Experiences Using Static Analysis to Find Bugs

Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh

Abstract—Static analysis examines code in the absence of input data and without running the code, and can detect potential security violations (e.g., SQL injection), runtime errors (e.g., dereferencing a null pointer) and logical inconsistencies (e.g., a conditional test that cannot possibly be true). While there is a rich body of literature on algorithms and analytical frameworks used by such tools, reports describing experiences with such tools in industry are much harder to come by.

We describe FindBugs, an open source static analysis tool for Java, and experience using it in production settings. FindBugs does not push the envelope in terms of the sophistication of its analysis techniques. Rather, it is designed to evaluate what kinds of defects can be effectively detected with relatively simple techniques and to help us understand how such tools can be incorporated into the software development process. FindBugs has been downloaded more than 580,000 times and used by many major companies and software projects.

We report on experience running FindBugs against Sun's JDK implementation, using Findbugs at Google where it has been used for more than a year and incorporated into their standard development process, and preliminary results from a survey of FindBugs users.

Index Terms—Static analysis, FindBugs, code quality, bug patterns, software defects, software quality

1 INTRODUCTION

SOFTWARE quality is important, but often imperfect in practice. Many different techniques are used to try to improve software quality, including testing, code review, and formal specification. FindBugs is an example of a static analysis tool that looks for coding defects [1], [2], [3]. These tools evaluate software in the abstract, without executing them or considering a specific input.

Rather than trying to prove that the code fulfills its specification, static analysis tools look for violations of reasonable or recommended programming practice. Thus, they look for places where code might dereference a null pointer or overflow an array. Tools might also flag an issue such as a comparison that can't possibly be true; while the comparison will not cause a failure or exception, the existence of such a comparison may suggest that it might have resulted from a coding error, leading to incorrect program behavior. Some tools also flag or enforce programming style issues, such as naming conventions or the use of curly braces in conditionals and looping structures.

The lint program for C programs [4] is generally considered to be the first widely used static analysis tool for defect detection, although by today's standards it is rather limited. There has been a huge amount of work in the area over the past decade, driven substantially by concerns over defects that lead to security vulnerabilities, such as buffer overflows, format string vulnerabilities, SQL injection and cross site scripting. There is a vibrant commercial industry in advanced (and expensive) static analysis tools, [5], [6] and a number of companies have their own proprietary in house tools, such as Microsoft's PREfix tool [7] Many commercial tools are very sophisticated, using deep analysis techniques. Some static analysis tools can use or depend upon annotations that describe invariants and other intended properties of software that can't be easily inferred, such as the intended relationship between function parameters.

Manuscript received Feb 14th

The FindBugs project started out as an observation, then an experiment, and has snowballed into a widely used tool that has been downloaded more than a half million times all over the world and used by many major companies. The observation was that some Java programs contained blatant mistakes that could be found with fairly trivial analysis techniques. Initial experiments showed that even "production quality" software contained such mistakes and that even experienced developers made such mistakes. FindBugs has grown over time with careful attention to mistakes that actually occur in practice and to the techniques and features needed to effectively incorporate it into production software development.

FindBugs now recognizes more than 300 programming mistakes and dubious coding idioms that can be identified using simple analysis techniques. FindBugs also includes some more sophisticated analysis techniques devised to help effectively identify certain issues, such as dereferencing of null pointers, that require such techniques and occur with enough frequency to warrant their development.

Unlike some other tools designed to provide security guarantees, FindBugs doesn't try to identify all defects of a particular category or provide confidence that software doesn't contain a particular kind of defect. Rather, FindBugs is designed to effectively identify "low hanging fruit" – to cheaply identify defects with a reasonable confidence that the issues found are ones that developers will want to review and remedy.

Many developers use FindBugs on an ad-hoc basis, and a growing number of projects and companies are making it part of their standard build and testing system. Google has incorporated FindBugs into their standard testing and code review process, and has fixed more than 1,000 issues in their internal code base identified by FindBugs.

This article will review the types of issues that are identified by FindBugs, discuss the techniques used to identify new bug patterns and to implement detectors for those bug patterns, discuss experiences with use of FindBugs on Sun's JDK and on Google's Java code base, and provide some preliminary results of surveys and interviews done with FindBugs users.

2 Defects in Real Code

In order to appreciate static analysis for defect detection, and FindBugs in particular, it is useful to be familiar with some sample defects that can be found in real code. All of the examples given in this section come from Sun's JDK 1.6.0 implementation, and are representative of code seen elsewhere.

One of the most unexpectedly common defects is the infinite recursive loop: a function that always returns the result of invoking itself. This bug detector was originally written because some freshman students had trouble understanding how Java constructors worked. But when we ran the detector against build 13 of Sun's JDK 1.6, we found 5 cases, including

```
public String foundType() {
    return this.foundType();
  }
```

This code was intended to be a getter method for the field `foundType`, but because of the extra parenthesis, it always recursively calls itself until the stack overflows. There are a variety of mistakes that lead to infinite recursive loops, but that can all be found with the same simple techniques. Google has found and fixed more than 70 infinite recursive loops in their codebase, and they occur fairly frequently in other code bases we've examined.

Another common bug pattern is when a method is invoked and its return value is ignored, despite the fact that it doesn't make sense to ignore the return value. An example is the statement `s.toLowerCase()` where `s` is a String. Since Strings in Java are immutable, the `toLowerCase()` method returns a new String, and has no effect on the string on which it was invoked. The developer probably intended to write `s = s.toLowerCase()`. Another example is when a developer creates an exception but forgets to throw it:

```
try { ... }
catch (IOException e) {
  new SAXException(....);
  }
```

FindBugs uses an intraprocedural dataflow analysis to identify places where a null pointer could be dereferenced [1], [3]. Although some defects require examining dozens of lines to understand, the majority of the issues that were detected can be understood by examining only a few lines of code. One common case is using the wrong relational or boolean operation, as in a test to see if `(name != null || name.length > 0)`. The `&&` and `||` operators are evaluated using short-circuit evaluation: the right hand side is evaluated only if it needs to be evaluated to determine the value of the expression. In this case, the expression `name.length` will only be evaluated when `name` is null, leading to a null pointer exception. The code would be correct if `&&` had been used rather than `||`. FindBugs also identifies situations where a value is checked for null in some places and unconditionally dereferenced in others. For example, in the following code, the variable `g` is checked to see if it is null, but if it is null the next statement will always deference it, resulting in a null pointer exception:

```
if (g != null)
  paintScrollBars(g,colors);
g.dispose();
```

FindBugs also performs an intraprocedural type analysis that takes into account information from instance of tests, and finds errors such as checked casts that are guaranteed to throw a class cast exception, and places where two objects that are guaranteed to be of unrelated types are compared for equality (e.g., where a StringBuffer is compared to a String or the bug shown in Figure 1).

There are many other bug patterns, some covering obscure aspects of the Java APIs and languages. A particular pattern might only find one issue in several million lines of code, but collectively they find a significant number of issues. Examples include checking if a double value is equal to `Double.NaN` (nothing is equal to `Double.NaN`, not even `Double.NaN`) or performing a bit shift of a 32 bit `int` value by a constant value greater than 31.

2.1 Defects Not Found By FindBugs

FindBugs does not look for or report a number of potential defects that are reported by more powerful tools [7], [5], [6]. This is motivated by two desires: to keep the analysis relatively simple, and to avoid generating too many warnings that do not correspond to true defects.

One such case is finding null pointer dereferences that occur only if a particular path through the program is executed. For example of such an issue was reported [8] by Reasoning in Apache Tomcat 4.1.24. The tool warns that if the body of the first `if` statement is not executed, but the body of the second `if` statement is executed, then a null pointer exception will occur:

```
HttpServletResponse hres = null;
if (sres instanceof HttpServletResponse)
  hres = (HttpServletResponse) sres;

// Check to see if available
if (!(...).getAvailable()) {
  hres.sendError(...)
```

The problem is that the analysis does not know if that path is feasible. Perhaps it is the case that the condition in the second statement can only be true if the condition in the first statement is true. In some cases, the conditions may be closely related and some simple theorem proving may be able to show whether the path is feasible or infeasible. But showing that a particular path is feasible can be much harder, and in general is undecidable. Rather than worry about whether particular paths are feasible, FindBugs looks for branches or statements that if executed, guaranteed that a null pointer exception will occur. We have found that almost all of the null pointer issues we report are either real bugs, or inconsistent code with branches or statements that can't be executed and that wouldn't pass a code review if the inconsistency was noticed.

We have also not pursued checks for array indices being out of bounds. Detecting these errors requires tracking relations between various variables (e.g., is `i` less than the length of `a`), and can become arbitrarily complicated. It is possible that some simply techniques could accurately report some obvious bugs, but we've

not yet pursued that.

3 Nuts and Bolts of FindBugs

FindBugs has a plugin architecture, in which detectors can be defined, each of which may report several different bug patterns. Rather than use a pattern language for describing bugs (as done in PMD [9] and Metal [10]), FindBugs detectors are simply written in Java, using a variety of techniques. Many simple detectors use a visitor pattern over the classfiles and/or the method bytecodes. Detectors have access to information about types, constant values and special flags, as well as values stored on the stack or in local variables. Detectors can also traverse the control flow graph, using the results of data flow analysis such as type information, constant values and nullness. The data flow algorithms all generally use information from conditional tests, so that information from `instanceof` tests and null tests are incorporated into the analysis results.

FindBugs does not perform interprocedural context sensitive analysis. However, many detectors make use of global information such as subtype relationships and which fields are accessed across the entire application. A few detectors use interprocedural summary information, such as which method parameters are always dereferenced.

Each bug pattern is grouped into a category (e.g., correctness, bad practice, performance and internationalization), and each report of a bug pattern is assigned a priority of high, medium or low. The priorities are determined by heuristics unique to each detector/pattern, and are not necessarily comparable across bug patterns. In normal operation, FindBugs does not report low priority warnings.

Perhaps the most important aspect of FindBugs is how new bug detectors are developed: by starting with real bugs, and developing the simplest possible technique that effectively finds those bugs. This approach often allows us to go from finding a particular instance of a bug to implementing a detector that can effectively find it in a matter of hours. Many bugs are really very simple; one of the bug patterns most recently added to FindBugs is when an `int` value is cast to a `char` and the result is checked to see if it is `-1`. Since the `char` type in Java is unsigned, this check will never be true. This bug detector was inspired by a post on `http://worsethanfailure.com/`, and within less than an hour this project had implemented a detector that found 11 such errors in Eclipse 3.3M6.

FindBugs can be run from the command line, using Ant or Maven, within Eclipse or NetBeans, or in a stand alone GUI (Figure 1). The analysis results can be saved in XML, which can then be further filtered, transformed, or imported into a database. FindBugs supports two different mechanisms that enable users and tools to identify corresponding warnings from different analysis runs even if line numbers and other program artifacts have changed [2]. This allows tools to determine which issues are new, and to keep track of audits and human reviews of an issue.

4 Experiences with and Usage of FindBugs

We previously reported [11] on an evaluation of the issues found by FindBugs in Sun's JDK 1.6.0 implementation. To briefly summarize, we looked at each FindBugs medium or high priority correctness warning that was present in one build and not reported in the next build, but the class containing the warning was still present. Of a total of 53 such warning removals, 37 were due to a small targeted program change that seemed to be narrowly focused on remedying the issue described by the warning. Five were program changes that changed the code such that FindBugs no longer reported the issue, but aspects of the underlying issue were not completely addressed. The remaining 11 warnings disappeared due to substantial changes or refactorings that had a larger scope than the removal of the one defect.

Our previous work also included a manual evaluation of all of the medium and high priority correctness warnings in build 105 of (the official release). We classified the 379 medium and high priority correctness warnings as follows:

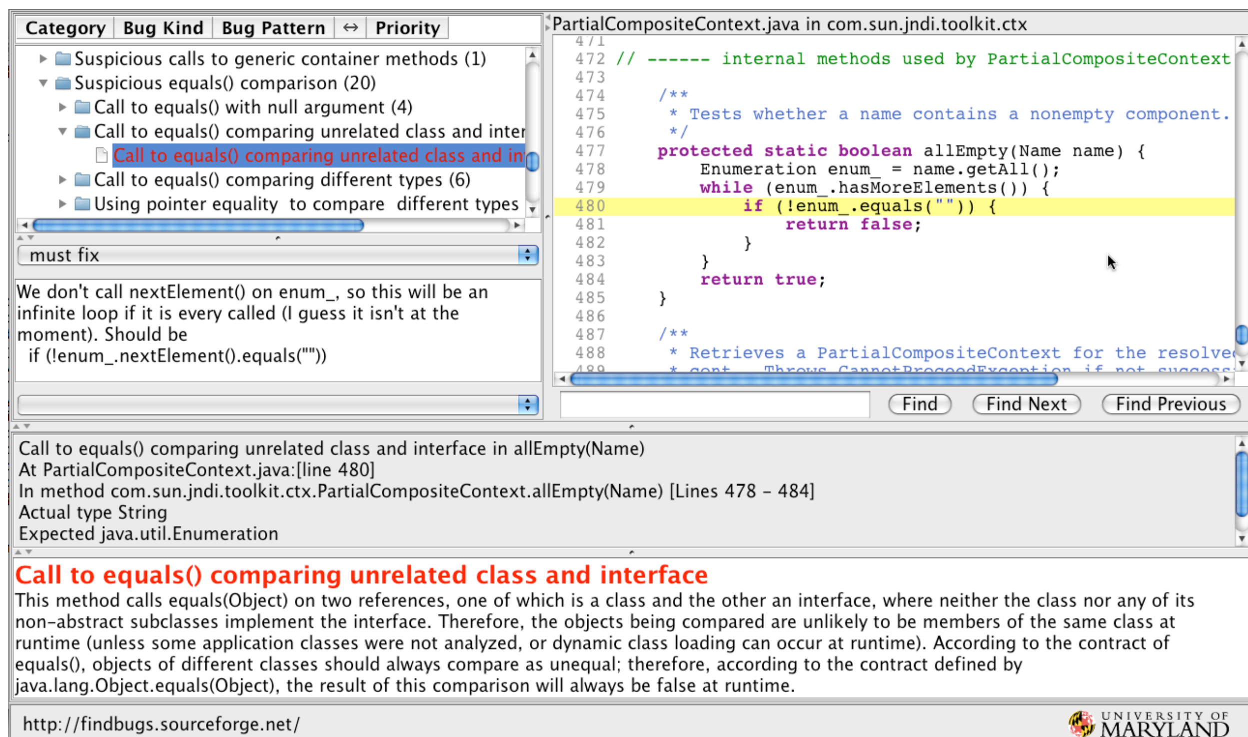

Fig. 1. Screenshot of the FindBugs Swing GUI, reviewing a bug in Sun's JDK

- 5 were due to bad analysis by FindBugs (in one case, due to not understanding that a method call could change a field).
- 160 were in unreachable code or likely to have little or no functional impact.
- 176 seemed to have functional impact.
- 38 seemed have substantial functional impact: the method containing the warning would clearly behave in a way substantially at odds with its intended function.

A detailed breakdown of the classification of the defects associated with each bug pattern are provided in our previous paper [11]. Clearly, any such classification is open to interpretation, and it is likely that other reviewers would produce slightly different classifications. Also, our assessment of the functional impact may differ from the actual end-user perspective. For example, even if a method is clearly broken, the method might never be called and might not be invokable by user code. However, given the localized nature of many of the bug patterns, we have some confidence in the general soundness of our classification.

4.1 Experiences at Google

Google's use of FindBugs has evolved over the last two years in three distinct phases. We used the lessons learned during each phase to plan and develop the next phase.

The first phase involved automating the running of FindBugs over all newly checked in Java source code, and storing the generated warnings. A simple web interface let developers check their project for possible bugs and mark false positives. Our initial database could not track warnings over different versions, and as a result the web interface saw little usage. Developers could not determine which warnings applied to which file versions, or whether warnings were fresh or stale. When a defect was fixed, this event was not reported by our process. Such stale warnings have a greater negative impact on the developer's user experience than a false positive. Successfully injecting FindBugs into Google's development process was not as simple as making all warnings available outside of an engineer's normal workflow.

For the second phase, this project implemented a service model where we (David and John) spent half the time evaluating warnings

and reporting those we decided were significant defects in Google's bug tracking systems. Over the course of six months this project evaluated several thousand FindBugs warnings and filed over 1000 bug reports. At first this effort focused on bug patterns chosen using our own opinions of the different patterns' importance. As we gained experience and feedback from developers, we prioritized the evaluation based on our prior empirical results. We ranked the different patterns using a combination of the observed false positive rate and the observed fix rate for issues we filed as bugs. Thus, we spent more time evaluating the warnings that were more likely to actually get fixed. This ranking scheme carried over into the third phase, as we noticed that our service model would not scale well as Google grew.

It was observed that in many cases, filing a bug report was more effort than simply fixing the code. To better scale the operation, we needed to move the analysis feedback closer to the development workflow. In the third and current phase, we take advantage of Google's code review policy and tools. Before code changes are checked in to Google's source control system, they must first be reviewed by another engineer. Different tools are available to support this review process; one of the more sophisticated is Mondrian, an internal web based review tool [12].

Mondrian allows a reviewer to add inline comments to the code that are visible to other Mondrian users, including the original requester. Engineers discuss the code using these comments, and note completed modifications. For example, a reviewer might request in an inline comment, "Please rename this variable." In response, the developer would make the requested change and reply to the original comment with an inline "Done." We let Mondrian users see FindBugs, and other static analysis, warnings as inline comments from our automated reviewer, BugBot. We provide a false positive suppression mechanism, and allow them to filter the comments displayed by 'confidence,' from highest to lowest. Each user selects the minimum confidence level he or she wishes to see, which suppresses all lower ranked warnings.

This system scales quite well, and we have seen more than 200 users verify or suppress thousands of warnings in the last six months. We still have improvements to make, such as automatically running FindBugs on each development version of a file while it is being reviewed and before it is checked in. The main lesson to take away from this experience is that developers will pay attention to, and fix, FindBugs warnings if they appear as a seamless part of their workflow. It helps that code reviewers can also see the warnings and request fixes as they review the code. Our ranking and false positive suppression mechanisms are crucial to keeping the displayed warnings relevant and valuable, so that users don't start ignoring the more recent, important warnings along with the older, more trivial ones.

4.2 Survey of FindBugs users

Many studies on static analysis tools focus on their correctness (are the warnings they identify real problems), their completeness (do they find all problems in a given category), or their performance in terms of memory and speed. As organizations start to integrate these tools into their software processes, other considerations need to be made about the interactions between these tools and the users or processes. Do these tools slow down the process with unnecessary warnings, or is the value provided by these tools (in terms of problems found) worth the investment in time? What is the best way to integrate these tools into a given process? Should all developers interact with the tools or should quality assurance specialists winnow out less useful warnings?

There are not many rules of thumb about the best ways to use static analysis tools. Instead there are a hodgepodge of methods used by different software teams. Many users do not even have a formal process for finding defects using tools—they only occasionally run the tools and are not consistent in the ways they respond to warnings. In the end users may not derive full value from static analysis tools, and some may discontinue use of these tools because of an incorrectly perceived lack of value.

The FindBugs team 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. At this early stage in our research, it is not clear what the problems are and what questions need to be investigated in more depth. Hence we are conducting some surveys and interviews to get qualitative feedback from FindBugs users. We want to find out who our users are, how they use FindBugs, how they integrate it into their processes, and what their perception of FindBugs' effectiveness is. Beyond surveys and interviews, we hope to spend time observing users in their work environments to capture the nuances of their interactions with this tool.

The following sections detail some observations from the surveys and interviews.

4.2.1 On FindBugs' utility and impact

The central challenge for tool creators is to identify warnings that users are concerned with. Tools like FindBugs assess each warning based on its *severity* (how serious is the problem in general) and the tool's *confidence* in the analysis. Though, as one user pointed out, users are really interested in *risk*—high risk warnings which are those that may actually cause the code to fail and expose the organization. A risk-based assessment will be different from organization to organization and from project to project. Since FindBugs does not have access to an all-knowing context-specific oracle, it cannot perfectly serve every user. Our survey and feedback from users show that FindBugs is finding many problems users are interested in, and users are willing to invest the time needed to review these warnings.

Recall that FindBugs prioritizes its warnings into high, medium and low priority levels. Our survey indicates that most users review at least the high priority warnings in all categories (Table 1). This is the expected outcome, since high priority warnings are intended to be the sorts of problems any user would want to fix. 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 or may not be relevant depending on the user's context. Users need to tune FindBugs to filter out detectors they don't care about at lower priority levels.

TABLE 1
Proportion of users that review at least high priority warnings for each category (out of 252)

Category	Proportion
Bad Practice	96% of users
Performance	96%
Correctness	95%
Multithreaded Correctness	93%
Malicious Code Vulnerability	86%
Dodgy	86%
Internationalization	57%

Many users run FindBugs out of the box without any tuning—55% of our survey respondents indicated that they do not do any filtering of bug patterns. One user suggested that FindBugs provide a number of preset configurations that selectively filter out detectors depending on the user's context. Users working on web applications have different priorities from those working on desktop applications; organizations want to be warned about debugging facilities such as references to JUnit when the code is about to be released but not while it is under development [6]. More research is needed to determine how to cluster users into different contexts, and which detectors are most relevant for each context.

The willingness of users to review warnings and fix issues also depends on some characteristics of their project and organization such as the time investment they are willing to put into each review and their tolerance for false positives. Users analyzing older, more stable code bases are less likely to change code in response to a warning than users analyzing recently written code. We suspect that FindBugs warnings have relatively low review times and are easy to fix, and that there are few false positives for those detectors that users care about. We plan to do more studies to examine this more closely.

Some users are wary of "tuning code" to

FindBugs by modifying the code to remove even low priority warnings or adding annotations. Some other users willingly make these modifications, even if they are convinced that the code in question cannot actually behave incorrectly. Of course, this is easier to do if the code is new. Some users do this to increase their confidence in the quality of their code (one user commented: “the effort to reformulate source code to avoid FindBugs warnings is time well spent”). Some users who are unaware of FindBugs’ warning suppression facilities fix all warnings to ensure that future warnings are not drowned out by older unresolved issues. Particularly on issues of style, this kind of tuning may lead to conflicts between different tools that users have to resolve. An example is the use of annotations to aid null pointer dereferencing detectors. FindBugs provides a set of annotations, but so do some other tools. To prevent a conflict for users, some vendors and users have come together to propose JSR 305, a Java Specification Request that standardizes annotations used to indicate nullness (among other things) [13], [14].

Another observation is that users may choose to ignore some warnings because they have taken steps to mediate the problems using other facilities. For example, a user indicated that he ignored warnings associated with web security because he relied heavily on input validation and white-listing to control program inputs. Input validation is a natural way to fight SQL injection, cross-site scripting and other security problems. Unfortunately static analysis tools are sometimes unaware of the input validation processes, and may report warnings even if effective input validation schemes are in place.

4.2.2 On organizational policies

Many survey participants do not have formal policies for using FindBugs (Table 2), and use it in an ad hoc way (i.e. a developer occasionally runs it manually). Sometimes there are weeks between two run of FindBugs, as users are focused on adding features and fighting the problems they are aware of. Indeed it appears that many users had not considered that formal policies may make their usage of tools more effective until they took the survey. Most respondents indicated that their organizations do not enforce any limits on how long warnings can go unreviewed. This makes it likely that many reviews may take place closer to the release date, when the pressure means that the emphasis is more on suppressing warnings than fixing code.

TABLE 2
Formal policies for using FindBugs

Our developers only occasionally run FindBugs manually	60% of users
No policy on how soon each FindBugs issue must be human reviewed	81%
Running FindBugs is NOT required by our process, or by management	76%
FindBugs warnings are NOT inserted into a separate bug tracking database	83%
No policy on how to handle warnings designated “Not A Bug”	55%

A few organizations do have policies ranging from requiring a FindBugs run as part of a quality assurance or release process, to breaking the central build or disallowing a code check-in if there are any unresolved FindBugs warnings. Other policies include automatically inserting warnings into a bug tracker, having one or two people that maintain FindBugs and review warnings, requiring that warnings are human reviewed within a given time limit or warning count threshold, integrating FindBugs into a code review process, running FindBugs automatically over night and emailing problems to developers, and using a continuous build server to display currently active warnings.

Many teams realize the need for a way to suppress warnings that are not bugs or that are low impact issues (Table 3). FindBugs filters were the most common method, followed by source level suppression using annotations (such as @SuppressWarnings). As mentioned above, some users change the code anyway to make the warning go away. Others use FindBugs filters, and some have internal scripts or processes for suppression. Source level suppression (by inserting line level, method level

TABLE 3
Handling issues designated "Not A Bug"

Filter out using FindBugs filters	25% of users
Suppress using @SuppressWarnings	17%
Close in a bug tracker or database	5%
No policy	55%

or class level annotations) is also attractive to some users because the suppression information is readily available to anyone who works on that code in the future. Source level suppression may be more effective if the annotations are automatically inserted in response to action by a reviewer.

In many cases, the person who writes the code is responsible for reviewing the warning, deciding if it is relevant, and resolving the issue. Many organizations place the responsibility for deciding if a warning is a bug in the hands of a single individual. (Eleven percent of users said a team does the review, and fourteen percent indicated that a reviewer can make independent decisions only for trivial cases.) This raises questions about whether two different individuals will see warnings the same way. We plan to study this effect in FindBugs.

5 Conclusion

It has become fairly clear that static analysis tools can find important defects in software. This is particularly important in the realm of security defects (such as buffer overflows and SQL injections), since the cost incurred by deploying such a defect can easily run into the millions. Many of the coding defects found by FindBugs, such as potentially throwing a null pointer exception, are less severe in the sense that fewer of them are likely to have multi-million dollar costs. Thus, it is particularly important for this research to look at the cost effectiveness of using static analysis tools.

Software developers are busy, with many different tasks and ways of reaching the goal of swift development of correct and reliable software. We need to develop procedures and best practices that make using static analysis tools more effective than alternative uses of developer time, such as spending additional time performing manual code review or writing test cases.

We believe that we have achieved that goal with FindBugs, although we have not yet measured or demonstrated it. Through user surveys, we found that actual use of FindBugs is more diverse than we had expected, and that many of the things we believe to be best practices have yet to be widely adopted. For example, very few users of FindBugs use an automatic build system where new issues are automatically identified and flagged. We are continuing studies with users and development organizations, as it seems clear to us that development, measurement, validation and adoption of best practices for static analysis tools is key to allowing these tools to be used effectively.

Acknowledgments

The authors would like to thank Fortify Software for sponsoring the FindBugs project, and thank Google and Sun Microsystems for additional support.

References

[1] 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*. New York, NY, USA: ACM Press, 2005, pp. 13–19.

[2] 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*. New York, NY, USA: ACM Press, 2006, pp. 133–136.

[3] 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*. New York, NY, USA: ACM, 2007, pp. 9–14.

[4] I. F. Darwin, *Checking C Programs with Lint*. O'Reilly, 1988.

[5] S. Hallem, D. Park, and D. Engler, "Uprooting software defects at the source," *Queue*, vol. 1, no. 8, pp. 64–71, 2003.

[6] B. Chess and J. West, *Secure Programming with Static Analysis*, 1st ed. Addison-Wesley Professional, Jul. 2007.

[7] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Softw. Pract. Exper.*, vol. 30, no. 7, pp. 775–802, 2000.

[8] Reasoning, Inc., "Reasoning inspection service defect data report for Tomcat, version 4.1.24," January 2003, http://www.reasoning.com/pdf/Tomcat_Defect_Report.pdf.

[9] T. Copeland, *PMD Applied*. Centennial Books, November 2005.

[10] B. Chelf, D. Engler, and S. Hallem, "How to write system-specific, static checkers in metal," in *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM Press, 2002, pp. 51–60.

[11] 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*. New York, NY, USA: ACM, 2007, pp. 1–8.

[12] "Mondrian: Code review on the web," Dec. 2006. [Online]. Available: http://video.google.com/videoplay?docid=-8502904076440714866

[13] D. Hovemeyer and W. Pugh, "Status report on jsr-305: annotations for software defect detection." Montreal, Quebec, Canada: ACM, 2007, pp. 799–800.

[14] "Jsr 305: Annotations for software defect detection." [Online]. Available: http://jcp.org/en/jsr/detail?id=305

Nathaniel Ayewah has a B.S. in Computer Engineering and an M.S. in Computer Science from Southern Methodist University. His research interests include understanding the way users interact with software tools and using information visualization to support creativity. He has a diverse research background in which he has explored testing concurrent software, visualizing proofs, using machine learning for speech analysis, visual temporal queries, web-based data collection, noise reduction in hearing aids and data mining. Nathaniel is currently a Ph.D. student of Computer Science at the University of Maryland, College Park.

David Hovemeyer developed FindBugs as part of his Ph.D. research the University of Maryland, College Park, in conjunction with his thesis advisor William Pugh. He is currently an Assistant Professor of Computer Science at York College of Pennsylvania, where he teaches introductory Computer Science courses and upper-level courses in programming languages, software engineering, and operating systems.

Previously, David was a Visiting Assistant Professor of Computer Science at Vassar College (2005–2006), and a Software Engineer at Cigital (1996–1998). He earned a B.A. in Computer Science from Earlham College in 1994.

David Morgenthaler received a B.A. in Geography from the University of California, Berkeley, an M.S. in Mathematics from California State University, Hayward, and a Ph.D. in Computer Science from the University of California, San Diego. He has taught Computer Science at the Hong Kong University of Science and Technology, and worked at several Silicon Valley startups. David is currently a software engineer at Google.

John Penix John Penix is a Senior Software Engineer in Google's Test Engineering organization, where he tries to detect more defects than he injects. He is currently working on the tools that are used to gather, prioritize and display static analysis warnings.

From 1998 to 2006, John was a Computer Scientist in the Intelligent Systems Division of NASA's Ames Research Center where he contributed to research projects in the areas of software model checking, deductive program synthesis and collaborative software engineering. John currently serves on the Steering Committee of the IEEE/ACM International Conference on Automated Software Engineering. John received a Ph.D. in Computer Engineering from the University of Cincinnati.

William Pugh received a B.S. in Computer Science from Syracuse University and received a Ph.D. in Computer Science (with a minor in Acting) from Cornell University. He is currently a professor at the University of Maryland, College Park. William Pugh is a Packard Fellow, and invented Skip Lists, a randomized data structure that is widely taught in undergraduate data structure courses.

He has also made research contributions in the fields of incremental computation, implementation of functional and object-oriented languages, the use of partial evaluation for hard real-time systems, in techniques for analyzing and transforming scientific codes for execution on supercomputers, and in a number of issues related to the Java programming language, including the development of JSR 133 - Java Memory Model and Thread Specification Revision.

Prof. Pugh consulted for Google in 2000 - 2003 on research that resulted in US Patent 665 8423, on detecting duplicate and near-duplicate files.

Prof Pugh's current research focus is on developing tools to improve software productivity, reliability and education. Current research projects include FindBugs, a static analysis tool for Java, and Marmoset, an innovative framework for improving the learning and feedback cycle for student programming projects