A Report on a Survey and Study of Static Analysis Users

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

As static analysis tools mature and attract more users, vendors and researchers have an increased interest in understanding how users interact with them, and how they impact the software development process. The FindBugs project has conducted a number of studies including online surveys, interviews and a preliminary controlled user study to better understand the practices, experiences and needs of its users. Through these studies we have learned that many users are interested in even low priority warnings, and some organizations are building custom solutions to more seamlessly and automatically integrate FindBugs into their software processes. We've also observed that developers can make decisions about the accuracy and severity of warnings fairly quickly and independent reviewers will generally reach the same conclusions about warnings.

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

When we create static analysis tools, we often focus on the problems we find, and our efficiency in finding these problems. Some of our tools are now becoming more popular as developers and managers recognize their value. FindBugs, an open source static analysis tool for Java that recognizes many patterns of faulty code, now has over 500,000 downloads [9, 14, 8].

There is still more work to be done to find more bugs more efficiently. But as organizations start to integrate these tools into their software processes, questions arise about their usability and their impact on developers and processes. How do organizations maximize their return on investment? How do tools affect programmer productivity, particularly in light of some of the false warnings they produce?

Many organizations recognize that tools, though imperfect, provide another layer of protection against quality and security flaws. Their developers may not place the same priority on tools. Hence organizations need policies to ensure the tools are being used and need standards to ensure consistency across all users. These could include rules about how soon warnings must be fixed, processes for classifying and filing warnings, and tools to suppress warnings that are not relevant so they do not keep coming back in future runs. What policies have organizations adopted and how effective have they been? If organizations do not adopt effective practices for using static analysis tools they may not retain all their benefits, and some may discontinue use of these tools because of an incorrectly perceived lack of value.

The FindBugs project has started a research project which aims to identify and evaluate tool features, validate or invalidate assumptions held by tool vendors, and provide guidance for individuals and teams wanting to use static analysis tools effectively. To this end, we have conducted some online surveys and informal interviews through which we have observed that though FindBugs traditionally emphasizes higher priority warnings, many users are interested in low priority warnings in certain categories. We have also observed that though many organizations do not have formal policies for using FindBugs, some organizations do want to encourage more regular and automatic use among their developers, but have to first jump over an initial hump due to the quantity of warnings found the first time they run the tool and the time it takes to build custom solutions to integrate FindBugs into their software processes. These and other findings are summarized in Section 2 and described in more detail in [6].

We have also conducted a preliminary controlled study (described in Sections 3 and 4) in which we observe users performing basic auditing tasks as they interact with some static analysis tools. We have observed that independent reviewers can make decisions about the accuracy and severity of warnings fairly quickly and will generally reach the same conclusions about warnings. This preliminary study gives us qualitative information that will inform future more comprehensive studies.

2. FINDBUGS SURVEY AND INTERVIEWS

To begin this research we have conducted a major online survey of FindBugs users. The survey received over 400 responses between November 2007 and April 2008, and was prepared and delivered using Survey Monkey [5] and advertised on the FindBugs web site and mailing lists. Most of the survey's 29 questions are presented as multiple-choice to make it easier to quantify and analyze the responses. The survey includes questions on how and how often users interact with FindBugs, what policies are in place for dealing with bug reports, how false positives are identified and handled, and some demographic questions. The survey also probes user opinions on how much value they get out of it and what warnings are most important.

We have also conducted 12 informal phone interviews with consenting survey participants in the US and Europe to better understand their context and to get more detailed information about their experiences, challenges and suggestions. We do not use the surveys and interviews to draw scientific conclusions but to provide qualitative insights and anecdotes about user practices and identify areas of future research. Our findings are summarized here but described in more detail in [6].

First, some background information on FindBugs. FindBugs is composed of numerous detectors, each of which report specific patterns of bugs. The detectors are written in Java and use heuristics to search for bug patterns, group them into categories (e.g. correctness, bad practice, performance and internationalization) and assign a priority of high, medium or low to each. The priority levels are not comparable across bug patterns, but FindBugs' developers have put in considerable effort to ensure that the warnings reported at higher priority levels are ones users will want to fix.

Our survey results indicate that most users are interested in high priority warnings in all categories but a surprising number of users also review lower priority warnings, though the review categories vary from user to user. This indicates that while high priority warnings are relevant to most users, lower priority warnings may also be relevant depending on the user's context. FindBugs does not show low priority warnings by default but it may be beneficial to some users to include low priority warnings in some categories. One suggestion is that FindBugs provide preset configurations that apply different detectors to users in predefined contexts. A user working on a web application, for example, has different needs from a user working on a desktop application.

Another observation from our surveys and interviews is that many users do not yet have formal policies for using FindBugs. As a result they may not run it for several weeks, or they may place more emphasis on warnings close to the release date, when there is more pressure to ignore minor warnings. During our interviews, we talked to many users who recognize that they need to integrate static analysis tools into their formal processes to make them more effective. But many of these users have to overcome barriers to this level of adoption, and in doing so, each organization is often reinventing the wheel. Some barriers mentioned by users include:

- A significant initial effort is needed to eliminate warnings found the first time the tool is run. During this initial effort developers may discover that certain bug patterns are not relevant, and may want to filter them out for future runs.
- Users often need to aggregate warnings from different tools, from unit tests, and from other quality assurance methods into one report or database and institute policies over these resources.
- Tools may need to be customized to fit the needs of the project. Customizations include turning on/off certain detectors in certain parts of the code, and creating new project-specific or organization-specific bug detectors.

3. CONTROLLED STUDY

To fully understand how users interact with tools, we need to observe real users as they work. Like driving a car, working with software tools involves activities that a user is not conscious of. These activities may not come out in surveys and open ended interviews. Eventually we plan to observe developers in their natural context and model their practices in hopes of better understanding the impact of static analysis tools. In the mean time, we have conducted a controlled study to measure some basic aspects of user interaction with static analysis tools and gain insights that will inform future on-site studies.

In this study, users were recruited and asked to review a small set of warnings from FindBugs and Fortify Source Code Analyzer [2]. Fortify SCA is a commercial static analysis tool that specializes on finding potentially exploitable security vulnerabilities in source code. Fortify SCA is part of a suite of tools (that includes dynamic and real time analyzers) for vulnerability detection. We hope to conduct future studies that include other tools.

During the review users were expected to choose a designation indicating the severity of the warning. We measured the amount of time users spent reviewing each warning and compared the conclusions of different users. But the primary goal of this study is to get qualitative information about how users interact with the tools when making these decisions.

3.1 Participants

We recruited 12 students (10 graduate and 2 undergraduate) from the University of Maryland's Computer Science Department using email, fliers and word of mouth. Users had between 1 and 10 years of experience programming with Java (the average was 6 years) and between 0 and 5 years of experience using the Eclipse IDE (the average was 3 years). Only three of the users had used any static analysis tools and none had any experience with the tools they used during the study.

3.2 Experiment Design

Our experiment used a between-subjects design: each participant only interacted with one of the two tools. This is in part because of the time needed to train users to use the tools and the desire to keep each experiment at around 1 hour. In particular, we trained the participants interacting with Fortify SCA on some important security vulnerabilities (see Section 3.3). We did not require our users to have any prior experience using static analysis tools. The first six users were assigned to the FindBugs experiment while the next six did the Fortify Study.

Both tools were run on DSpace (version 1.4.2), an open source web based application for accessing and managing text, audio, video and other resources generated during research and teaching [1]. DSpace was one of the benchmarks in a recent Static Analysis Tool Exposition organized by the National Institute of Standards and Technology (NIST) [4]. Both FindBugs and Fortify SCA participated in the exposition.

Participants were asked to review 23 FindBugs warnings or 21 Fortify SCA warnings. The FindBugs issues included 8 High priority warnings and 15 Medium priority warnings. Most warnings were Correctness warnings but there were also 6 Bad practice and 2 Multithreaded correctness warnings. The Fortify SCA issues included 13 High priority warnings and 8 Medium priority warnings. The high priority warnings included 6 HTTP Response Splittings, 3 SQL Injections and 4 Race conditions involving servlets with member fields.

One consideration in selecting warnings is how many similar warnings are included. Often when a user reviews a particular type of warning, similar warnings can be reviewed quicker. We decided in most cases to retain only one or two warnings from each cluster of similar warnings, but the review times may be different if we designed a study with many similar issues.

With these considerations accounted for, warnings were randomly selected from both high and medium priority categories, with more emphasis on high priority issues. Participants were also given four different and unrelated warnings to practice on so that they would be familiar with the user interface before the start of the study. Users were asked to rate warnings on a 3 level scale using labels native to the tools. For FindBugs the levels were “Must Fix”, “Low Impact”, and “Not a Bug”. For Fortify SCA the levels were “Exploitable”, “Suspicious”, and “Not an Issue”. We left it up to the users judgment to decide what criteria to use to designate warnings into each level, and queried users afterwards about their choices.

The experiments were conducted using the Eclipse IDE and corresponding plugins for both tools. To facilitate post-experiment analysis we logged some user actions (such as selecting a view or rating a warning) using a customized version of the HackyStat Eclipse plugin [3, 11], which transparently collects data about user activities and sends it to a central repository.

3.3 Experiment Procedure

The experiment was divided into four parts: a tutorial, a practice session, the main session and a background survey.

During the tutorial, participants viewed a web page which described the tools with illustrations and outlined the tasks the user was expected to perform. In particular, the tutorial showed users how to navigate through warnings, designate a rating to each one, and add comments. The Fortify SCA tutorial also included a checklist of steps for users to follow. This was intended to reduce the complexity of some of the tasks and to ensure participants consider all relevant factors before choosing a designation. An example of a checklist for SQL Injections is shown in Figure 1. Participants were encouraged to ask any questions they had to the experimenter.

During the practice session, participants are asked to review four warnings (different from and unrelated to the warnings in the main session). Participants “thought out loud” as they performed the review to provide qualitative information about what decisions they were making and why.

Use the following Checklist to determine if a segment of code is an SQL injection

1. Does data enter the program from an untrusted source? If NO, then not an SQL injection
2. Is the data used to construct an SQL query? If NO, then not an SQL injection
3. Is the data validated between its entry and where the constructed SQL statement is executed? If YES, then GOTO 3.a., otherwise GOTO 4
   (a) Is the data validated using blacklisting (removing or escaping potentially malicious characters)? If YES, then code is still vulnerable to SQL injection because blacklisting is not as effective
   (b) Is the data validated using white listing (only allow certain predetermined inputs)? If YES, then not an SQL injection
4. Do you see any other security mechanism to prevent SQL injection? If YES, then use your best judgment to determine if the security mechanism is effective

Figure 1: SQL Injection Checklist

Table 1: Review Times for FindBugs and Fortify SCA

FindBugs	Fortify SCA
73	88
76	90
90	103
97	108
98	143
151	189
Average: 98	120

Once participants completed this review and were comfortable with the interface the experiment moved to the main session.

In the main session participants reviewed the assigned issues starting with the highest priority warnings. This mimics the way tools usually present the warnings to users. This session was timed and the screen and audio were recorded. The main session was followed by a brief online survey.

4. RESULTS AND DISCUSSION

4.1 Review Times

Table 1 shows the review times for FindBugs and Fortify SCA. The times for each tool are sorted from shortest to longest. Users spent an average of 98 seconds reviewing each FindBugs warning which drops to about 87 seconds when the last (outlier) is excluded. Users spent about 120 seconds for each Fortify SCA issue.

4.2 Comparing User Designations

Table 2 summarizes user designations for the warnings in FindBugs and Fortify SCA respectively. Each entry corresponds to a single warning and represents the number of users that reviewed it at the three different levels. For example the entry (6,0,0) means all users considered the warning to be Exploitable/Must Fix. High and medium priority warnings are shown in separate columns and entries are ordered such that those near the top of the table were consid-

Table 2: Summary of User Designations for each FindBugs Warning (Must Fix, Low Impact, Not A Bug) and each Fortify SCA Issue (Exploitable, Suspicious, Not an Issue)

FindBugs		Fortify SCA	
High Priority	Medium	High	Medium
6, 0, 0	6, 0, 0	6, 0, 0	3, 3, 0
6, 0, 0	6, 0, 0	6, 0, 0	2, 4, 0
6, 0, 0	5, 0, 1	6, 0, 0	2, 1, 3
5, 0, 1	5, 0, 1	5, 0, 1	1, 5, 0
4, 2, 0	4, 2, 0	4, 2, 0	1, 2, 3
3, 2, 1	4, 2, 0	4, 2, 0	1, 2, 3
3, 1, 2	4, 0, 2	3, 3, 0	0, 5, 1
2, 4, 0	2, 4, 0	3, 3, 0	0, 4, 2
	2, 4, 0	3, 3, 0	
	2, 4, 0	3, 2, 1	
	1, 5, 0	3, 2, 1	
	1, 5, 0	3, 0, 3	
	1, 4, 1	1, 1, 4	
	0, 6, 0		
	0, 6, 0		

ered more severe by more users and those near the bottom were more likely to be reviewed as low impact or not an issue. This ordering is based on a simple heuristic in which we treat each entry as a 3-digit number.

Of the 23 FindBugs warnings, there are 12 where at least 5 users agreed on a designation and 21 where at least 4 users agreed. One question that arises is whether users disagree because the code and warning are unclear, or if it is because users miss important details. The high level of agreement among FindBugs reviewers may indicate that the cases and warnings are generally clear.

One interesting exception occurred during the practice session and is illustrated in Figure 2. Here, a switch statement is missing breaks and each case falls through to the next one. FindBugs flags this as a bug but 4 users concluded that the programmer intended the fall through to initialize all variables. The other 2 users reviewed this as Must Fix, and may have not noticed the programmers possible intent. Of course, the programmer in this example should probably insert comments indicating that breaks were omitted intentionally if in fact this is the case. In another case, FindBugs flagged a possible null pointer dereference that would only occur if an earlier exception was thrown (and caught). The programmer commented in the catch-block that the exception "should never happen", but 4 users still concluded that the warning was a "Must Fix" while the other 2 reviewed this as Not an Issue.

Of the 21 Fortify warnings, there are 6 where at least 5 users agreed on a designation and 11 where at least 4 users agreed. Some of the disagreement may have resulted from reviewers getting confused as they went through the trace. In one case half the users rated a HTTP Response Splitting warning as Exploitable while the other half rated it as Not an Issue. The comments indicate that some of those who thought it was not an issue concluded that the offending variable was sanitized using the `URLEncoder.encode` method, but in fact a different variable was sanitized.

```
public DCDate(String fromDC) {
    ...
    switch (fromDC.length()) {
    case 20:

        // Full date and time
        hours = Integer.parseInt(fromDC.substring(11, 13));
        minutes = Integer.parseInt(fromDC.substring(14, 16));
        seconds = Integer.parseInt(fromDC.substring(17, 19));

    case 10:

        // Just full date
        day = Integer.parseInt(fromDC.substring(8, 10));

    case 7:

        // Just year and month
        month = Integer.parseInt(fromDC.substring(5, 7));

    case 4:

        // Just the year
        year = Integer.parseInt(fromDC.substring(0, 4));
    }
    ...
}
```

Figure 2: Switch statement with no breaks. Some users concluded this was not a bug while others declared this a Must Fix.

4.3 Qualitative Results

A brief survey administered after each study captured more feedback about the user's experience. In the survey 5 of the 6 FindBugs users indicated that they generally understood the warnings or that they were familiar with the problems from previous experience, while 4 of 6 users indicated that it was not difficult to decide if a warning represented a bug. But users were split over whether it was easy to distinguish between "Must Fix" and "Low Impact" bugs. Some of these users complained that they were not familiar with the code and could not investigate too deeply, so it was hard to decide the real impact of the warning.

In the Fortify SCA survey, 5 of 6 users indicated that they understood the warnings, but most still thought it was difficult to decide if a warnings was a bug. In addition 5 of 6 users found it hard to distinguish between "Exploitable" and "Suspicious" issues. Some users said they were conservative, rating as Exploitable any issue for which a reasonable chance of failure existed.

Both FindBugs and Fortify SCA provide a clickable trace for each warning that contains links to relevant parts of the code. FindBugs trace links to affected fields and classes and the line where the warning occurs. Fortify SCA's traces contained a call hierarchy tracing the cause of each issue from the source (e.g. where a taint enters the program) to the target where the vulnerability is exposed. Fortify SCA's traces were much longer than those in FindBugs and users relied more on the traces in Fortify SCA to understand the warnings.

We observed that users often looked beyond the trace, referring to the type hierarchy or just doing a text search to find out more about variables and types. Future studies could identify all the information used by users during an audit and encourage tools to make this information more readily available.

Some reviewers may have found the traces confusing—their comments indicate that they focused on the wrong variable and hence rated the warning as not a bug. Other users indicated that even after going through the trace to confirm the warnings, they did not know whether to trust the tools. Such users would spend some time trying to find a clue that might suggest that the tool was wrong. One Fortify SCA user indicated that they were not confident enough to rate any issues as Not an Issue (there was also one FindBugs reviewer that did not rate any warnings as Not a Bug).

4.4 Threats to Validity

Our goal in this experiment was to mimic the experience software developers have as they review warnings. One obvious limitation is that users were not familiar with the code base and may have drawn some incorrect conclusions. Another is that we used the default ordering of warnings provided by the tools for all users (though users were not constrained by this order), and hence the position of a warning near the beginning or end of the list may affect the way users reviewed it. But the results of this study give us clues of some of the issues we should look for in an on-site study, such as how much users rely on traces, and how often reviewers err or disagree.

5. RELATED WORK

Recent research has examined users of static analysis tools (and the artifacts they create) to better understand how users work. In an earlier paper, we investigated why real bugs sometimes go unfixed [7]. We determined that some of these were deliberate errors (inserted perhaps as a surrogate for throwing an exception), masked errors (that have no impact on the function of the program because they are masked by some other code), and errors on infeasible paths. Other research has examined why developers do not always fix warnings that are reported moments after the code is written (when they are cheap to fix) and concluded that the reports need to match the developer's current goals and fit into the developer's workflow [12].

Recent research has also focused on the needs of organizations. Researchers at eBay tried to calculate the return on investment of FindBugs and other tools by comparing their cost/benefit with that of manual testing [10]. They conclude that FindBugs had a compelling value proposition relative to manual testing. Other researchers have tried to attach a value to specific warnings before they are human reviewed. These researchers sampled static analysis warnings from Google's code base (with notes on which ones were resolved) and used this to build models to predict whether a warning is a false positive or not [13].

6. CONCLUSION AND FUTURE WORK

Our current and future research focuses on the understanding how users work so we can build tools that better support them. So far we have observed that general tools like FindBugs have diverse users that are interested in different categories of warnings, even at low priorities. We have also identified some formal policies instituted by organizations to use static analysis tools more effectively and observed in controlled studies that independent reviewers generally make similar judgments about the severity of warnings. In the future, we plan to observe our users in their natural contexts to more effectively model their work practices. This direct interaction with developers and quality assurance specialists will help us identify how static analysis tools enhance or interfere with their work.

7. REFERENCES

[1] Dspace. http://dspace.org/, 2008.

[2] Fortify software. http://fortify.com/, 2008.

[3] Hackystat. http://hackystat.org, 2008.

[4] Static analysis tool exposition, organized by software assurance metrics and tool evaluation (samate) project at nist. http://samate.nist.gov/index.php/SATE, 2008.

[5] Survey monkey. http://surveymonkey.com/, 2008.

[6] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Experiences using static analysis to find bugs. *Software, IEEE*, 25(5), 2008. To appear.

[7] 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.

[8] 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.

[9] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19, New York, NY, USA, 2005. ACM Press.

[10] C. Jaspan, I.-C. Chen, and A. Sharma. Understanding the value of program analysis tools. In *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 963–970, Montreal, Quebec, Canada, 2007. ACM.

[11] P. Johnson, H. Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, S. Zhen, and W. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 641–646, 2003.

[12] L. Layman, L. Williams, and R. S. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 176–185, 2007.

[13] J. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proceedings of the International Conference on Software Engineering*, 2008.

[14] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136, New York, NY, USA, 2006. ACM Press.