Using Static Analysis to Find Bugs

Nathaniel Ayewah and William Pugh, *University of Maryland*

David Hovemeyer, *York College of Pennsylvania*

J. David Morgenthaler and John Penix, *Google*

FindBugs, an open source static-analysis tool for Java, evaluates what kinds of defects can be effectively detected with relatively simple techniques.

Software quality is important, but often imperfect in practice. We can use many techniques to try to improve quality, including testing, code review, and formal specification. Static-analysis tools evaluate software in the abstract, without running the software or considering a specific input. Rather than trying to prove that the code fulfills its specification, such tools look for violations of reasonable or recommended programming practice. Thus, they look for places in which code might dereference a null pointer or overflow an array. Static-analysis tools might also flag an issue such as a comparison that can't possibly be true. Although the comparison won't cause a failure or exception, its existence suggests 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[1] is generally considered the first widely used static-analysis tool for defect detection, although by today's standards it's rather limited. Researchers have done significant 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. A vibrant commercial industry has developed around advanced (and expensive) static-analysis tools,[2,3] and several companies have their own proprietary in-house tools, such as Microsoft's PREfix.[4] Many commercial tools are sophisticated, using deep analysis techniques. Some can use or depend on annotations that describe invariants and other intended software properties that tools can't easily infer, such as the intended relationship between function parameters.

FindBugs is an example of a static-analysis tool that looks for coding defects.[5–7] The FindBugs project began as an observation, developed into an experiment, and snowballed into a widely used tool with more than half a million downloads worldwide. The observation that started it all was that some Java programs contained blatant mistakes that were detectable with fairly trivial analysis techniques. Initial experiments showed that even "production quality" software contained such mistakes and that even experienced developers made them. FindBugs has grown, paying careful attention to mistakes that occur in practice and to the techniques and features needed to effectively incorporate it into production software development.

Here, we review the types of issues FindBugs identifies, discuss the techniques it uses, and look at some experiences using FindBugs on Sun's Java Development Kit (JDK) and Google's Java code base.

0740-7459/08/$25.00 © 2008 IEEE

FindBugs in practice

In its current form, FindBugs recognizes more than 300 programming mistakes and dubious coding idioms that it can identify using simple analysis techniques. FindBugs also uses more sophisticated analysis techniques, devised to help effectively identify certain issues—such as dereferencing of null pointers—that occur frequently enough to warrant their development. Unlike some other tools designed to provide security guarantees, FindBugs doesn't try to identify all defects in a particular category or prove that software doesn't contain a particular defect. Rather, it's designed to effectively identify low-hanging fruit—to cheaply detect defects we believe developers will want to review and remedy.

Many developers use FindBugs ad hoc, and a growing number of projects and companies are integrating it into their standard build and testing systems. Google has incorporated FindBugs into its standard testing and code-review process and has fixed more than 1,000 issues in its internal code base that FindBugs has identified.

Defects in real code

To appreciate static analysis for defect detection in general, and FindBugs in particular, it helps to be familiar with some sample defects found in real code. Let's look at some examples from Sun's JDK 1.6.0 implementation, which also are representative of code seen elsewhere.

One unexpectedly common defect is the infinite recursive loop—that is, a function that always returns the result of invoking itself. We originally extended FindBugs to look for this defect because some freshman at the University of Maryland had trouble understanding how Java constructors worked. When we ran it against build 13 of Sun's JDK 1.6, we found five infinite recursive loops, including

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

This code should have been a getter method for the field foundType, but the extra parenthesis means it always recursively calls itself until the stack overflows. Various mistakes lead to infinite recursive loops, but the same simple techniques can detect them all. Google has found and fixed more than 70 infinite recursive loops in their code base, and they occur surprisingly frequently in other code bases we've examined.

Another common bug pattern is when software invokes a method but ignores its return value, despite the fact that doing so makes no sense. An example is the statement s.toLowerCase(), where s is a String. Because Strings in Java are immutable, the toLowerCase() method has no effect on the String it's invoked on, but rather returns a new String. 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(....);
}
```

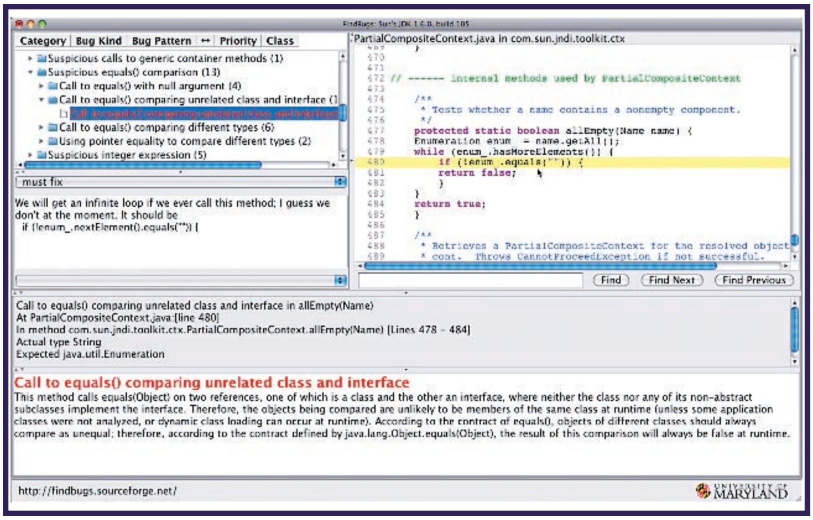
FindBugs uses an intraprocedural dataflow analysis to identify places in which the code could dereference a null pointer.[5,7] Although developers might need to examine dozens of lines to understand some defects reported by FindBugs, most can be understood by examining only a few lines of code. One common case is using the wrong relational or Boolean operation, as in a test to see whether (name != null || name.length > 0). Java evaluates the && and || operators using short-circuit evaluation: the right-hand side is evaluated only if needed in order to determine the expression's value. In this case, Java will evaluate the expression name.length only when name is null, leading to a null pointer exception. The code would be correct if it had used && rather than ||. FindBugs also identifies situations in which the code checks a value for null in some places and unconditionally dereferences it in others. The following code, for example, checks the variable g to see if it's 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 instanceof tests and finds errors such as checked casts that always throw a class cast exception. It also finds places in which two objects guaranteed to be of unrelated types are compared for equality (for example, where a StringBuffer is compared to a String, or the bug Figure 1 shows).

Many other bug patterns exist, some covering obscure aspects of the Java APIs and languages. A particular pattern might find only one issue in several million lines of code, but collectively these find a significant number of issues. Examples include checking whether a double value is equal to Double.NaN (nothing is equal to Double.NaN, not even Double.NaN)

FindBugs doesn't try to identify all defects in a particular category.



FindBugs: Sun's JDK 1.6.0, build 105
Category | Bug Kind | Bug Pattern ↔ | Priority | Class

- Suspicious calls to generic container methods (1)
- Suspicious equals() comparison (13)
  - Call to equals() with null argument (4)
  - Call to equals() comparing unrelated class and interface (1
    - Call to equals() comparing unrelated class and interface
  - Call to equals() comparing different types (6)
  - Using pointer equality to compare different types (2)
- Suspicious integer expression (5)

must fix

We will get an infinite loop if we ever call this method; I guess we don't at the moment. It should be
  if (!enum_.nextElement().equals("")) {

PartialCompositeContext.java in com.sun.jndi.toolkit.ctx

```
    }


// ------ Internal methods used by PartialCompositeContext

    /**
     * Tests whether a name contains a nonempty component.
     */
    protected static boolean allEmpty(Name name) {
    Enumeration enum_ = name.getAll();
    while (enum_.hasMoreElements()) {
        if (!enum_.equals("")) {
        return false;
        }
    }
    return true;
    }

    /**
     * Retrieves a PartialCompositeContext for the resolved object
     * cont.  Throws CannotProceedException if not successful.
```

Call to equals() comparing unrelated class and interface in allEmpty(Name)
At PartialCompositeContext.java:[line 480]
In method com.sun.jndi.toolkit.ctx.PartialCompositeContext.allEmpty(Name) [Lines 478 - 484]
Actual type String
Expected java.util.Enumeration

Call to equals() comparing unrelated class and interface

This method calls equals(Object) on two references, one of which is a class and the other an interface, where neither the class nor any of its non-abstract subclasses implement the interface. Therefore, the objects being compared are unlikely to be members of the same class at runtime (unless some application classes were not analyzed, or dynamic class loading can occur at runtime). According to the contract of equals(), objects of different classes should always compare as unequal; therefore, according to the contract defined by java.lang.Object.equals(Object), the result of this comparison will always be false at runtime.

http://findbugs.sourceforge.net/

Figure 1. The FindBugs Swing GUI. The interface shows FindBugs reviewing a bug in Sun's Java Development Kit.

or performing a bit shift of a 32-bit int value by a constant value greater than 31.

What FindBugs doesn't find

FindBugs doesn't look for or report numerous potential defects that more powerful tools report.[2–4] We designed it this way for two reasons: to keep the analysis relatively simple and to avoid generating too many warnings that don't correspond to true defects.

One such case is finding null pointer dereferences that occur only if a particular path through the program is executed. Reasoning reported such an issue in Apache Tomcat 4.1.24.[8] Reasoning warns that if the body of the first if statement isn't 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 doesn't know whether that path is feasible. Perhaps the condition in the second statement can be true only if the condition in the first statement is true. In some cases, the conditions might be closely related and some simple theorem proving can show whether the path is feasible or infeasible. But showing that a particular path is feasible can be much harder, and is in general undecidable.

Rather than worry about whether particular paths are feasible, FindBugs looks for branches or statements that, if executed, guarantee that a null pointer exception will occur. We've found that almost all null pointer issues we report are either real bugs or inconsistent code with branches or statements that can't be executed. Code that is merely inconsistent might not be changed if it's already used in production, but generally would be considered unacceptable in new code if found during code review.

We also haven't pursued checks for array indices that are out of bounds. Detecting such errors requires tracking relations between various variables (for instance, is i less than the length of a), and can become arbitrarily complicated. Some simple techniques might accurately report some obvious bugs, but we haven't yet investigated this.

FindBugs nuts and bolts

FindBugs has a plug-in architecture in which detectors can be defined, each of which might report several different bug patterns. Rather than use a pattern language to describe bugs (as PMD[9] and Metal[10] do), FindBugs detectors are simply written in Java using various techniques. Many simple detectors use a visitor pattern over the class files or method byte codes. Detectors can access 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 the analysis results incorporate information from instanceof and null tests.

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

FindBugs groups each bug pattern into a category (such as correctness, bad practice, performance, and internationalization) and assigns each bug pattern report either high, medium, or low priority. FindBugs determines priorities via heuristics unique to each detector or pattern that aren't necessarily comparable across bug patterns. In normal operation, FindBugs doesn't report low-priority warnings.

The most important aspect of the FindBugs project is perhaps how we develop new bug detectors: we start with real bugs and develop the simplest possible technique that effectively finds such bugs. This approach often lets us go from finding a particular instance of a bug to implementing a detector that can effectively find instances of it within hours. Many bugs are quite simple—one bug pattern most recently added to FindBugs occurs when the code casts an int value to a char and checks the result to see whether it's –1. Because the char type in Java is unsigned, this check will never be true. A post on http://worsethanfailure.com inspired this bug detector, and within less than an hour, we had implemented a detector that found 11 such errors in Eclipse 3.3M6.

We can run FindBugs from the command line, using Ant or Maven, within Eclipse or NetBeans, or in a stand-alone GUI (see Figure 1). We can save the analysis results in XML, which we can then further filter, transform, or import into a database. FindBugs supports two mechanisms that let users and tools identify corresponding warnings from different analysis runs, even if line numbers and other program artifacts have changed.[6] This lets tools determine which issues are new and track audits and human reviews.

FindBugs experiences

We've evaluated the issues FindBugs uncovered in Sun's JDK 1.6.0 implementation elsewhere.[11] To briefly summarize, we looked at each FindBugs medium- or high-priority correctness warning that was in one build and not reported in the next, even though the class containing the warning was still present. Out of 53 such warning removals, 37 were due to a small targeted program change that seemed to narrowly focus on remedying the issue the warning described. Five were program changes that changed the code such that FindBugs no longer reported the issue, even though the change didn't completely address aspects of the underlying issue. The remaining 11 warnings disappeared owing to substantial changes or refactorings that had a larger scope than just removing the one defect.

In previous research, we also manually evaluated all the medium- and high-priority correctness warnings in build 105 (the official release of Java 1.6.0). We classified the 379 medium- and high-priority correctness warnings as follows:

- 5 occurred owing to bad analysis on FindBugs' part (in one case, it didn't understand 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; and
- 38 seemed to have substantial functional impact—that is, the method containing the warning would clearly behave in a way substantially at odds with its intended function.

A detailed breakdown of the defect classification associated with each bug pattern appears in our previous paper.[11] Clearly, any such classification is open to interpretation, and other reviewers would likely produce slightly different classifications. Also, our assessment of functional impact might differ from the actual end-user perspective. For example, even if a method is clearly broken, it might never be called or be invokable by user code. However, given many bug patterns' localized nature, we have some confidence in our classifications' general soundness.

Experiences at Google

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

The first phase involved automating FindBugs to run over all newly checked-in Java source code and store any generated warnings. A simple Web interface let developers check projects for possible bugs and mark false positives. Our initial database couldn't track warnings over different versions, so the Web interface saw little use. Developers couldn't determine which warnings applied to which file versions or whether the warnings were fresh or stale. When a defect was fixed, this event wasn't 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 required more than just making all warnings available outside an engineer's normal workflow.

Stale warnings have a greater negative impact on the developer's user experience than a false positive.

In our project's second phase, we implemented a service model in which two of the authors (David Morgenthaler and John Penix) spent half the time evaluating warnings and reporting those we decided were significant defects in Google's bug-tracking systems. Over the next six months, we evaluated several thousand FindBugs warnings and filed more than 1,000 bug reports. At first, this effort focused on bug patterns we chose on the basis of our own opinions about their importance. As we gained experience and developer feedback, we prioritized our evaluation on the basis of our prior empirical results. We ranked the different patterns using both the observed false-positive rate and the observed fix rate for issues we filed as bugs. Thus, we spent more time evaluating warnings that developers were more likely to fix. This ranking scheme carried over into the third phase, as we noticed that our service model wouldn't scale well as Google grew.

We observed that, in many cases, filing a bug

Table 1

Users that review at least high-priority warnings for each category (out of 252)

Bug category reviewed	Percentage of users
Bad practice	96
Performance	96
Correctness	95
Multithreaded correctness	93
Malicious code vulnerability	86
Dodgy	86
Internationalization	57

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 exploit Google's code-review policy and tools. Before a developer checks code changes into Google's source-control system, another developer must first review them. Different tools help support this process, including Mondrian, a sophisticated, internal Web-based review tool.[12]

Mondrian lets reviewers add inline comments to 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 let developers filter the comments displayed by "confidence" from highest to lowest. Users select the minimum confidence level they wish to see, which suppresses all lower-ranked warnings.

This system scales quite well, and we've seen more than 200 users verify or suppress thousands of warnings in the past six months. We must still make some improvements, such as automatically running FindBugs on each development version of a file while developers are reviewing it and before they check it in. The main lesson we learned from this experience is that developers will pay attention to, and fix, FindBugs warnings if they appear seamlessly within the 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.

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 begin integrating these tools into their software processes, we must consider other aspects of the interactions between these tools and users or processes. Do these tools slow down the process with unnecessary warnings, or is the value they provide (in terms of problems found) worth the investment in time? What's 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?

Few rules of thumb exist about the best ways to use static-analysis tools. Rather different software teams use a hodgepodge of methods. Many users don't even have a formal process for finding defects using tools—they run the tools only occasionally and aren't consistent in how they respond to warnings. In the end, users might not derive full value from static-analysis tools, and some might discontinue their use, incorrectly perceiving that they lack value.

The FindBugs team has started a project that aims to identify and evaluate tool features, validate or invalidate assumptions tool vendors hold, and guide individuals and teams wanting to use static-analysis tools effectively. At this early stage, it isn't clear what the problems are and what questions we should investigate in more depth. So, we're conducting some surveys and interviews to get qualitative feedback from FindBugs users. We want to determine who our users are, how they use FindBugs, how they integrate it into their processes, and what their perception of its effectiveness is. Beyond surveys and interviews, we hope to spend time observing users in their work environments to capture the nuances in their interactions with this tool.

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

On FindBugs' utility and impact. The central challenge for tool creators is to identify warnings that users are concerned with. Tools such as FindBugs assess each warning on the basis of its severity (how serious the problem is in general) and the tool's confidence

Table 2
Formal policies for using FindBugs

Policy for using FindBugs	Percentage of users
Our developers only occasionally run FindBugs manually	60
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
Internationalization	57

in the analysis. As one user pointed out, however, users are really interested in risk—especially high-risk warnings, or those that might 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. Because FindBugs doesn’t have access to an all-knowing, context-specific oracle, it can’t perfectly serve every user. Our survey and user feedback show that FindBugs is detecting many problems users are interested in, and these users are willing to invest the time needed to review the warnings.

Recall that FindBugs prioritizes its warnings into high, medium, and low. Our survey indicates that most users review at least the high-priority warnings in all categories (see Table 1). This is the expected outcome because such warnings are intended to be the sorts of problems any user would want to fix. We were surprised by the number of users that also review lower-priority warnings (although the review categories vary from user to user). This indicates that although high-priority warnings are relevant to most users, lower-priority ones are relevant depending on the user’s context. Users can tune FindBugs to include low-priority warnings in the categories in which they’re particularly interested.

Many users run FindBugs out of the box without any tuning—55 percent of our survey respondents indicated that they don’t filter any bug patterns. One user suggested that FindBugs provide 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 warnings about debugging facilities, such as references to JUnit when the code is near release but not while it’s under development.[3] We must conduct more research to determine how to cluster users into different contexts and which detectors are most relevant for each context.

Users’ willingness to review warnings and fix issues also depends on project characteristics and organization, such as the time investment they’re 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 few false positives exist 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’re convinced that the code in question can’t actually behave incorrectly. Of course, this is easier to do if the code is new. Some users do this to increase their confidence in their code’s quality (one user commented that “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 aren’t drowned out by older, unresolved issues. Particularly on style issues, such tuning can lead to conflicts between different tools that users must then resolve. One 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 Java Specification Request (JSR) 305, which standardizes annotations used to indicate nullness (among other things).[13,14]

Another observation is that users might choose to ignore some warnings because they’ve 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 might report warnings even if effective input validation schemes are in place.

On organizational policies. Many survey participants don’t have formal policies for using FindBugs (see Table 2) and use it in an ad hoc way (that is,

Table 3

Handling issues designated "Not a bug"

Mechanism for suppressing issues	Percent of users
Filter out using FindBugs filters	25
Suppress using @SuppressWarnings	17
Close in a bug tracker or database	5
No policy	55

a developer occasionally runs it manually). Sometimes weeks go by between two runs of FindBugs because users are focused on adding features and fighting problems of which they're aware. Indeed, it appears that many users hadn't considered that formal policies might make their tool use more effective until they took the survey. Most respondents indicated that their organizations don't enforce any limits on how long warnings can go unreviewed. This makes it likely that many reviews take place closer to the release date, when the pressure means that the emphasis is more on suppressing warnings than fixing code.

A few organizations 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 any unresolved FindBugs warnings exist. Other policies include automatically inserting warnings into a bug tracker, having one or two people who maintain FindBugs and review warnings, requiring that warnings are human reviewed within a given time limit or warning-count threshold, integrating FindBugs into code review, running FindBugs automatically overnight and emailing problems to developers, and using a continuous-build server to display active warnings.

Many teams realize the need for a way to suppress warnings that aren't bugs or that are low-impact issues (see Table 3). FindBugs filters were the most common method, followed by source-level suppression using annotations (such as @SuppressWarnings). As we mentioned, 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 (inserting line-level, method-level, or class-level annotations) also attracts some users because the suppression information is readily available to anyone who works on that code in the future. Source-level suppression might be more effective if the annotations are automatically inserted in response to an action by a reviewer.

In many cases, the person who writes the code is responsible for reviewing the warning, deciding whether it's relevant, and resolving the issue. Many organizations place the responsibility for deciding whether a warning is a bug into a single individual's hands. (Eleven percent of users said a team does the review, and 14 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.

It's become fairly clear that static-analysis tools can find important defects in software. This is particularly important when it comes to security defects (such as buffer overflows and SQL injections) because the cost incurred by deploying such a defect can easily run into the millions of dollars. Many coding defects that FindBugs discovers, such as potentially throwing a null pointer exception, are less severe in the sense that fewer of them will likely have multimillion dollar costs. So, it's particularly important for our research to look at static-analysis tools' cost effectiveness.

Software developers are busy, with many different tasks and ways to achieve 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've achieved that goal with FindBugs, although we haven't yet measured or demonstrated it. Through user surveys, we found that actual FindBugs use is more diverse than we'd expected and that many things we believe to be best practices have yet to be widely adopted. Very few FindBugs users, for example, use a build system that automatically identifies and flags new issues. We're continuing studies with users and development organizations because it seems clear to us that development, measurement, validation, and adoption of best practices for static-analysis tools is key to enabling their effective use.

Acknowledgments

We thank Fortify Software for sponsoring the FindBugs project, and SureLogic, Google, and Sun Microsystems for additional support.

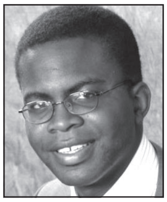
References

1. I.F. Darwin, *Checking C Programs with Lint*, O'Reilly, 1988.
2. S. Hallem, D. Park, and D. Engler, "Uprooting Software Defects at the Source," ACM Press *Queue*, vol. 1, no. 8, 2003, pp. 64–71.

3. B. Chess and J. West, *Secure Programming with Static Analysis*, 1st ed., Addison-Wesley Professional, 2007.
4. W.R. Bush, J.D. Pincus, and D.J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software Practice and Experience*, vol. 30, no. 7, 2000, pp. 775–802.
5. D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs," *Proc. 6th ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.* (PASTE 05), ACM Press, 2005, pp. 13–19.
6. J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking Defect Warnings across Versions," *Proc. 2006 Int'l Workshop Mining Software Repositories* (MSR 06), ACM Press, 2006, pp. 133–136.
7. D. Hovemeyer and W. Pugh, "Finding More Null Pointer Bugs, but Not Too Many," *Proc. 7th ACM SSIGPLAN-SIGSOFTWorkshop Program Analysis for Software Tools and Eng.* (PASTE 07), ACM Press, 2007, pp. 9–14.
8. *Reasoning Inspection Service Defect Data Report for Tomcat, Version 4.1.24*, tech. report, Reasoning, Inc., Jan. 2003.
9. T. Copeland, *PMD Applied*, Centennial Books, 2005.
10. B. Chelf, D. Engler, and S. Hallem, "How to Write System-Specific, Static Checkers in Metal," *Proc. 2002 ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.* (PASTE 02), ACM Press, 2002, pp. 51–60.
11. N. Ayewah et al., "Evaluating Static Analysis Defect Warnings on Production Software," *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.* (PASTE 07), ACM Press, 2007, pp. 1–8.
12. "Mondrian: Code Review on the Web," Dec. 2006, http://video.google.com/videoplay?docid=-8502904076440714866.
13. D. Hovemeyer and W. Pugh, "Status Report on JSR-305: Annotations for Software Defect Detection," *Companion to the 22nd ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications*, ACM Press, 2007, pp. 799–800.
14. *JSR 305: Annotations for Software Defect Detection*, http://jcp.org/en/jsr/detail?id=305.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

About the Authors



Nathaniel Ayewah is a PhD student in computer science at the University of Maryland, College Park. His research interests include understanding the way users interact with software tools and using information visualization to support creativity. Ayewah received his MS in computer science from Southern Methodist University. Contact him at ayewah@cs.umd.edu.

David Hovemeyer is an assistant professor of computer science at York College of Pennsylvania. His research interests include static analysis to find bugs. Hovemeyer received his BA in computer science from Earlham College. He developed FindBugs as part of his PhD research at the University of Maryland, College Park, in conjunction with his thesis advisor William Pugh. Contact him at david.hovemeyer@gmail.com.

J. David Morgenthaler is a senior software engineer at Google. His research interests include software tools, in particular static analysis for code visualization, refactoring, and defect detection and combinatorial algorithms for alphabetic binary trees. Morgenthaler received his PhD in computer science from the University of California, San Diego. Contact him at jdm@google.com.

John Penix is a senior software engineer in Google's Test Engineering organization, where he tries to detect more defects than he injects. He's currently working on the tools that are used to gather, prioritize and display static-analysis warnings. Penix received his PhD in computer engineering from the University of Cincinnati. He serves on the steering committee of the IEEE/ACM International Conference on Automated Software Engineering. Contact him at jpenix@google.com.

William Pugh is a professor at the University of Maryland, College Park. His research interests include developing tools to improve software productivity. Pugh received his PhD in computer science from Cornell University. He is a Packard Fellow, and invented Skip Lists, a randomized data structure that is widely taught in undergraduate data structure courses. Contact him at pugh@cs.umd.edu.