2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

Identifying Successful Strategies for Resolving Static Analysis Notifications

Justin Smith
North Carolina State University
Raleigh, NC, USA
jssmit11@ncsu.edu

ABSTRACT

Although static analysis tools detect potential code defects early in the development process, they do not fully support developers in resolving those defects. To accurately and efficiently resolve defects, developers must orchestrate several complex tasks, such as determining whether the defect is a false positive and updating the source code without introducing new defects. Without good *defect resolution strategies* developers may resolve defects erroneously or inefficiently. In this work, I perform a preliminary analysis of the successful and unsuccessful strategies developers use to resolve defects. Based on the successful strategies identified, I then outline a tool to support developers throughout the defect resolution process.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

Keywords

Static analysis, Strategies, Human Factors

1. INTRODUCTION

Static analysis tools help developers locate various code defects early in the development process, even before the code executes. For example, static analysis tools detect access control vulnerabilities [13], potential null dereferences [6], and concurrency bugs [14] by analyzing source code. Detecting and, more importantly, resolving defects like these early can prevent more costly failures later in the development process [1].

In their defect reports, static analysis tools provide information to developers in the form of textual notifications. These notifications typically describe possible defects. However, they often fail to fully support developers in actually resolving the defects they detect [8]. Resolving defects often requires developers to orchestrate many interleaving activities. For example, accurately resolving defects can require developers to identify false positives, explore the existing code, invoke additional tools, modify the code, and verify the correctness of their fix, among other activities [12]. Consider one of the most common [2] notifications produced by FindBugs, a static analysis tool. In this case, it does not suggest how to resolve the defect it detects:

> "There is a branch of statement that, if executed, guarantees that a null value will be dereferenced, which would generate a NullPointerException when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception can't ever be executed; **deciding that is beyond the ability of FindBugs.**"

While the tool identifies a problem with the code (a null value may be dereferenced), it does not provide a suggestion for how the developer should validate and resolve the defect. The actions developers take to validate and resolve defects, I refer to collectively as the developer's *defect resolution strategy*. There are many strategies for resolving a given defect. For instance, one strategy for resolving this defect decomposes into three sub-strategies: Google search for information about NullPointerExceptions; use Eclipse's call hierarchy tool to explore which branches get executed; finally, add an additional null check.

I envision an alternative paradigm in which static analysis tools explicate successful defect resolution strategies. To that end, this paper makes the following contributions:

- A preliminary analysis of the successful and unsuccessful strategies developers use to resolve defects detected by static analysis, and
- A description of a tool that explicitly provides developers with successful defect resolution strategies.

2. RELATED WORK

Several researchers have stressed the importance of supporting defect resolution. For example, Path Projection [9] facilitates defect resolution by presenting program path visualizations. Quick Fix Scout [11] performs speculative analysis, enabling developers to preview and compare fixes. Taken alone, Path Projection and Quick Fix Scout each support only one step in the defect resolution process. In contrast, I am working toward supporting developers throughout every step of the defect resolution process.

Previous research has also focused on the importance of supporting successful strategies in the use of complex computer applications. In computer-aided design applications, for example, Bhavnani and John have measured the performance costs of inefficient strategies [3]. Their results suggest the use of more efficient strategies leads to faster task completion time and more accurate results. They also show that offline educational interventions can increase the use of efficient strategies. Leaning on the work of Bhavnani and John, Cockburn and colleagues review interface research and state

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX,USA

ACM ISBN 978-1-4503-4205-6/16/05.

DOI: http://dx.doi.org/10.1145/2889160.2891034

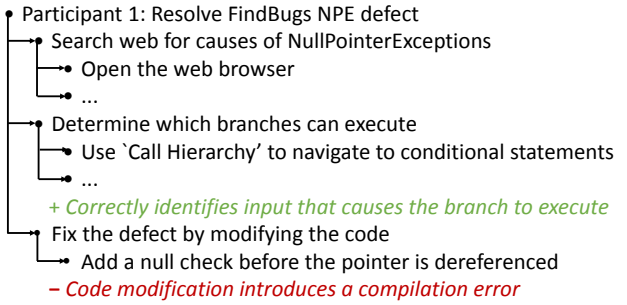

Figure 1: Example strategy tree for the null dereference defect.

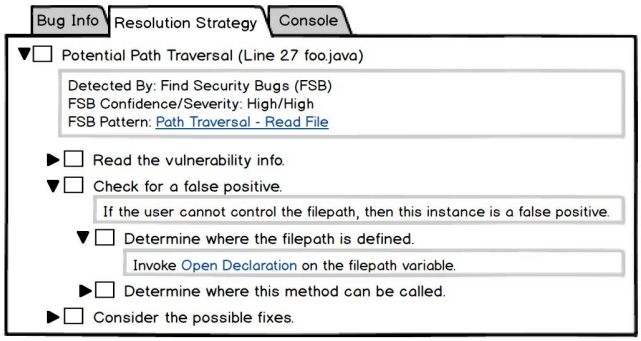

Figure 2: A mockup of a tool that presents successful strategies.

that knowledge of efficient strategies is also an important factor influencing the novice to expert transition [4]. In his work, Cockburn discusses several techniques for designing interfaces that implicitly encourage the use of efficient strategies. These research efforts emphasize the importance of educating users about efficient strategies, however neither proposes to do so by explicitly prescribing effective strategies to users while they complete the relevant task. Building on this previous work, my approach aims to proliferate strategic knowledge by explicitly describing successful strategies.

3. APPROACH

To enable tools to support developers in executing accurate and efficient strategies, I must first identify which strategies are successful. I conducted a study [12] in which I recorded 10 developers resolving four security defects while using Find Security Bugs (FSB) [7] — a security-oriented extension of FindBugs [6]. In this work I reanalyzed the audio and video recordings from [12], identifying those successful and unsuccessful strategies developers used to resolve the defects FSB detected.

Returning to the example of a defect resolution strategy from Section 1, Figure 1 depicts the notation I used to represent defect resolution strategies. This notation draws from the notion of attack trees [10]. Attack trees hierarchically organize the actions an attacker could take to exploit a system. Similarly, this hierarchical representation describes defect resolution strategies as a structured sets of actions. I refer to this notation as a *strategy tree*. Recursively, each child of the root is itself a strategy tree representing the sub-strategies developers used. Finally, to measure success granularly, I annotated a sub-strategy in the tree whenever it either contributed to or detracted from the accurate resolution of the defect — green and red lines prefixed with $+/-$, respectively.

4. RESULTS

Because different developers might use different strategies to resolve the same defect, I constructed 40 strategy trees — one for each top-level strategy I observed in the study. With an average branching factor of approximately four at the roots, these 40 top-level strategies decompose into 155 sub-strategies.

I inspected the strategies for similarity to determine the feasibility of recommending strategies that apply more broadly. The strategies I observed vary depending on the individual developer. For example, the three developers who reported the least familiarity with security vulnerabilities started 11 of the 12 tasks by reading the notification text. In contrast, more experienced developers started by reading the code. The two developers who reported the highest familiarity with security vulnerabilities started only two of eight tasks by reading the notification text.

Though all the strategies I identified exhibit slight differences (such as how developers chose to start resolving the defect), some common patterns promisingly emerged across participants and defects. In all 40 strategies, developers read the code surrounding the defect. Furthermore, in 90% of the strategies developers read the notification text and in 75% of the strategies developers tried to determine if the notification was a false positive.

After examining the strategies based on the degree to which they led to successful outcomes, I observed 62 strategic failures (i.e, instances where a developer's strategy was not efficient or accurate). For example, one developer used Eclipse's call hierarchy tool attempting to locate all the call locations of a `doPost()` method. The tool located several explicit call locations within test classes. However, it did not locate implicit calls that originated from HTML pages. This led the developer to incorrectly conclude that the code was safe, because users did not have access to the vulnerable method. I observed at least one strategic failure in 27 of 40 tasks; across the four tasks, each developer's strategies were undermined by at least one failure. The prevalence of strategic failures, even among the more experienced developers, further motivates this work.

Despite the lack of tool support, some developers resolved some defects correctly using successful strategies. I observed 105 instances where developers' defect resolution strategies successfully led them to make observable progress on the resolution task.

Drawing from the successful strategies I observed, I sketched (Figure 2) a tool that presents explicit defect resolution strategies to developers. The strategy this tool presents is a composite strategy, combining the 10 strategies I observed developers using for one task. Since the developers I studied may not have executed all of the most successful strategies, I supplement the tool by also including defect resolution recommendations published by trusted authorities such as OWASP and CVE [5,15]. The tool also includes checkboxes to allow developers to track their progress. Additionally, the tool exposes other tools (Open Declaration in this example) in contexts when they would be most helpful.

5. CONTRIBUTIONS

My preliminary analysis of developers' strategies suggests that strategic failures may undermine developers in accurately and efficiently resolving defects. I propose a new tool that helps developers resolve defects by providing them with explicit descriptions of successful strategies. In practice, such an approach may improve code quality and also educate developers by increasing their awareness of more successful strategies.[1]

[1]This work is supported by NSF grant number 131832

6. REFERENCES

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, 2008.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8. ACM, 2007.

[3] S. K. Bhavnani and B. E. John. The strategic use of complex computer systems. *Human-Computer Interaction*, 15(2):107–137, Sept. 2000.

[4] A. Cockburn, C. Gutwin, J. Scarr, and S. Malacria. Supporting novice to expert transitions in user interfaces. *ACM Computer Survey*, 47(2):31:1–31:36, Nov. 2014.

[5] CVE. https://cve.mitre.org/.

[6] Findbugs. http://findbugs.sourceforge.net.

[7] Find security bugs. http://h3xstream.github.io/find-sec-bugs/.

[8] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681. IEEE Press, 2013.

[9] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. Path projection for user-centered static analysis tools. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 57–63. ACM, 2008.

[10] S. Mauw and M. Oostdijk. Foundations of attack trees. In D. Won and S. Kim, editors, *Information Security and Cryptology - ICISC 2005*, volume 3935 of *Lecture Notes in Computer Science*, pages 186–198. Springer Berlin Heidelberg, 2006.

[11] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. *SIGPLAN Not.*, 47(10):669–682, Oct. 2012.

[12] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 248–259. ACM, 2015.

[13] T. Thomas, J. Smith, E. Murphy-Hill, B. Chu, and H. Lipford. A Study of Interactive Code Annotation for Access Control Vulnerabilities. In *Proceedings of Visual Languages and Human Centric Computing*, 2015.

[14] Threadsafe. http://www.contemplateltd.com/threadsafe.

[15] Owasp. http://owasp.org/index.php/Main_Page.