Using Checklists to Review Static Analysis Warnings

Nathaniel Ayewah
Dept. of Computer Science
Univ. of Maryland
College Park, MD
ayewah@cs.umd.edu

William Pugh
Dept. of Computer Science
Univ. of Maryland
College Park, MD
pugh@cs.umd.edu

ABSTRACT

Static analysis tools find silly mistakes, confusing code, bad practices and property violations. But software developers and organizations may or may not care about all these warnings, depending on how they impact code behavior and other factors. In the past, we have tried to identify important warnings by asking users to rate them as severe, low impact or not a bug. In this paper, we observe that the user's rating may be more complicated depending on whether the warning is feasible, changes code behavior, occurs in deployed code and other factors. To better model this, we ask users to review warnings using a checklist which enables more detailed reviews. We find that reviews are consistent across users and across checklist questions, though some users may disagree about whether to fix or filter out certain bug classes.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; D.2.4 [**Software/Program Verification**]: Reliability

General Terms

Experimentation, Reliability, Security

Keywords

FindBugs, static analysis, bugs, software defects, bug patterns, false positives, Java, software quality

1. INTRODUCTION

Static code analysis is emerging as an attractive way to detect violations of code quality and security requirements. Static analysis scans code looking for silly mistakes, confusing code, bad practices and property violations. Unlike code review and dynamic testing, static code analysis is automatic and exhaustive, and modern static analysis tools can verify the absence of entire classes of bugs. But static analysis generally cannot reason completely about the intent of a program and tools sometimes conservatively report false warnings. Furthermore developers and organizations may not be interested in every problem found, especially if the problem does not cause deviant behavior [2].

As tool creators and researchers, we would like to know where static analysis is useful in practice, what issues organizations are interested in, and what factors affect the way a warning is perceived by users. One way to do this is to study the artifacts of software development, including code repositories and bug databases, measuring the removal rate of issues and looking for other evidence of interaction with static analysis [2, 6, 7]. These *software mining* efforts can be large and provide significant quantitative trends but are often limited to finding correlations and can only indirectly infer the user's intent. We have to supplement these efforts with direct interactions with users through surveys, interview and lab studies. These user studies give us more qualitative information about what users do and want but are often limited in scope and size (an hence generalizability).

In this paper we use lab studies to explore the aforementioned mismatch between issues found by a static analysis tool and the problems users in different contexts are looking for. In earlier studies, we asked users to give a single review for each warning indicating whether it was severe, low impact, or not a bug [1]. We found that reviewers independently gave the same review when operating on this three-level scale. But a single review may not capture all the factors users have to consider or all the caveats users may have. For example, users may observe that an issue can lead to significant deviations from intended code behavior but they may also conclude that the issue is unlikely to occur in practice. Even if the issue occurs, users may decide that it is not serious enough to send patches to fix deployed code, but that the issue should only be fixed in house for future releases. On the flip side, users may decide that even though a particular warning instance is not serious or not a bug, they still want to be shown similar warnings in the future and may even change the code to remove the warning.

To get more details about some of these factors, we have designed another study in which users provide answers to a *checklist* of questions, not just a single three-level review. This checklist asks users if they think an issue occurs in practice or causes deviation from intended behavior, and when the issue should be fixed or shown in future code reviews. Checklists encourage users to be more consistent about the way they review each warning, and have been used to facilitate and speed up the review process [5]. One eventual goal

DEFECTS'09, July 19, 2009, Chicago, Illinois, USA.

Table 1: Bug Patterns used in controlled study

Bug Pattern	Issue Numbers
NP (Always Null): A NullPointerException is always thrown when the referenced line is executed	1, 2, 3
NP (Null On Some Path): There is a path through the code that, if executed, is guaranteed to throw a NullPointerException	4, 5
NP (Null Parameter Dereference): A method call passes null to an unconditionally dereferenced parameter	6, 7
NP (Unwritten Field): An uninitialized field is read	8
PZLA: Prefer to return a zero-length array instead of null	9
RCN (Redundant Check for Null): Check of a value that is known to be non-null. May indicate a logic error.	10, 11, 12, 13

of our studies is to understand if and when using checklists is a best practice for helping developers review warnings.

This study also examines how certain factors influence reviewers' decision making. These factors include the priority label assigned by the tool, the order in which developers go through warnings, and the programmers perception of the accuracy of the tool. A diligent reviewer should judge each bug on its own merit, but some reviewers may assign more weight to warnings that are high priority, or defer to the tool if they are unsure about the correctness of an analysis. To facilitate this examination we randomized the priority labels and the order in which warnings are displayed. We also introduced some 'fake' warnings (where the analysis is wrong) to see if the user would spot the error or simply assume the tool was correct. Finally we compare the reviews of our lab users with that of some 'expert' users to see if there is a difference in responses on the checklist.

This study found that most warnings solicited consistent answers across reviewers and across checklist questions. This study also saw no discernable impact of the priority label assignment or order on the way users reviewed. Instead user responses correlated the most with the underlying bug pattern, and users were not fooled by fake warnings. Finally this study saw no discernable distinction between the reviews of experts and non-experts.

We describe the study design and procedure in Section 2 and discuss our results in Section 3.

2. CHECKLIST USER STUDY

We designed this study around FindBugs, a static analysis tool that analyzes Java byte code and looks for preestablished patterns of defective code [3, 4]. FindBugs is available as a standalone graphical user interface (GUI) or as a plugin to many popular IDEs and build tools. We used the FindBugs plugin for the Eclipse IDE[1] which displays warnings inline with the offending code. The Eclipse plugin features a custom view in which warnings are displayed in a list with priority labels including high priority (red), normal priority (orange), and low priority (yellow). We modified the plugin to randomize the priority labels and order of warnings. We also used filters to restrict the number of warnings to 13

[1]http://www.eclipse.org

```
636:    private void handleUidl( String argument ) {
637:
638:        //Return all messages unique ids
639:        if( argument == null || argument.length() == 0 ) {
...
655:        }
678:    }
```

FindBugs: "Null pointer dereference of `argument` on line 639"

FindBugs incorrectly asserts that the dereference of 'argument' in the if-statement will throw a NullPointerException

Figure 1: Fake NP Always Null Warning

warnings from the bug patterns in Table 1 (including 3 fake warnings indicated by red shading).

2.1 Fake Warnings

Figures 1 illustrates one of the fake warnings that we inserted. FindBugs incorrectly asserts that `argument` will always be null when it is dereferenced in the if-statement. But careful inspection of the closed-circuit disjunction should convince the reader that `argument` is dereferenced ONLY when it is NOT null. Still some reviewers may assume FindBugs got its analysis correct, especially those who have reviewed a similar warning with correct analysis.

2.2 Using a Checklist

To review each warning, participants completed a checklist (shown in Table 2). The first checklist question tests the reviewer's understanding of the FindBugs warning. Our past research has indicated that most FindBugs warnings should be easy to understand, but some users may feel they need more information (through unit tests). This question also gives users the opportunity to quickly identify those warnings they think are bogus, and hence avoid answering the other checklist questions. Users who do not understand the warning also skip the remaining checklist questions.

The four checklist questions that follow give reviewers multiple scales for measuring the severity of the warning and the level of their response. Reviewers indicate severity in terms of how often the issue occurs and how the issue affects code behavior. Reviewers indicate the level of their response by indicating whether they would fix the bug (and in what contexts) and whether they would filter this bug pattern out of future reviews.

The checklist responses all range from strong responses (e.g. substantial deviation) to weak responses (e.g. minor deviation) to negative responses (e.g. no deviation). This design is useful when we analyze the results because it allows us to compare the different checklist questions (by considering only strongest responses, for example).

2.3 Study Participants

We recruited 12 students to participate in this study, 11 graduate students and 1 undergraduate student. All students had experience programming in Java and using the Eclipse IDE, but few were familiar with FindBugs. Only 2 students had experience with FindBugs and only 4 students had experience with any fault detection tools.

This study used a within-subjects design. All participants saw the same warnings and other variables were randomized. The application under review was Java Email Server (Version 1.6.1) an SMTP and POP3 email server[2]. The ap-

[2]http://ericdaugherty.com/java/mailserver/

Table 2: Checklist Questions for each Issue

ISSUE UNDERSTANDING: Which of the following statements best describes your understanding of the problem?

- I have enough information to understand this problem
- I need to write a test case to better understand this issue
- I think this is a bogus issue which cannot occur and does not affect code behavior
- I do NOT understand this issue

ISSUE OCCURRENCE: Under what circumstances can the behavior described by this issue occur?

- Under normal, intended use
- Only in situations that do not appear to be among intended use cases
- I do NOT think it can occur at all

CODE BEHAVIOR: What is the apparent impact of the issue on the behavior of the code?

- It behaves in a way clearly at substantial odds with the intended behavior
- It does NOT behave as intended, but difference does NOT appear to be substantial
- No apparent difference in behavior

FIX DECISION: What do you recommend?

- Definitely change the code to fix the problem
- Change the code only if risk of impacting deployed code is not high
- Change the documentation to make code clearer
- No changes necessary, code is OK

FILTERING DECISION: Would you want a static analysis tool to show you issues like this?

- Yes, definitely, even in old code
- OK, particularly in new code, or if there aren't a lot of them
- I'd rather not bother looking at such issues

plication was modified to insert more warnings (including some fake warnings) so we could have a more diverse range of null pointer warnings within the small application. All the inserted code was derived from real warnings seen in other applications.

2.4 Comparing with Expert Participants

We compared the results from the student participants to results from more experienced participants. These experienced reviewers are real FindBugs users, recruited from a FindBugs-interest mailing list, and asked to perform the study remotely using a Java Web Start interface to access the warnings and submit their reviews. In other words, the more experienced users were not in a controlled environment, but their opinions are solicited to compare with the student users.

The Java Web Start interface had some limitations. Users could not drill down into the source code to get more details and some users had trouble starting the interface. This expert review served as a test drive of an automated review system that we wish to use for future larger studies.

Table 3: Issue Understanding vs Bug Patterns

	NP	PZLA	RCN	Fake
Understand Real Bug	63	12	30	25
Understand with Test Case	4	0	2	2
Bogus Warning	5	0	3	9
Don't Understand	0	0	1	0

2.5 Study Procedure

The first step in the study was a tutorial which was designed to introduce participants to FindBugs, the Eclipse plugin environment, and the checklist. Participants were informed that they would answer the checklist for each of the 13 displayed warnings, but they were not told the the priority labels and order of the warnings had been randomized, nor were they told that some of the warnings were bogus warnings.

Following the tutorial, all participants performed a practice review so they would be familiar with the experiment environment and go through the survey once. In the subsequent main session, participants were asked to perform their reviews at their own pace – they were not timed. After the study, participants were asked to fill out a brief survey to get more information about their background and collect qualitative information about their experience during the study.

3. RESULTS

Most of our analysis focus on the four review questions, not on the issue understanding question. As Table 3 shows, most users understood most of the issues. In the case where a user rated an issue as a bogus warning, the negative response is automatically entered for the four review questions. One observation is that the 3 fake warnings received a bogus warning only 9 out of 36 times. But as our discussion in Section 3.2.1 will reveal, reviewers were not fooled by the fake warnings because they gave them the most negative responses. The low count on the issue understanding question likely indicates that users misunderstood the question and assumed it only referred to how much they understood the static analysis warning.

3.1 Consistency of reviews

There are two types of consistency we are interested in. One is the consistency across reviewers for each checklist question; in other words, how much do reviewers agree with each other when they review an issue. The second is the consistency across checklist questions; in other words, do reviewers give tend to give a strong response on one question but a weak or negative response on another question for the same issue.

To consider the consistency across reviewers, we count the number of times they agree for each question and issue in Table 4. (For example, the table shows that 8 reviewers agreed on the issue occurrence decision question for the first issue.) In 30 of the 52 cases, 8 or more reviewers agreed, and all issues had at least one question in which 8 or more reviewers agreed. But only three issues had 8 or more agreements for all questions (issues 2, 7 and 8). This highlights the idea that the consistency across reviewers for a particular issue depends on what question they are trying to answer. For example, all users agree that issue 10 (a redundant check

Table 4: Level of Agreement for Each Issue

Issue #	Occurs	Behavior	Fix	Filter
1	8	7	8	8
3	6	9	11	10
5	8	10	8	7
6	5	7	7	8
7	8	8	8	8
8	10	12	10	11
9	10	5	7	7
10	7	12	6	6
12	5	8	6	6
13	8	7	8	6
2	10	11	10	9
4	8	8	7	4
11	6	12	7	7

Table 5: Correlation Coefficients for Checklist Responses

	Occurs	Behavior	Fix	Filter
Occurs	1	.69	.75	.75
Behavior		1	.80	.80
Fix			1	.75
Filter				1

for null) does not cause deviation from intended behavior, but are split on whether to fix or filter this issue.

Table 4 also supports our investigation into the consistency across checklist questions by shading each cell, with darker shades representing stronger reviews. The results show agreement at the strongest level for most questions among the real issues and agreement at the weakest level for the fake issues. The exception is with the redundant check for null issues (10 - 13) where users rate the issues as normally occurring, but give weak responses to the other questions.

Another way we measure consistency across checklist questions is to count the number of reviews in which all four questions got exactly the same level of review. Out of 156 reviews, 82 (or 53%) presented exactly the same level of response for all four questions, and 123 (or 79%) had all questions at the same level or off by one.

Another measure we use is Pearson's correlation coefficient for each pair of questions for all reviews (see Table 5). To enable this, we encode the checklist responses numerically from 3 to 1 with 3 representing strongest responses. While this is not a perfect measure, the high positive correlations do suggest that most reviews were consistent across questions.

3.2 Factors Affecting Reviews

In this section we consider how different bug patterns, displayed priorities and presentation orders affect the reviews of issues. One way to do this is to consider the number of strong responses for each factor level relative to the number of strong responses across all factor levels.

Consider Table 6 where the columns represent different bug pattern groups and the rows represent the number of strong responses for each checklist question. The last column (labeled *Agg*) aggregates the number of strong responses

Table 6: Strongest Checklist Reviews vs Bug Pattern Groups

	NP	PZLA	RCN	Fake	*Agg*
# of Reviews	72	12	36	36	156
Normally Occurs	44	10	20	9	83
chi-test ($p < 0.05$)	N	(+)	N	(-)	53%
Substantial Deviation	53	3	9	4	69
chi-test ($p < 0.05$)	(+)	N	(-)	(-)	44%
Always Fix	52	3	10	5	70
chi-test ($p < 0.05$)	(+)	N	(-)	(-)	45%
Always Show	52	5	10	8	75
chi-test ($p < 0.05$)	(+)	N	(-)	(-)	48%

across all bug patterns for each question. For example, 83 out of 156 reviews (53%) gave strong responses to the Issue Occurrence question. We go on to calculate the proportion of reviews giving strong responses for each bug pattern group. For example, 44 out of 72 reviews (61%) for the NP bug pattern group gave strong responses to the Issue Occurrence question. To test the significance of this relative to the overall strong response rate of 53%, we do a chi test comparing the number of strong responses (44) and remaining responses (28) to the expected value for these two quantities (based on the overall rate). In Table 6, we indicate the result of the chi test with a blue-shaded (+) or a red-shaded (-) if the ratio for the bug pattern is significantly greater than or less than the ratio for all bug patterns respectively, or with N if there is no significant difference. The chi test is limited because the size of some factors is quite small, but this allows us to visualize some general trends. We are also assuming the factor levels are independent of each other.

3.2.1 Bug Pattern and Fake Warnings

The results in Table 6 indicate an effect due to the bug pattern group: NP issues were more likely to receive strong responses while RCN and Fake issues were less likely to receive strong responses. We can further break down the NP bug group into distinct bug patterns. Table 7 shows that the Always Null and Read of Unwritten Field patterns had many strong responses, while the Null on Some Path and Dereference of Null parameter had fewer strong responses. The low count of strong responses for Fake issues indicates that users were not fooled by the fake issues and did not assume the analysis was correct.

3.2.2 Displayed Priority and Presentation Order

Table 8 shows another analysis, this time with the priority label displayed next to each issue. The results indicate that while the rate of strong responses was slightly higher for issues with a high priority label, the difference does not appear to be significant. Similarly, when we construct a table in which columns represent all strong responses for a particular index in the presentation order, we observe no significant difference due to ordering.

3.3 Comparison with Expert Participants

In this section we briefly describe the results of an online study for regular users of FindBugs, or "expert users". The patterns observed among experts was generally similar to that of regular participants. In other words, NP warnings

Table 7: Strongest Checklist Reviews vs NP Patterns

	Always Null	Null on Some Path	Param Deref	Unwritten Field
# of Reviews	24	12	24	12
Normally Occurs	13	8	13	10
chi-test ($p<0.05$)	N	N	N	(+)
Subst. Deviation	16	10	15	12
chi-test ($p<0.05$)	(+)	(+)	N	(+)
Always Fix	19	8	15	10
chi-test ($p<0.05$)	(+)	N	N	(+)
Always Show	18	7	16	11
chi-test ($p<0.05$)	(+)	N	N	(+)

Table 8: Strongest Checklist Reviews vs Displayed Priority

	High	Normal	Low
# of Reviews	58	46	52
Normally Occurs	28	27	28
chi-test ($p < 0.05$)	N	N	N
Substantial Deviation	28	23	18
chi-test ($p < 0.05$)	N	N	N
Always Fix	29	22	19
chi-test ($p < 0.05$)	N	N	N
Always Show	29	24	22
chi-test ($p < 0.05$)	N	N	N

were rated more strongly, and users were not fooled by fake warnings. (The experts' study did not randomize the order or displayed priority.) Reviews were also consistent among participants.

Experts gave strong responses at a slightly lower rate than regular participants, though this was not statistically significant except for the Issue Occurrence question. On the other hand, experts selected the "Always Show" filtering decision at a slightly higher rate, though also not statistically significant.

3.4 Qualitative Feedback from Reviewers

Most participants indicated in the post experiment survey that the warnings were generally easy to understand and were not new to users. This is not surprising since the warnings selected, and FindBugs warnings in general, tend to refer to simple errors. Users were more split about whether it was easy to decide if an issue was a bug or if it should be fixed. Half the users said it was easy to decide both of these properties, while the rest disagreed or were indifferent. One user commented that for some warnings (like the redundant null check warnings) it was difficult to decide whether to fix or not, because he was concerned about possible side effects in other parts of the code. Half the users also indicated that the displayed priority did not influence their review, though some users complained that the color coding for priority labels (red, orange and yellow) made it hard to distinguish between them.

3.5 Threats to Validity

Lab studies like this always have an external validity problem; it is unclear how much the results generalize. This is particularly pronounced in studies of static analysis warnings because, as our results indicate, the choice of bug patterns affects the responses received. We also believe that in practice some of the checklist responses would be impacted by factors not considered in this study including the policies of the reviewer's organization.

4. CONCLUSIONS

In future work, we plan to get a large number of developers to review (and possibly fix) many warnings. Some reviews will necessarily be brief, especially if the developer chooses to fix the code to remove the issue. Other reviews will benefit from our experience using checklists in this study to get more detailed information. We will aim to get multiple independent reviews for each warning, and this study suggests that independent reviews using checklists will be consistent.

Beyond the themes explored by the checklists in this study, we hope in future work to ask users to do more detailed forensic work to identify whether bugs are causing code to misbehave and why they are missed by the developer's quality assurance process.

5. REFERENCES

[1] N. Ayewah and W. Pugh. A report on a survey and study of static analysis users. In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, New York, NY, USA, 2008. ACM.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, New York, USA, 2007. ACM.

[3] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM.

[4] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–14, New York, USA, 2007. ACM.

[5] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. Path projection for user-centered static analysis tools. In *PASTE '08: Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 57–63, New York, NY, USA, 2008. ACM.

[6] S. Kim and M. D. Ernst. Which warnings should i fix first? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54, New York, NY, USA, 2007. ACM.

[7] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 580–586, New York, NY, USA, 2005. ACM.