve around an understandability gap between the analysis and the developer, where the developer struggles in understanding the warnings, and the analysis does not have all of the information the developer knows.

To enhance explainability in static analysis, we thus zoom out of the analysis itself, and envision an interactive system between the analysis tool and its user. Based on user input, the analysis tool could help the user perform their tasks, as an *assistant*, while at the same time building their knowledge of the tasks and of how the tool works, as a *teacher*.

For the *assistant*, we can build on top of the current research that hides the complexity of the analysis from the user, as is done in different domains, such as compilation errors, which often consist in short, actionable messages "Undeclared variable `x`". Spell checkers push the abstraction to the extreme by not explaining the warning at all, and directly proposing fixes. A potential research area would thus be to analyze and generate potential fixes from the analysis rules, and to report warnings based on such fixes rather than explanations of the warning itself. Taking this a step further, the analysis tool could also learn from the user's past fixes and reuse them. Another approach would be to prevent the warning from being generated in the first place, similarly to **T8** [24] where the tool generates secure code from the start.

When it comes to teaching complex concepts to the user, we can turn to human-centric applications such as StackOverflow or GitHub issues, where "warning" messages are answered by humans and yield adapted solutions. Many solutions are inaccurate but in the case of static analysis, the automated tool is less prone to errors than humans. The important factor for analysis tool design is the discussion between the different members of the community, who propose different solutions, ask for more details, and elect the correct answer. Learning how to address the user and engage in a human-like manner is an interesting research area for a *teacher* analysis tool, especially considering that in industry, developers rarely work alone, and the community aspect of teamwork can be leveraged to build a reliable knowledge base. In this system, the user would be able to learn from the tool in a more engaging way, and the tool would learn from the community

and could apply that knowledge when detecting vulnerabilities.

Another way of teaching complex concepts is shown in video games, which use different tricks to engage the player and involve them in quests through stories, difficulty calibration, positive feedback, and interactive graphical interfaces. Integrating elements of gamification into the analysis tool would allow for a more engaging experience.

The more the analysis knows about the code base, and the user's requirements and knowledge, the more precise and relevant its warnings. As a result, it is important to take into account the user's knowledge and assumptions of the code, and also what has already been taught to them, in order to provide warning messages that are relevant to particular users, in contrast to current tools that display the same message for everyone. In past research, we have started to investigate how to use machine learning to adapt static analysis to particular code bases (**T12**) [21], and propose to extend this work to particular users and tasks. We advocate for collaborations between the domains of artificial intelligence and static analysis, to study how to provide static-analysis users with adapted messages to both teach and assist them.

An important aspect of such a tool is knowing the user and their needs. Depending on the type of user (e.g., software developer, analysis developer, manager, etc.), their goals might fundamentally differ. As a result, there is a need to first understand the user requirements and how they influence their interaction with the tool, to determine the aspects that the analysis tool should take into account. Furthermore, it is important for the user's experience to know when to provide what kind of message, to avoid, for example, entering a long teaching-heavy session if the user has no time on their hands. Along with building a usable, engaging interface that allows them to use the tool to its full potential, this falls under the scope of human-computer-interaction, a domain with which we also encourage collaborations.

VI. Conclusion

Research on the usability static-analysis tools has identified a large body of usability issues, which we have grouped in six main challenges that center around the notion of explainability. While recent analysis tools address some of those challenges, the usability problems persist. We argue that their root cause lies in the currently unclear two-way communication between the analysis and the user. Therefore, we suggest the concepts of teaching and assisting systems. The assistant undertakes specific tasks and hides the need for explanation as it satisfies the user's requests. In contrast, the teacher provides tailored explanations required for the user's understanding of a specific task or warning. To this end, we encourage collaborations with experts from other domains such as human-computer interaction, gamification, and artificial intelligence, which share similar challenges or provide transferable approaches.

Acknowledgment

This research has been partially funded by the Heinz Nixdorf Foundation, the DFG projects RUNSECURE and CRC 1119 CROSSING, the BMBF within the Software Campus initiative, and the NRW Research Training Group on Human Centered Systems Security (nerd.nrw).

References

[1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681.

[2] E. Bodden and L. Nguyen Quang Do, "Explainable static analysis," in *Software Engineering und Software Management 2018*, M. Tichy, E. Bodden, M. Kuhrmann, S. Wagner, and J.-P. Steghöfer, Eds. Bonn: Gesellschaft für Informatik, 2018, pp. 205–208.

[3] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," ser. ICSE '13, 2013.

[4] L. N. Q. Do and E. Bodden, "Gamifying static analysis," ser. FSE '18, 2018. [Online]. Available: https://doi.org/10.1145/3236024.3264830

[5] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," ser. ASE'16, 2016. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970347

[6] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill, "From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration," ser. ICSME'16. IEEE, 2016, pp. 211–221.

[7] L. Nguyen Quang Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill, "Just-in-time static analysis," ser. ISSTA'17, 2017. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092705

[8] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton, "Aside: Ide support for web application security," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011.

[9] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 462–473.

[10] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," ser. DEFECTS '08. New York, NY, USA: ACM, 2008, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1390817.1390819

[11] Codacy, https://www.codacy.com/, 2019.

[12] Cppcheck, http://cppcheck.sourceforge.net, 2019.

[13] RIPS technologies, http://rips-scanner.sourceforge.net, 2019.

[14] FindBugs, http://findbugs.sourceforge.net, 2019.

[15] Fortify Software, https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview, 2019.

[16] Chechmarx, https://www.checkmarx.com/, 2019.

[17] Grammatech, https://www.grammatech.com/products/codesonar, 2019.

[18] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, "Cognicrypt: Supporting developers in using cryptography," ser. ASE '17. IEEE Press, 2017.

[19] L. Nguyen Quang Do, S. Krüger, P. Hill, K. Ali, and E. Bodden, "Debugging static analysis," *IEEE Transactions on Software Engineering*, 2018.

[20] L. Nguyen Quang Do, "Mudarri," https://github.com/secure-software-engineering/mudarri, 2019.

[21] G. Piskachev, L. Nguyen, and E. Bodden, "Codebase-adaptive detection of security-relevant methods," ser. ISSTA'19. ACM, 2019.

[22] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oct. 2011.

[23] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594299

[24] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," ser. ECOOP '18, 2018, pp. 10:1–10:27. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2018.10

[25] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for c/c++," ser. TACAS'19, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-17465-1_22