Evaluating vulnerability-reporting models in secure software development

Kyle Wang
University of Wisconsin-Madison
kyle.wang@wisc.edu

Harikrishnan Venugopalan
University of Wisconsin-Madison
hvenugopalan@wisc.edu

ABSTRACT

Numerous studies show that structuring error messages in a way that aligns closely with how developers explain anomalies to themselves improves the easiness with which these errors can be fixed. This paper aims to study how the features of a report generated by a static code analysis tool and the level of knowledge a person has about software security influences their efficiency in fixing the reported security vulnerabilities. To show that the presence of either or both of the Common Weakness Enumeration(CWE) information and tool rule description in such a report improves a participant's efficiency, they are given four reports each consisting of different combinations of these two kinds of information, and are asked to fix as many vulnerabilities as they can within a given amount of time. Inferential analysis of the observed data shows the contrary, but a comparison of the means of different groups shows that these features do help improve the efficiency of those who have no knowledge about software security. However, a post-experiment survey shows that all participants, regardless of their knowledge-levels find these features useful. Hence, a larger and more diverse sample is needed to further substantiate the results from this study.

Author Keywords

Secure Software Development; Survey; Theoretical models of argument; Human-Computer Interaction

CCS Concepts

•**Human-centered computing** → **Usability testing;** •**Security and privacy** → *Software security engineering;* •**Software and its engineering** → Software maintenance tools;

INTRODUCTION

Enterprises are leveraging a myriad of open-source products for a range of business use cases for its speed, flexibility, and cost benefits. But it also comes with unique security challenges. Owing to its short release cycles, open-source projects can be hard to keep up with. Research [4] has shown that the shortage of time is the single most important reason why secure code practices are not prioritized in such projects. Numerous studies have been conducted on how using theoretical models of argument and structuring error messages in a way that aligns closely with how developers explain anomalies to themselves, improve the easiness with which errors are fixed. However, these studies were restricted to generic syntax errors and not security vulnerabilities. Meanwhile, security experts have made progress in improving the usability of software security tools, such as trying multiple approaches to reduce the rate of false positives reported [9]. This paper aims to find out if providing feedback about security vulnerabilities using such theoretical models of argument in the early stages of software development would lead to a more secure code. To be more specific, this paper will address the following research questions: ***RQ1****: How developers with different knowledge levels about software security will respond to the presence or absence of different features in a report generated by a static code analysis tool?* ***RQ2****: How the information in a vulnerability report should be presented to increase a programmer's efficiency in fixing vulnerabilities?*

CHI'20, April 25–30, 2020, Honolulu, HI, USA

ISBN 978-1-4503-6708-0/20/04…$15.00

DOI: https://doi.org/10.1145/3313831.XXXXXXX

RELATED WORKS

Programming Experience

There has been quite a lot of research focused on the programming experience (PX), and the relationship between tools and software developers. In their review over literature that has been published over the past ten years (2009-2019), Morales et al. [13] found that most papers related to PX were also closely related to usability in User Experience (UX), and actually, more than 70% of papers implemented usability studies as evaluation methods. 69% of studies were related to the programming environment. To be more specific, people have been concerned with the informative, aesthetic, and trustworthiness of the coding environment. Through their interviews with fifteen professional software developers, Xie, Lipford, and Chu [17] figured out a strong need for a programming environment with assistance on software security. Participants tended to rely on other people and technologies when it came to issues related to software security.

Security Tools for Programmers

Kupsch and Miller [12] from the University of Wisconsin - Madison did a comprehensive comparison between manual assessments for software security and automated assessments conducted by software security tools. Even though the results showed that the two commonly used tools they tested failed

on detecting nine vulnerabilities, they anticipated this gap between tools and security experts. They concluded that even though tools are not perfect, they could still be useful to detect certain types of bugs if operators were skillful enough to justify results and to fix problems. Static analysis tools (SATs) have been used for software security in both studies and practices. Both Chess and McGraw's paper [10] and Nagappan and Ball's paper[15] looked into how the SAT can be used to improve software security.

Tjoa et al. [16] surveyed various underlying tools of the OSCAR framework using Webgoat, a web application developed by OWASP to teach security lessons(especially penetration testing). They observed that different tools reported the same security vulnerabilities like SQL injections at different lines of code. For example, Yasca detected SQL injections where the string for the database was built. By contrast, VisualCodeGrepper detected SQL injections where the query was executed and IDEA recognized that the SQL statement was called with a non-constant string.

Christopher [11] evaluated three properties that are important to static analysis frameworks and that makes them ideal for software developers: documentation, example analyses, and the developer community. Frameworks that had documentation and tutorials for how each component works and they work together, frameworks that come with the same analyses for developers to peruse and modify, and those frameworks that have a strong developer community were found to be ideal.

On the PX perspective, Baca et al. [5] conducted an experiment in the software development industry in which it was found that participants had difficulty explaining and classifying outputs from Coverity, one of the best SATs for software security at that time. Besides, participants also could not tell the correctness of those vulnerabilities that got reported. This paper mentioned that SATs should be carefully designed to lower the false positive rate so that the interaction between the tools and developers can go more smoothly. Researchers did focus on lowering the false alarms by SATs using all sorts of approaches as mentioned in Yoon et al.'s paper [18]. However, there is more to be done from other perspectives that can be used to help developers to use SATs for software security more efficiently. It should be noticed that both Baca et al.'s paper and Nagappan and Ball's paper indicated that SATs should be used as an early-stage fault detector, which means that more work needs to be done by programmers following the outputs from SAT. Given that current outputs from SAT are not informative enough, there have also been studies about how the outputs from tools for programmers can be improved.

Better models for explaining anomalies to developers

The outputs from SATs could be improved based on results from some studies with similar tools. Barik et al. [8] discussed how IDEs should provide error notifications for potential syntax errors that could be easily comprehended by developers. The results of their paper suggested the diagrammatic techniques could be a self-explanatory way to visually communicate to developers. Murphy-Hill and Black [14] suggested a graphical way called Refactoring Annotations to present refactoring errors, which is not a bug and that may not be related to software security as well. However, both papers mentioned above-conducted experiments and found graphical ways to be a solution to improve a tool's usability for programmers. Furthermore, Murphy-Hill and Black also listed several guidelines for presenting errors to programmers that can also be applied on SAT: expressiveness, locatability, estimability, relationality, completeness, perceptibility, and distinguishability.

In [8], Barik et al. explains the duplicate work that the developer has to do- starting with the error notification, identifying what they think the problem might be from the IDE's presentation, mentally collecting all of the program components related to this problem, and finally identifying the area or areas of source code necessary to correct the particular defect - when trying to understand the error reported by a compiler. They hence provide a set of visual annotations that expose the compiler's internal reasoning process and aid developers in reasoning about the causes of errors.

In [6], Barik proves that re-framing error messages in static analysis frameworks as error explanations that align closely with how developers explain anomalies to themselves, significantly improves the comprehensibility and utility of these messages. He also found that users preferred quick fixes when minimal or no design exploration was necessary to apply the fix.

In [7], Barik et al. present theories of explanation such as Toulmin's model of argument to improve the quality of error messages. He found that when developers are given a pair of error messages, they significantly prefer the one with a proper argument structure over one with a deficient argument structure when neither offers a resolution. However, they would accept a deficient argument structure if it provides a resolution to the problem.

METHOD

Hypotheses

We aim to test two main-effect and two interaction-effect hypotheses. In the main-effect hypotheses described below, CWE(Common Weakness Enumeration)[1] refers to a detailed explanation or a link to an external website that has vulnerabilities carefully documented, and the tool rule description refers to an explanation of the possible solutions to fix a vulnerability with examples.

H_{11} :The presence of tool rule description improves the developer's efficiency in addressing security vulnerabilities.

H_{12} :The presence of CWE information improves the developer's efficiency in addressing security vulnerabilities.

The interaction-effect hypotheses we would like to test are:

H_{13} : The presence of control flow and CWE information significantly improves the developer's efficiency in addressing security vulnerabilities.

H_{14} : People with different knowledge levels show varying efficiencies in fixing vulnerabilities on the same kind of report.

These hypotheses are based on the general guidelines given by Murphy-Hill and Black in [14]. As discussed in previous work, the false-positive rates in reports may vary from tool to tool, but the ways to present errors may also vary and will both have an impact on users' experience. Also, it is highly probable that the terms used in the reports generated do not make any sense to programmers with no knowledge about software security. Thus it might take them longer to fix those vulnerabilities than those who have more experience in software security. Certain features, like the CWE information and tool rule description, included in reports from SATs could help security novices to quickly learn and understand those vulnerabilities.

Participants

The ideal target for this experiment would have been software developers with different knowledge levels about software security and also those with varying levels of coding experience.

Due to the accessibility limits imposed by the spread of COVID-19, the participants for this study were restricted to six students currently majoring in Computer Sciences at the University of Wisconsin - Madison. Specifically, two male undergraduate students, a male graduate student, and three female graduate students took part in the experiment. Considering that there are two courses related to security (CS 639 and CS 642) and varying coding experiences (which mainly depends on the school year and internship experience) between the individuals, this convenience sample could simulate the real world composition of the population of software developers in general.

The knowledge levels of the participants - None, Basic, Intermediate, or above - was chosen to be the blocking factor in this experiment. The six participants chosen were such that there were two participants under each of the knowledge levels. Participants who have neither taken a security course nor have any industry experience were grouped under the 'None' category. Participants who had either taken a security course or have had industry experience were classified under the 'Basic' category. Participants who have both taken a security course and had industry experience were classified under the 'Intermediate or above' category.

Study Design

The experiment followed a 2×2 mixed-model design by combining within-participants design to test how the presence or absence of CWE information and tool rule description affects a programmer's efficiency, and between-participants design to test how a developer's knowledge level about software security affects their efficiency in fixing vulnerabilities.

The independent variables corresponded to the presence or absence of CWE information and the presence or absence of tool rule description. The dependent variable corresponded to the efficiency of participants to spot and solve those reported vulnerabilities. It should be noted that knowledge levels of software security was considered to be a blocking factor to divide the participants into different groups.

Study Task

All of the participants were asked to read four reports each in sessions spanning ten minutes each. The four reports were crafted from a report generated by SpotBugs[3] in the Software Assurance Marketplace (SWAMP) [2] such that the first one had neither the CWE description nor the tool rule description, the second had just the tool rule description, the third only had the CWE description, and the fourth report had both kinds of information. To prevent any learning-induced bias between each session, the four reports generated were for four different programs.

SpotBugs is a static analysis tool that looks for bugs in Java code. The code can be analyzed using the SpotBugs standalone application or can be analyzed within Eclipse using a plug-in. There are options to choose minimum warning priority and enabled bug categories. SpotBugs reports the following vulnerabilities: cross-site scripting, HTTP response splitting, absolute path traversal, relative path traversal, hard-coded or empty database passwords,OS command injection, XML injection, and SQL injection[3]

At the end of each ten-minute session, the participants were asked to write down their understanding of the issue, its location in the code, and a possible solution. Because of the potential differences in coding experiences among participants, they were asked to write down the solution they came up with or that they found online in plain English.

Study Procedure

The participants were recruited through social media and by circulating emails.

The participants were then asked to complete a short survey to evaluate their knowledge levels of software security. Since there are no standard tests in the industry to gauge expertise in software security for programmers, a customized survey to fit with the convenience samples was used in this study. The survey consisted of five multiple-choice questions. The first one asked if participants had taken any courses related to security. The second one asked if participants had worked in the industry before and had experience with deploying secure code. If both of the questions' answers were negative, participants' knowledge levels was considered to be "None". For the third to fifth question, participants were asked to check if a given piece of code was immune to a particular kind of attack or tested on explanations of specific terms in the field of security. If participants selected "Yes" for one or both of the first two questions and answered all of the remaining questions correctly, their knowledge level was considered as "Intermediate or advanced", and "Basic" otherwise. Only two students under each knowledge level category were recruited.

The participants were grouped into three different blocks based on the knowledge levels determined by the survey above.

The participants were then asked to leave an hour of free time to complete the study. To prevent the spread of COVID-19, an online experiment was set up. Participants were invited to a one-to-one remote meeting on Zoom or Microsoft Teams and an email containing a program written in Java and a report about the vulnerabilities of the program were sent to them once

they were online. The report neither had the CWE information nor the tool rule description. To prevent any bias, it was made sure that each block started with a different Java program. The vulnerabilities in the four programs matched in severity. The participants were then given ten minutes to fix the issues in the program by going through the reports and searching online. The participants were free to fix the issues in any order they wished, and were allowed to use external resources to come up with the solutions. At the end of the session, the participants were asked to write down their understanding of the issues reported, their location in the code, and how to fix them.

The above step was repeated for each participant in a block for another three sessions to ensure that they worked on all four types of reports.

The participants were then asked to fill a post-experiment survey. Each of the questions in the survey was related to a possible factor around the participants' experience of fixing vulnerabilities with the help from different reports. The factors included understanding codes, familiarity with Java, friendly interfaces, categorizations of vulnerabilities, ordering of vulnerabilities, explanation about the vulnerabilities, examples related to the vulnerabilities, marks for vulnerabilities among lines of codes, and quality of external resources. Participants were asked to rate each statement from 'Strongly disagree (1)' to 'Strongly Agree (5)'.

Finally, they were asked several questions based on their survey responses to get qualitative responses about factors listed in the survey, For example, "which feature from the report helped you most when you were trying to understand and fix vulnerabilities?".

Measures

The efficiency with which programmers identified and solved vulnerabilities using a report was calculated based on the number of vulnerabilities they fixed within 10 minutes. A successful fix is considered as correctly stating the exact location to fix the vulnerability, the potential solution, and why the solution could work. A vulnerability was considered to be fixed by the participant only if its exact location in the code and a possible solution was identified. Therefore, the efficiency was calculated as

$$\frac{\text{Number of issues fixed}}{10}$$

Analysis

Since there were only one objective measure and two independent variables with one blocking factor, multi-way ANOVA was chosen as the model to analyze the measurement for this study. The four reports that each participant read correspond to one of the combinations of the independent variables in the 2×2 study. Since the participants' knowledge level of computer security has been used as a blocking factor, its error term is not included in the model.

RESULTS

Participants' knowledge level and efficiency

The inferential test done on the experimental data using multi-way ANOVA showed that the level of knowledge a person had about software security determined the number of vulnerabilities that they were able to fix (F(1,4) = 3.88, p = 0.12).

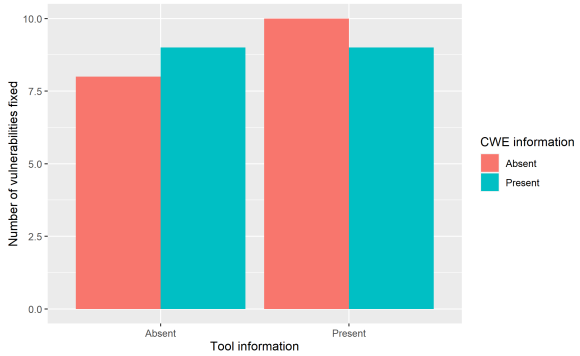

Figure 1. The mean of the number of vulnerabilities fixed by participants depending on the kind of information provided in a report

On average, participants with an intermediate level of knowledge about security were able to fix 7.62 issues(SD = 1.06). Participants with basic knowledge about security were able to fix 4.25 issues(SD = 2.82). Meanwhile, participants with no knowledge about software security were able to solve 3.38 issues(SD = 3.11).

Efficiency using reports with CWE information

It was found that the presence of CWE information additional to the finding details in the report did not improve a participants' efficiency in fixing the vulnerabilities described (F(1, 4) = 0.93, p = 0.39). Also, the level of knowledge a person had did not have any interaction effect with their efficiency in fixing issues using a report with CWE information. (F(1,4) = 0.350, p = 0.59).

On average, participants across various knowledge levels were able to solve only 4.58 issues(SD = 2.91) in ten minutes using a report with CWE information compared to 5.58 issues(SD = 3.2) using a report with just the finding details. However, participants who had no knowledge about software security were able to solve 3.75 issues on average (SD = 3.77) using a report with CWE information, whereas such participants could only solve 3 issues on average using a report with just the finding details.

Efficiency using reports with tool rule description

It was found that the presence of tool rule description additional to the finding details in the report did not improve a participants' efficiency in fixing the vulnerabilities described (F(1,4) = 3.2, p = 0.15). Also, the level of knowledge a person had did not have any interaction effect with their efficiency in fixing issues using a report with tool rule description. (F(1,4) = 1.2, p = 0.34).

On average, participants across various knowledge levels were able to solve 5.42 issues(SD = 3.15) in ten minutes using a report with tool rule description compared to 4.75 issues(SD = 3.02) using a report with just the finding details. Participants with an intermediate level of knowledge about security were able to fix 8 issues on average (SD = 1.15) using a report with

		Statistics	
Knowledge level	*CWE information*	*Mean*	*SD*
Intermediate	Present	8	1.15
	Absent	7.25	0.96
Basic	Present	5	3.37
	Absent	3.5	2.38
None	Present	3.25	2.87
	Absent	3.5	3.79

Table 1. Number of issues fixed using a report with, and without tool rule description.

		Statistics	
Knowledge level	*Tool rule description*	*Mean*	*SD*
Intermediate	Present	7.25	1.26
	Absent	8	0.82
Basic	Present	2.75	0.5
	Absent	5.75	3.5
None	Present	3.75	3.77
	Absent	3	2.83

Table 2. Number of issues fixed using a report with, and without tool rule description.

tool rule description, while they could fix only 7.25 issues(SD = 0.96) without it. Participants with a basic level of knowledge about software security were able to fix 5 issues (SD = 3.37) on average using a report with tool rule description, while they could solve only 3.5 issues(SD = 2.38) using a report without it. However, participants with no knowledge about software security were able to fix only 3.25 issues(SD = 2.87) using a report with tool rule description, while they could solve 3.5 issues (SD = 3.79) using a report without it.

Efficiency using reports with both tool rule description and CWE information

It was found that the presence of both tool rule description and CWE information additional to the finding details did not improve a participant's efficiency in fixing the vulnerabilities described ($F(1, 4) = 1.538$, $p = 0.283$). Also, the level of knowledge a person had did not have any interaction effect with their efficiency in fixing issues using a report with or without CWE information and tool rule description. ($F(1,4) = 3.32$, $p = 0.14$).

The descriptive analysis showed that participants were able to fix 4.5 issues(SD = 2.95) using a report with both CWE information and tool rule description in addition to the finding details, compared to 4.83 issues(SD = 3.19) when the report just contained the finding details. A more detailed analysis is given in Table 3.

DISCUSSION

The analysis shows that the presence of tool rule description in a vulnerability report increases the efficiency of those participants who have no knowledge about software security, but decreases the efficiency of those with a basic to intermediate level of knowledge about security. This aligned with the results from the post-experiment survey for those who did not

			Statistics	
Knowledge level	*CWE*	*Tool rule desc.*	*Mean*	*SD*
Intermediate	Absent	Absent	8	0
	Present	Absent	6.5	0.71
	Absent	Present	8	1.41
	Present	Present	8	1.41
Basic	Absent	Absent	4.5	3.54
	Present	Absent	2.5	0.71
	Absent	Present	7	4.24
	Present	Present	3	0
None	Absent	Absent	2	1.41
	Present	Absent	5	5.66
	Absent	Present	4	4.24
	Present	Present	2.5	2.12

Table 3. Number of issues fixed using a report with, and without CWE information and tool rule description.

have any knowledge about software security. However, it is interesting that those who had a basic to intermediate level of knowledge about software security thought that the tool rule description was helpful when it actually brought down their efficiency. The inferential analysis also shows that the presence of CWE information increases the efficiency of participants who have no knowledge about software security but doesn't help those with a basic to intermediate knowledge level. In fact, the post-experiment survey showed that participants with basic to intermediate knowledge levels felt strongly that the CWE information was not useful and disturbed the conciseness of the report.

Thus, it can be inferred that the presence of CWE information and tool rule description increases the efficiency of those who have no knowledge about software security. But to those with a basic to intermediate knowledge-level, the CWE information could be a nuisance that disturbs the conciseness of the report and the tool rule description may need to be more detailed or refined to significantly improve their efficiency.

Limitations

It should be noticed that there are quite a lot of limitations to this study. Due to the circumstances surrounding COVID-19, the study was pushed online, causing challenges to the experimental setup and restrictions on the way the report was presented to the subjects. Furthermore, the situation limited the number and diversity of participants whom we could recruit. The number of participants only fulfills the lower bound of the number of participants required by the design for this study. With more participants, the results could be more reliable from a statistical perspective, and could also alleviate the effects of personal differences in coding experience on the result. One of the possible causes of the analysis showing an insignificant difference between different formats of the reports could be due to the part that familiarity with Java and coding plays in understanding the reports. Also, since all the participants were undergraduate and graduate students, the result cannot be generalized to the entire developer community. Thus, scaling this experiment to a wider population would be an interesting next step. Also, given that this was

a class project spanning a single semester, the strict timeline limited the study from recruiting more participants as well as enhancing the quality of the design of the experiment. For instance, we should include more vulnerabilities in each report since some of the participants were able to finish all of the ten vulnerabilities within the time limit. The types of vulnerabilities included should also be more diverse and exclusive from one report to another. Based on the feedback we received, we learned that participants were able to identify vulnerabilities in a new report when the same type of vulnerability appeared in a report that they read before. Even though the four subsections of the experiment was administered in the order of "1. None", "2. CWE", "3. Tool Info", and "4. Both" to try to avoid this factor as much as possible, the overlapping type of vulnerabilities can still be concerning. Last but not least, the potential difficulties to go through four different software packages, the potential difficulties to fix different types of vulnerabilities, and even the potential difficulties to fix the same type of vulnerabilities located in different places are confounding factors that this study fails to exclude. One possible solution to this is constructing customized software packages with intentionally added vulnerabilities. It could take much more effort and, again, the strict timeline didn't allow this to be a possible choice.

CONCLUSION

In this study, two research questions were explored. To look into how the knowledge level a person has about software security, and how the features of the reports generated by static code analysis tools can influence their efficiency in fixing security vulnerabilities, a 2×2 mixed-model experiment with the knowledge level as a blocking factor was designed. The presence of two features - CWE information and tool rule description - in vulnerability reports were manipulated and their effects on the efficiency to debug were compared within participants. The effect of the blocking factor, security knowledge-levels, and interaction effects with the above two random factors (presence of two features) on the efficiency to debug are compared between participants. In the experiment, four different software packages and their corresponding vulnerability reports were given to participants. Each of these four reports uniquely contained any one of the four different combinations of the CWE information and tool rule descriptions. There were ten vulnerabilities reported for a package in its report and the efficiency was measured by counting the number of vulnerabilities in the package a participant was able to fix successfully within ten minutes. The result showed that neither the two features in the report nor the participant's knowledge-level had any significant effect on their efficiency in fixing security issues. This could be caused by the limited sample size and multiple confounding factors that this study failed to exclude due to multiple limitations.

It was difficult to answer the second research question about the features of vulnerability reports that could increase a programmer's efficiency in fixing them through an inferential analysis. However, both the feedback from participants during the experiment (interviews) and after the experiment (surveys) shows that the presence of CWE information and tool rule description could make a difference in debugging efficiency. To be more specific, the description based on rules reported by the static code analysis tools could contribute a lot to the thinking process and come up with potential solutions. Also, the CWE information could be used to find external resources online, which could also help with the debugging process. It should be noticed that these conclusions were limited to the tool and the programming language used in this study. More research needs to be done to generalize these findings.

Also, even though our statistical tests showed that the blocking factor, i.e. the software security knowledge-level, did not affect a person's efficiency, a comparison of the mean efficiency between groups given the same features in reports showed that the influence of the two features could be different from one group to another. A more concrete conclusion could be drawn with a larger sample.

ACKNOWLEDGEMENTS

We would like to extend our gratitude to James A. Kupsch, Professor Bilge Mutlu, Professor Barton P. Miller, and instructor Hannah Strohm for their generous help in developing the experiment and their support throughout this project. James specifically offered a significant amount of suggestions on the technical side and taught us about how we could use the platform SWAMP to simplify the experiment of this project.We would also like to thank all the participants of this study for their time and cooperation.

REFERENCES

[1] 2020. Common Weakness Enumeration. (2020). https://cwe.mitre.org/

[2] 2020. Software Assurance Marketplace. (2020). https://continuousassurance.org/

[3] 2020. SpotBugs. (2020). https://spotbugs.github.io/

[4] Hala Assal and Sonia Chiasson. 2019. *'Think secure from the beginning'*: A Survey with Software Developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. ACM Press, Glasgow, Scotland Uk, 1–13. DOI: http://dx.doi.org/10.1145/3290605.3300519

[5] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. 2009. Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?. In *2009 International Conference on Availability, Reliability and Security*. 804–810. DOI: http://dx.doi.org/10.1109/ARES.2009.163 ISSN: null.

[6] 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 (FSE 2016)*. Association for Computing Machinery, Seattle, WA, USA, 1118–1120. DOI: http://dx.doi.org/10.1145/2950290.2983968

[7] 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 (ESEC/FSE 2018). Association for Computing Machinery, Lake Buena Vista, FL, USA, 633–643. DOI: http://dx.doi.org/10.1145/3236024.3236040

[8] 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, Victoria, BC, Canada, 87–96. DOI: http://dx.doi.org/10.1109/VISSOFT.2014.24

[9] F. Cheirdari and G. Karabatis. 2018. Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools. In *2018 IEEE International Conference on Big Data (Big Data)*. 4782–4788. DOI: http://dx.doi.org/10.1109/BigData.2018.8622456

[10] B. Chess and G. McGraw. 2004. Static analysis for security. *IEEE Security Privacy* 2, 6 (Nov 2004), 76–79. DOI:http://dx.doi.org/10.1109/MSP.2004.111

[11] Ciera Christopher. 2006. Evaluating Static Analysis Frameworks. (12 2006).

[12] James Kupsch and Barton Miller. 2009. Manual vs. Automated vulnerability assessment: A case study. 469 (01 2009).

[13] J. Morales, C. Rusu, F. Botella, and D. Quiñones. 2019. Programmer eXperience: A Systematic Literature Review. *IEEE Access* 7 (2019), 71079–71094. DOI: http://dx.doi.org/10.1109/ACCESS.2019.2920124

[14] Emerson Murphy-Hill and Andrew P. Black. 2012. Programmer-Friendly Refactoring Errors. *IEEE Transactions on Software Engineering* 38, 6 (Nov. 2012), 1417–1431. DOI: http://dx.doi.org/10.1109/TSE.2011.110

[15] N. Nagappan and T. Ball. 2005. Static analysis tools as early indicators of pre-release defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 580–586. DOI: http://dx.doi.org/10.1109/ICSE.2005.1553604

[16] Simon Tjoa, Patrick Kochberger, Christoph Malin, and Andreas Schmoll. 2015. An Open Source Code Analyzer and Reviewer (OSCAR) Framework. In *2015 10th International Conference on Availability, Reliability and Security*. 511–515. DOI: http://dx.doi.org/10.1109/ARES.2015.36 ISSN: null.

[17] J. Xie, H. R. Lipford, and B. Chu. 2011. Why do programmers make security errors?. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 161–164. DOI: http://dx.doi.org/10.1109/VLHCC.2011.6070393

[18] Jongwon Yoon, Minsik Jin, and Yungbum Jung. 2014. Reducing False Alarms from an Industrial-Strength Static Analyzer by SVM. In *2014 21st Asia-Pacific Software Engineering Conference*, Vol. 2. 3–6. DOI: http://dx.doi.org/10.1109/APSEC.2014.81 ISSN: 1530-1362.