Why do or why don't software developers use static analysis tools? A literature review

Kristóf Umann[1], Zoltán Porkoláb[1]

[1]ELTE, Eötvös Lóránd University, Budapest, Hungary. Faculty of Informatics, Department of Programming Languages and Compilers

Abstract

In the context of software technologies and literature, the field of static analysis is ancient. If we consider that compilation errors and warnings themselves are a form of static analysis, one could claim with ease that this field is about as old computer theory itself. Even involved algorithms, like symbolic execution, specifically tailored to find programming errors were around since the 1970s, and have enjoyed widespread use in the industry with the increase of computing power since the early 2000s.

While academic papers on new exciting techniques in the field of static analysis praise the technology, reading actual user experience reports paint a different picture. Static analyzer tools are frequently described as emitting numerous false positive or irrelevant reports and having poor quality warning messages, among other things. Since the early 2010s, renewed interest have sparked the publication of several surveys with human participants.

This paper serves as a literature review on the user experience with static analysis. We discuss six papers conducting surveys on tens, hundreds, or even thousands of participants all to learn how static analyzers are used, what are its pain points, how are they configured, among many other aspects.

Keywords

static analysis, survey, false positives, bug report quality, usability

1. Introduction

Our society is more dependent on software systems than ever before. Software appears in all aspects of life: from convenience functions to critical support systems. Vulnerabilities, especially in the latter systems, can cause serious material damage, and even can threaten human life. To avoid such situations software systems are intensively checked in various ways from different levels of testing to human code review. Among these methods, static analysis is one of the most promising, rapidly developing area with a number of open source and proprietary tools [1, 2, 3, 4].

Contrary to testing and dynamic analysis, static analysis works on the source code to detect various properties of the system, including possible errors, without running the program. Therefore, static analysis does not require the complete running environment (network, databases, etc.), and does not depend on carefully selected data to cover the most interesting parts of the software. Usually static analysis is applicable much earlier than the other software checking methods, many cases available even during the early stages of the development process. As the earlier the bug is detected is the cheaper to fix it [5], static analysis is a perfect fit method for the development-bug detection-bug fixing cycle.

On the other hand static analysis works using *heuristics*. Therefore it may *underestimate* or *overestimate* the program behavior [6]. In practice this means that the tools sometimes do not detect existing errors in the code which situation is called as *false negative*, and sometimes they report errors on correct code which is called *false positive*. While developers of static analysis tools are working hard to minimize these situations, the complete elimination of false negatives and false positives are theoretically impossible [7].

SQAMIA 2025: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 10–12, 2025, Maribor, Slovenia

szelethus@inf.elte.hu (K. Umann); gsd@inf.elte.hu (Z. Porkoláb)

gsd.web.elte.hu (Z. Porkoláb)

0000-0002-6679-5614 (K. Umann); 0000-0001-6819-0224 (Z. Porkoláb)

As such, static analyzers have not always enjoyed unanimous praise [8]. Scholarly work in the field often leads to vastly different opinions: papers on the new techniques and approaches on static analysis are usually positive, whereas surveys on the applicability of these tools are often negative or pessimistic.

To demonstrate, the authors of [9] describe the following:*„Symbolic execution (SE) is a highly successful method to automatically find bugs in programs"* and *„This basic bug-finding strategy has proved to be a powerful technique, supporting some of the most effective tools for analyzing large codebases"*. On the other hand, the authors of [10] discuss the actual application of a static analyzer, and write *„Can we just take KLEE and use it in the company as is? The answer is no. The designer of KLEE could not foresee the obstacles inevitable when working with real-life code."* While neither of these excerpts serve as a summary of the cited papers, they highlight the academic-industrial gap.

Fortunately, since the early 2010s, there is growing interest in user-focused surveys to better understand this gap [11, 12, 13, 14, 15, 16]. In particular, surveys where human software developers are involved are especially valuable, as they receive this data straight from the source [8, 17, 18, 19, 20, 21].

This paper serves as a comprehensive literature review for surveys on the usability of static analyzer tools involving human participants. There are also numerous papers on the usability, explainability, and applicability of static analyzers *not* involving human participants. While the human aspects of this research is important, we discuss three such papers we found were particularly relevant to this survey. The last comprehensive literature review on a similar topic was [22] in 2011.

Our paper is structured as follows. In Section 2., we discuss six articles involving software developer participants. When novel techniques were used to develop the questionaire or interview, we also show that alongside the results. In Section 3., we take a look at three articles from a tool designer perspective. We conclude our paper in Section 4.

2. Surveys with software developer participants

In 2008, the authors of paper [23] reported that while the static analysis methods are frequent research areas in the academy, there are no many usage examples in the industry. Current trends show a growing industrial interest of the static analysis tools [24, 25], but still, most tools are hindered by a high number of false positives and poor quality warning messages. Since the early 2010, several excellent surveys were conducted with software developers on why they do, or do not use static analyzers. In this section, we overview these articles.

We selected papers based on the following criteria:

- The article must be relatively recent, from 2010 or later. This lead to the omission of [26] from 2008.
- Have at least 20 participants. This lead to the omission of [27] from 2018, as they only had 8 participants, the 2018 article [28], as they only had 10, and the 2024 article [**?**], as they only had 11.
- Be focused on static analyzers beyond compiler errors and warnings. This lead to omission of the 2018 article [29] and the 2024 article [30].

It should be emphasized that papers we cite as an example here are also valuable additions to the academic community and worth reading.

2.1. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?

In 2013, authors in [8] investigate why the use of static analysis tools is not as widespread as it could be possible. They focused on the developer's perception on using the tools, including the interaction with the user interface. The research was conducted via 40-60 minutes long semi-structured interviews with 20 developers with experience ranging from 3 to 25 years. Among these 20 developers, 14 people expressed negative impact on the way in which the warnings are presented. Apart from the possibility of overwhelming false positive warnings they mentioned that the reports are non-intuitive.

As the result of their research, they conclude that developers are not able to understand what the tool is telling them, which is a definite barrier to use static analysis tools. Nineteen of the 20 participants, felt that many static analysis tools do not present their results in a way that gives enough information for them to assess what the problem is, why it is a problem and what they should be doing differently.

2.2. What developers want and need from program analysis: An empirical study

In 2016, the authors in [17] survey what authors want and need from static analyzers. The article focuses on industrial applicability: *So far, much research and many studies on program analysis tools have focused on the completeness of these tools [...], their soundness [...]. However, as companies integrate program analyzers as part of their development process, more investigation is needed into how these tools are used in practice and if practitioners' needs are being met.* The article frequently compares the similarities and differences in results against [8] (discussed in Section 2.1).

The authors used Kitchenham and Pfleeger's guidelines for personal opinion surveys in software engineering research [31] when designing and deploying their survey. First, they invited five developers for interviews to gauge the clarity of the survey questions. Based on their feedback, an enhanced version was sent out to 100 participants to further improve the survey format and wording, and the collected responses were not added to the final data set. Finally, they sent out the survey to 2000 randomly selected developers, and recieved 375 reponses. All participants in all three rounds were from Microsoft.

On their question *What Makes Program Analyzers Difficult to Use?*, developers ranked *Wrong checks are on by default* the highest, stating that several enabled checks *do not match what the developer wants*, such as specific convetion checks not used in their project (e.g. follow Hungarian Notation). *Too many false positives* was ranked only third behind *Bad warning messages*. Among the top 7 pain points also appeared the *lack of suggested fixes* and *bad visualization*. Interestingly, among 15 other choices, *Misses too many issues* was ranked second to last ahead of *Not cross platform*. Among other things, they conclude that *To build trust in their analyzers, tool designers should keep in mind that developers care much more about too many false positives than too many false negatives.*

The authors also note that *There are relatively few empirical studies [...] from the point of view of software engineers. So far, many studies have analyzed the functionality of program analyzers, mostly from the point of view of tool designers.* Fortunately, many excellent surveys have emerged since their publishing.

2.3. Context is king: The developer perspective on the usage of static analysis tools

In the 2018 research published in paper [18] the authors consider the development context in which analyzer tools are used. The authors sent out forms to 52 developers, and have gotten response from 42 users of static analyzers. 29 developers worked in the industry, while 13 were in open-source. Then, they interviewed 11 industrial experts to understand the possibilities of better prioritization of static analysis tool reports. Among other interesting results, they found that ... *warnings hard to integrate in case they do not have teammates having enough expertise for fixing them. However, those warnings can be easily understood if the tools provide exhaustive descriptions.*

2.4. How developers engage with static analysis tools in different contexts

The 2019 paper [19] surveys how the usage of automatic static analysis tools (*ASATs*, in their terminology) in the development workflow evolved over the years. They approach their survey from 3 angles: (1) A survey on the practical usage of ASATs with 56 participants, (2) A semi-structured interview with 11 participants to validate the findings from the previous point, (3) Manual inspection of ASAT configuration files of 176 open-source projects.

The authors put heavy emphasis on what analyzers developers use, the types of programming errors they want uncovered, and the development contexts in which the analysis is conducted. They conclude, among other findings, that *developers usually pay attention to different categories of defects while working locally, in code review or rather in CI. Specifically, they mainly look at Error Handling in CI, at Style*

Convention in Code Review, and at Code Structure locally. These warnings are not mutually exclusive though and some categories appear in different contexts with different weights. In terms of why ASAT findings are not being enforced (e.g. gating in pull requests) they find that the tools are *„buggy" and hard to configure.*

2.5. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations

As software systems are getting more and more complex and the same time vulnerable against various errors and attacks, a growing number of software companies use static analysis tools. In their 2022 paper [20] the authors investigate the main motivation of the developers to use these tools and their strategies to derive requirements and applicable guidelines for designing and using them. While about 30% of the developers used a static analysis tool because of company policies, more than 20% said that they help them to code faster. Meanwhile almost 80% admitted that the tools help to write better code. When they start to work on fixing the bug, they chose the most severe issue, which indicates that *bug severity* should be included in the bug report. The other mandatory information is the *bug type* (e.g., memory leak, uninitialized variable) so that the developer can start the report investigation based on that. The authors also measured that the developers investigate only the 65.1% of reports in detail. The authors state that *the ability of the analysis tool to explain the warning and showcase relevant data is key to supporting its users* [20].

2.6. Do Developers Use Static Application Security Testing (SAST) Tools Straight Out of the Box? A large-scale Empirical Study

Static Application Security Testing (SAST) is a security focused branch of Automated Static Analysis Tools (ASATs). The authors in the 2024 article [21] ask whether developers use SAST tools, and if so, how they configure them. They sent out screening test to 1263 developers, on which 260 (∼21%) failed by selecting fake SAST tools. Of the remaining 1003 participants, only 204 (∼20%) claimed to have used SAST tools, though the authors note that its possible that that 20% might be responsible for a greater portion of security checking in their codebases.

To the remaining 204 participants the authors sent out a survey on their usage of SAST tools, receiving 175 responses. The reason participants use SAST tools is that *they believe they are good at detecting vulnerabilities (78%) and convenient to use (57%). However, when asked how effective SAST tools are [...], most developers (53%) believed SAST tools to be only moderately effective [...], suggesting improvements can still be made to the detection rate of SAST tools to increase their effectiveness.*

SAST (and ASAT) tools can be configured to enable, disable, edit or create new bug detecting modules. The article later notes that the majority of developers *do not configure SAST tools (54%) or use more than a single tool (59%) and 40% do neither, potentially missing many vulnerabilities.* While the survey did not explicitly ask about easy configurability, the result suggests that *there is some difficulty when configuring the tool or, alternatively, integrating SAST tools into CI/CD pipelines may make configuring tools less accessible.* The fact that SonarQube is the most popular, yet least configured tool among the participants makes this an even more likely explanation.

However, among those that did configure their tool, they more often enabled, rather than disabled modules, suggesting that *developers prefer fewer false negatives over fewer false positives.* This is in sharp contrast with general ASATs, for which numerous studies [8, 17] found that a lower false positive ratio is preferred.

3. Tool designer perspective surveys

The literature in surveys on the usability of static analysis *not* involving human participants is vast and diverse. Because of this, we selected only a few we felt were particularly relevant to our paper.

3.1. Lessons from building static analysis tools at Google

Industry leader software companies show the most positive approach towards static analysis and its application in every day development. Google, Apple, Microsoft, Facebook, and others also participate in the creation of such tools. The paper [14] from 2018 reports about the lessons learned while developing static analysis tools at Google. The authors list the most frequent problems resulting in the developers not using static analysis tools or ignoring their warnings. These are the lack of tool integration into the developer's workflow; the fact that *many warnings are not actionable*; the high number of false positives; situations where the bug is theoretically possible but in practice it does not manifest; the possibly high cost of the fix; and that the users do not understand the warnings.

The authors emphasize the importance of *actionable* messages: the warnings should include a suggestion to the (possible) fix, which in the best case could be applied automatically. However, the authors state that many serious issues cannot be detected correctly or automatically fixed. The fix depends on the correct understanding of the report. They also claim that the developer's happiness is a crucial factor for the successful introduction of static analysis on an organizational level. Non-understandable reports cause frustration among engineers and work against trust in static analysis tools.

3.2. Explaining Static Analysis – A Perspective

In their 2019 short paper [15] the authors specifically target a known pain point in the usability of static analyzers: explainability. They claim that this is a major step to *overcome the gap between the academically perceived potential of static analysis and its use in practice.*

The article surveys recent (at the time of writing) research into the following challenges static analyzer tools face: (1) the impact of warning message quality, (2) fixin support, when the tool offers to help fix the finding, (3) dealing with false positives, (4) integrating user feedback into the analysis results, (5) workflow integration, (6) user interface developers need to interact with.

Then, they survey 14 state-of-the-art static analyzer tools based on these challenges. They conclude that *static-analysis tools used in industry show three main weak points: (1) Warning messages are generic and do not provide ade- quate 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.*

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

From the same authors as [15], the 2022 study [16] presents the *first systematic usability evaluation in a wide range of static analysis tools.* They collected 243 tools for popular programming languages, excluded those that emitted no warnings or hints, and those that were either inaccessible, proprietary (with one exception), or unmaintained for longer than 2 years. The remaining 46 Static Analysis Tools (*SAT*) were evaluated.

After a thorough literature review, the authors compiled 36 criteria for evaluation. Each criterion was grouped to one of the 6 challenges identified in the previous article. Each SAT either „clearly fulfilled", „somewhat fulfilled" or „virtually fulfilled" all each challenge based on the challenge's criteria. While the evaluation methodology is described in the paper, it is not clearly established that how many criteria need to be met in order for the corresponding challenge to be fulfilled, or if some criteria are weighed more against others. Still, the results confirm the conclusions of the earlier paper, but with concrete measurements on each challenge. A highlight, on the issue of bug report quality, they 30 of 46 tools have too poor warning messages, and *Explaining the code issue with details exceeding the most basic aspects also remains a common problem.*

Furthermore, the reference list of this article is impressive, providing an excellent source for further research.

4. Conclusion

Static analyzers are about as old as programming itself. Specifically for finding bugs and programming errors, lightweight analyses in the form of compiler warnings have been in use for a long time, but more thorough analyses like symbolic execution only enjoyed widespread use since the early 2000s with the increase of computing power. Static analyzers, even when they are computationally demanding, provide an early feedback on the quality of the software and offer wider code coverage than dynamic analyzers.

With that said, static analyzers are less precise than dynamic analyzers, they tend to emit false positives, and struggle to create easy to understand warning messages. As the userbase of these tools grew, so did the desire for academic work in how software developers use static analyzers, and what aspects hinder the experience.

Our paper surveyed six papers involving human participants on the usability, explainability and applicability of static analyzer tools, ranging from tens of participants to thousands, from as early as 2013 to as late as 2024. We learned that high number false positives and poor quality warning messages are leading causes for dissatisfaction. It is often stated that developers prefer a lower false positive rate even at the cost of losing real findings. The most recent paper, which surveyed the usage of Static Application Security Testing found that security focused users may think otherwise, and consider enabling more checks to find more bugs.

We also surveyed three papers from a tool designer perspective, but still focusing on usability. The papers also find that among other things, lowering the false positive ratio and improving the analyzer tool – human expert relationship is important.

Acknowledgment

Project no. C2314106 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the KDP-2023 funding scheme.

Declaration on Generative AI

We have not used AI in any part of the research or writing process.

References

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler, A few billion lines of code later: Using static analysis to find bugs in the real world, Commun. ACM 53 (2010) 66–75. URL: http://doi.acm.org/10.1145/1646353.1646374. doi:10.1145/1646353.1646374.

[2] Synopsys, Coverity, 2019. https://scan.coverity.com/ (last accessed: 24-04-2023).

[3] Roguewave, Klocwork, 2019. https://www.roguewave.com/products-services/klocwork (last accessed: 28-02-2019).

[4] CodeSecure, CodeSonar, 2025. https://codesecure.com/our-products/codesonar/ (last accessed: 29-07-2025).

[5] B. Boehm, V. R. Basili, Software defect reduction top 10 list, Computer 34 (2001) 135–137. URL: http://dx.doi.org/10.1109/2.962984. doi:10.1109/2.962984.

[6] M. Anders, I. S. Michael, Static program analysis., 2012. URL: https://users-cs.au.dk/amoeller/spa/spa.pdf.

[7] H. G. Rice, Classes of recursively enumerable sets and their decision problems, Trans. Amer. Math. Soc. 74 (1953) 358–366. URL: https://www.bibsonomy.org/bibtex/207eadfff0be3322a169ba4ac8dad06a4/idsia.

[8] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs? (2013) 672–681. URL: http://dl.acm.org/citation.cfm?id=2486788.2486877.

[9] Z. Susag, S. Lahiri, J. Hsu, S. Roy, Symbolic execution for randomized programs, Proc. ACM Program. Lang. 6 (2022). URL: https://doi.org/10.1145/3563344. doi:10.1145/3563344.

[10] A. Misonizhnik, S. Morozov, Y. Kostyukov, V. Kalugin, A. Babushkin, D. Mordvinov, D. Ivanov, KLEEF: Symbolic Execution Engine (Competition Contribution), Springer Nature Switzerland, 2024, p. 314–319. URL: http://dx.doi.org/10.1007/978-3-031-57259-3_18. doi:10.1007/978-3-031-57259-3_18.

[11] C. Wijayarathna, N. A. G. Arachchilage, Why johnny can't develop a secure application? a usability analysis of java secure socket extension api, Computers & Security 80 (2019) 54–73. URL: http://dx.doi.org/10.1016/j.cose.2018.09.007. doi:10.1016/j.cose.2018.09.007.

[12] B. Hartmann, D. MacDougall, J. Brandt, S. R. Klemmer, What would other programmers do: suggesting solutions to error messages, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 1019–1028. URL: https://doi.org/10.1145/1753326.1753478. doi:10.1145/1753326.1753478.

[13] N. Imtiaz, A. Rahman, E. Farhana, L. Williams, Challenges with responding to static analysis tool alerts, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 245–249. doi:10.1109/MSR.2019.00049.

[14] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, C. Jaspan, Lessons from building static analysis tools at google, Communications of the ACM 61 (2018) 58–66. URL: https://doi.org/10.1145%2F3188720. doi:10.1145/3188720.

[15] M. Nachtigall, L. Nguyen Quang Do, E. Bodden, Explaining static analysis - a perspective, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), 2019, pp. 29–32. doi:10.1109/ASEW.2019.00023.

[16] M. Nachtigall, M. Schlichtig, E. Bodden, A large-scale study of usability criteria addressed by static analysis tools, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, Association for Computing Machinery, New York, NY, USA, 2022, p. 532–543. URL: https://doi.org/10.1145/3533767.3534374. doi:10.1145/3533767.3534374.

[17] M. Christakis, C. Bird, What developers want and need from program analysis: An empirical study, in: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 332–343.

[18] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, H. C. Gall, 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), 2018, pp. 38–49. doi:10.1109/SANER.2018.8330195.

[19] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, A. Zaidman, How developers engage with static analysis tools in different contexts, Empirical Software Engineering 25 (2019) 1419–1457. URL: http://dx.doi.org/10.1007/s10664-019-09750-5. doi:10.1007/s10664-019-09750-5.

[20] L. N. Q. Do, J. R. Wright, K. Ali, Why do software developers use static analysis tools? a user-centered study of developer needs and motivations, IEEE Transactions on Software Engineering 48 (2022) 835–847. doi:10.1109/TSE.2020.3004525.

[21] G. Bennett, T. Hall, S. Counsell, E. Winter, T. Shippey, Do developers use static application security testing (sast) tools straight out of the box? a large-scale empirical study, in: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 454–460. URL: https://doi.org/10.1145/3674805.3690750. doi:10.1145/3674805.3690750.

[22] S. Heckman, L. Williams, A systematic literature review of actionable alert identification techniques for automated static code analysis, Information and Software Technology 53 (2011) 363–387. URL:

https://www.sciencedirect.com/science/article/pii/S0950584910002235. doi:https://doi.org/10.1016/j.infsof.2010.12.007, special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.

[23] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix, Using static analysis to find bugs, IEEE Software 25 (2008) 22–29. URL: https://doi.org/10.1109%2Fms.2008.130. doi:10.1109/ms.2008.130.

[24] M. Beller, R. Bholanath, S. McIntosh, A. Zaidman, Analyzing the state of static analysis: A large-scale evaluation in open source software, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2016. URL: https://doi.org/10.1109%2Fsaner.2016.105. doi:10.1109/saner.2016.105.

[25] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, M. D. Penta, How open source projects use static code analysis tools in continuous integration pipelines, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017. URL: https://doi.org/10.1109%2Fmsr.2017.2. doi:10.1109/msr.2017.2.

[26] N. Ayewah, W. Pugh, A report on a survey and study of static analysis users, in: Proceedings of the 2008 Workshop on Defects in Large Software Systems, DEFECTS '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 1–5. URL: https://doi.org/10.1145/1390817.1390819. doi:10.1145/1390817.1390819.

[27] T. D. Oyetoyan, B. Milosheska, M. Grini, D. Soares Cruzes, Myths and facts about static application security testing tools: An action research at telenor digital, in: J. Garbajosa, X. Wang, A. Aguiar (Eds.), Agile Processes in Software Engineering and Extreme Programming, Springer International Publishing, Cham, 2018, pp. 86–103.

[28] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, H. R. Lipford, How developers diagnose potential security vulnerabilities with a static analysis tool, IEEE Transactions on Software Engineering 45 (2019) 877–897. doi:10.1109/TSE.2018.2810116.

[29] T. Barik, D. Ford, E. Murphy-Hill, C. Parnin, 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, ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 633–643. URL: https://doi.org/10.1145/3236024.3236040. doi:10.1145/3236024.3236040.

[30] N. Vánder, G. Antal, P. Hegedüs, R. Ferenc, On the usefulness of python structural pattern matching: An empirical study, in: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024, pp. 501–511. doi:10.1109/SANER60148.2024.00058.

[31] B. A. Kitchenham, S. L. Pfleeger, Personal Opinion Surveys, Springer London, London, 2008, pp. 63–92. URL: https://doi.org/10.1007/978-1-84800-044-5_3. doi:10.1007/978-1-84800-044-5_3.