M. Tichy, E. Bodden, M. Kuhrmann, S. Wagner, J.-P. Steghöfer (Hrsg.): SE 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 205

Explainable Static Analysis

Eric Bodden[1] Lisa Nguyen Quang Do[2]

Abstract: Static code analysis is an important tool that aids in the early detection of programming errors, e.g. functional flaws, performance bottlenecks or security vulnerabilities. Past research in static analysis has mainly focused on the precise and efficient detection of programming mistakes, allowing new analyses to return more accurate results in a shorter time. However, end-user experience or static analysis tools in industry shows high abandonment rates. Previous work has discovered that current analysis tools are ill-adapted to meet the needs of their users, taking a long time to yield results and causing warnings to be frequently misinterpreted. This can quickly make the overall benefit of static analyses deteriorate.

In this work, we argue for the need of developing a line of research on aiding users of static analysis tools, e.g., code developers, to better understand the findings reported by those tools. We outline how we plan to address this problem space by a novel line of research that ultimately seeks to change static analysis tools from being tools for static analysis experts to tools that can be mastered by general code developers. To achieve this goal, we plan to develop novel techniques for formulating, inspecting and debugging static analyses and the rule sets they validate programs against.

Keywords: static analysis; debugging; visualization; program understanding

1 State of the Art

Historically, static code analysis tools have always faced the problem of unsatisfied users [Jo13, XWM14, CB16, Wi15]. In order to provide complete results, static analysis algorithms are known to over-approximate, returning all potential errors to the end-user, and sometimes overwhelming them with a "wall of bugs" [Jo13]. To lower the number of false positives, analyses are made more complex, but they take longer to run: minutes to hours to days. This forces developers to wait for a long time before they receive feedback on their code. Additionally, analysis tools often present warnings without explaining why they are returned, making it difficult for the code developer to distinguish false positives from genuine warnings. As researchers from the static-analysis tool vendor Coverity [Sy] (now owned by Synopsis) pointed out in 2010 [Be10], the lack of easy-to-understand warning messages is a primary cause of user dissatisfaction. Another one is the insufficient integration of the analysis tool with the developer's workflow [CB16].

As static analysis is generally an undecidable problem, all static-analysis tools must approximate their computations, thus unavoidably reporting false positives. In practice,

[1] Heinz Nixdorf Institute, Paderborn University & Fraunhofer IEM eric.bodden@upb.de
[2] Fraunhofer IEM lisa.nguyen@iem.fraunhofer.de

no tool can be precise enough to be false-positive free, which means that ultimately, the developer is the final judge in deciding whether a warning is true or false. Situations have been observed where complex warnings explained poorly by the tool misled the developer into thinking they were false positives [Be10]. Moreover, the notion of a false positive can by highly subjective, as even a true warning might not be relevant to certain developers. Therefore, there is a dire need for tools that efficiently help developers in determining which warnings are relevant to them.

2 Challenges

When trying to address the problems presented above, one faces the following challenges:

- To allow developers to better understand warnings, an analysis tool must provide richer information about *why* the analysis thinks that a certain program part is erroneous. Such information is not readily available. In fact, the static analysis tool must compute it while it conducts the analysis or as part of a post-processing module that runs after the analysis finishes. However, if no care is taken, such additional computations can significantly increase the time and memory consumption of the static analysis.
- Warnings must be presented to the developer in a way that is easy to understand. Little research has been performed on assessing which representations aid and don't aid code developers, and what elements should be part of an ideal representation [SBMH17].
- A method known to help developers assess a warning is the presentation of a witness, i.e., a real execution trace triggering the programming error [Le14]. But how can one generate witnesses also for incomplete code, and what are the program interfaces in such a case?

3 Required Research

In past work, we have investigated different means of adapting static analysis to the developers' needs [Ng17], and different ways of presenting the analysis results in an intuitive, user-friendly manner [Ng]. Through user studies, we have demonstrated that integrating developer information into the analysis and presenting results in a more transparent way helps developers fix errors significantly faster and provide them with a better overall experience of static analysis. While those first results are promising, there is still a lot to research to be conducted, especially in the areas of visualizations and responsiveness.

The envisioned line of research should aim at producing novel and improved means to present static analysis warnings to developers, ideally allowing them to assess a problematic situation quickly and correctly in all cases. To address the above problems, one must bring together expertise from the areas of static and dynamic program analysis and human-computer interaction. User studies would need to be an essential part of a work plan to address the problems mentioned above.

4 Possible solution outline

Concretely we propose to address the challenges outlined above by moving from a bare code-analysis technology to a developer-assistant system that will collaborate interactively with the developer to reach a joint understanding of the vulnerability situation at hand. As one of its design principles, such an assistant could include elements of gamification. For instance, the assistant could present to the developer potential vulnerabilities in a prioritized way, showing first those that are directly relevant to the developer, e.g. because they are close to the code they are currently editing, and who are also known to be easy to fix, taking into account the bug-fixing knowledge the particular developer is known to have acquired over the past. Then, when known to master well vulnerabilities at one level, the assistant could move the developer to the next level, displaying vulnerabilities e.g. of a different kind, or which are deemed slightly harder to judge. With knowledge of the developer's workflow and team setup, the assistant could even make use of the knowledge of the developer's team colleagues, e.g. by automatically including them into the judgment of vulnerabilities that the developer at hand is known not yet to be able to handle. This would allow the assistant to help one both derive the correct judgement (vulnerability or not) but would also aid the transfer of bug-fixing knowledge within the team.

As a second important element, however, we see dedicated new views and corresponding interactions that such an assistant should allow developers to make use of. In the past, we have seen a positive analysis developer response to a prototype that we built and which displays side by side the analyzed code with statements highlighted that contribute to a vulnerability, the implementation of the analysis but also a graph representation showing how the analysis executes over the given program part [Ng]. The user interface also allows novel interactions, e.g. setting special breakpoints that halt not the analyzed program but the static analysis when it processes the statement of the analyzed program that carries the breakpoint. In user experiments, we were able to show that such an approach eases the developers of static analyses in debugging the analyses that they write. The same views will likely be too complex to understand for developers who have no experience in static analysis. Yet, at the same time, we do believe that novel views are needed to explain developers *why* a static analysis arrives at a particular judgement, i.e., why it thinks that a certain vulnerability actually exists. Such novel views could also, for instance, display information about code locations at which the program analysis is uncertain, or might even ask dedicated, simple questions to the developer to help judge a vulnerability situation [DDA12], e.g.: "Is this untrusted input known to be sanitized elsewhere?" Those views, would then enable the developer to judge the vulnerability situation with much greater confidence, and in turn boost their ability to correctly judge future situations.

References

[Be10] Bessey, Al; Block, Ken; Chelf, Ben; Chou, Andy; Fulton, Bryan; Hallem, Seth; Henri-Gros, Charles; Kamsky, Asya; McPeak, Scott; Engler, Dawson: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. Commun. ACM, 53(2):66–75, February 2010.

[CB16] Christakis, Maria; Bird, Christian: What Developers Want and Need from Program Analysis: An Empirical Study. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE 2016, ACM, New York, NY, USA, pp. 332–343, 2016.

[DDA12] Dillig, Isil; Dillig, Thomas; Aiken, Alex: Automated Error Diagnosis Using Abductive Inference. SIGPLAN Not., 47(6):181–192, June 2012.

[Jo13] Johnson, Brittany; Song, Yoonki; Murphy-Hill, Emerson R.; Bowdidge, Robert W.: Why don't software developers use static analysis tools to find bugs? In: International Conference on Software Engineering (ICSE). pp. 672–681, 2013.

[Le14] Lerch, Johannes; Hermann, Ben; Bodden, Eric; Mezini, Mira: FlowTwist: Efficient Context-sensitive Inside-out Taint Analysis for Large Codebases. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014, ACM, New York, NY, USA, pp. 98–108, 2014.

[Ng] Nguyen Quang Do, Lisa; Krüger, Stefan; Hill, Patrick; Ali, Karim; Bodden, Eric: VisuFlow. https://blogs.uni-paderborn.de/sse/tools/visuflow-debugging-static-analysis/.

[Ng17] Nguyen Quang Do, Lisa; Ali, Karim; Livshits, Benjamin; Bodden, Eric; Smith, Justin; Murphy-Hill, Emerson: Just-in-time Static Analysis. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2017, ACM, New York, NY, USA, pp. 307–317, 2017.

[SBMH17] Smith, J.; Brown, C.; Murphy-Hill, E.: Flower: Navigating Program Flow in the IDE. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'17). October 2017.

[Sy] Synopsys: Coverity. http://www.coverity.com/.

[Wi15] Witschey, Jim; Zielinska, Olga; Welk, Allaire; Murphy-Hill, Emerson; Mayhorn, Chris; Zimmermann, Thomas: Quantifying Developers' Adoption of Security Tools. In: Foundations of Software Engineering (FSE). pp. 260–271, 2015.

[XWM14] Xiao, Shundan; Witschey, Jim; Murphy-Hill, Emerson R.: Social influences on secure development tool adoption: why security tools spread. In: Computer Supported Cooperative Work & Social Computing (CSCW). pp. 1095–1106, 2014.