ABSTRACT

Title of dissertation: SIMPLE AND EFFECTIVE STATIC ANALYSIS TO FIND BUGS

David H. Hovemeyer, Doctor of Philosophy, 2005

Dissertation directed by: Professor William W. Pugh
Department of Computer Science

Much research in recent years has focused on using static analysis to find bugs in software. Many new approaches employing sophisticated program analysis techniques—inter-procedural, context-sensitive, or path-sensitive—have been developed. However, comparatively little work has been done on determining what bugs can be found using *simple* analysis techniques. We have found that simple static analysis techniques are effective at finding hundreds or thousands of serious software defects in several large commercial software applications.

In our research, we have attempted to characterize the bugs that can be found in production software using simple analysis techniques. Examples of simple analysis techniques include inspection of class hierarchies and method signatures, sequential scanning of program instructions using a state machine recognizer, intra-procedural dataflow analysis, and flow-insensitive whole program analysis. To determine what bugs may be found using these techniques, we performed *bug-driven* research. Starting from examples of real bugs, we tried to develop simple analysis techniques to find similar bugs. Using this approach, we found a large number of serious bugs in production applications and libraries with a relatively low percentage of false positives. The types of bugs our analysis uncovered in production code include null pointer dereferences, infinite recursive loops, data races, deadlocks, and missed thread notifications. One product of this work is a static analysis tool called FindBugs, which analyzes Java programs at the bytecode level. We have distributed FindBugs under an open-source license, and it has been widely adopted by commercial organizations and open-source projects. FindBugs has been downloaded more than 110,000 times since its initial release.

Our work makes several contributions to the field. First, we have cataloged many commonly-occurring bug patterns, described effective ways of finding occurrences of those patterns automatically, and classified common reasons why these bugs occur. Second, we have measured the accuracy of our bug detectors on production software and student programming projects, validating the effectiveness of the underlying static analysis techniques. Finally, we have described techniques for determining when static analysis warnings are added or removed as software evolves.

SIMPLE AND EFFECTIVE STATIC ANALYSIS TO FIND BUGS

by

David H. Hovemeyer

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

This dissertation is dedicated to my parents.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Professor William Pugh, for guiding me through the wilderness that is graduate school. I have had the privilege of working with him on a variety of interesting projects, and his approach to problem solving has been an inspiration. Whenever I face a difficult problem I always ask myself “What would Bill do?”

The members of my dissertation committee gave me a huge number of valuable suggestions on content and presentation.

I could not have gotten through my Ph.D. without the support of my friends and family. My deepest thanks go to Chadd Williams, Jaime Spacco, Jeremy Manson, Reiner Schulz, Debbie Heisler, Brian Postow, Jordan Landes, Adam Lopez, Jennifer Baugher, and the Bridges family: Adrienne, Don, Hannah, and Molly.

I would like to thank my parents, Lois and Hank Hovemeyer, for their constant love and support throughout my life.

Finally, and most of all, I would like to thank Kate Swope for sharing her life with me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

According to a NIST study published in 2002 [57], bugs in software cost the U.S. economy \$60 billion annually. Thus, research on techniques to find bugs early in the development process, before they enter production systems, is an important research topic. In this dissertation, we show that a significant class of bugs can be found through simple, local static analysis.

1.1 What is a Bug?

Before we can think about techniques to find bugs in software, we must first understand what bugs are. A software bug is a *defect* in the software: a failure of the software to work as intended. This definition points out a critical problem: in order to identify bugs in a software artifact, we need to know what the software is intended to do.

Unfortunately, the intent behind software—what it is supposed to do—is often nebulous. Most software is developed in an environment of vague and constantly changing requirements. Since these requirements are never known with certainty, the scope of any automatic technique to find bugs is limited. For example, an automatic tool will not be able to diagnose the fact that a software artifact does not work as intended because an important requirement was never conveyed to the developers of the software.

In the absence of requirements and specifications, there are still two kinds of defects we can find. First, some software behaviors may be considered erroneous no matter what the software is intended to do. For example, there are very few situations where dereferencing a null pointer results in useful behavior, so classifying null pointer dereferences as bugs should be uncontroversial. Another, more subtle, means of finding defects is looking for *inconsistencies* in the code. For example, checking a condition usually implies that the programmer believes that the condition can be either true or false. If a condition is actually predetermined at compile time, it may mean that the programmer did not understand important code invariants, suggesting that a logic error exists in the code. This way of finding bugs is interesting because it does not directly identify a defect; instead, it identifies code suspected of being related to an unknown defect.

Neither of these approaches identify bugs with 100% certainty. Occasionally, a method will dereference a null pointer deliberately in order to raise a runtime exception. A condition whose behavior is fixed at compile time may simply indicate that debugging code is present. Even so, techniques that identify *probable* bugs are still valuable. As long as the probability of finding a real bug is sufficiently high to make inspecting a report worth a developer's time, these approaches may be used productively to find real errors in software.

1.2 Techniques to Find Bugs in Software

Many techniques exist to find bugs in software, each of which has different advantages and disadvantages.

In *software testing*, software is executed with varying input to determine if the correct output is produced in each case. *Unit tests* are used to check fine-grained behavior, such as individual functions, methods, and classes. Larger units of code may be tested at the module, component, subsystem, or system level. Testing is useful to verify that functionality at each level of the software system works as intended. Because tests can generally be automated (run without human intervention), they may be applied frequently. For example, a test suite may be executed nightly to help detect bugs that were introduced since the last testing cycle.

The main limitation in testing is that it is notoriously difficult to construct a test suite that achieves a high degree of code coverage. For example, error paths are difficult to exercise through testing. In general, even relatively simple programs can exhibit an infinite number of possible behaviors at runtime, making exhaustive testing an impossibility. For these reasons, it is easy to use testing to verify that basic software functionality works as expected, but difficult to use testing to find as yet unknown bugs.

In *code inspection* [28], software developers inspect source code to look for bugs. One advantage of code inspection over testing is that humans can employ high-level reasoning about code to find erroneous behaviors. Code inspections also have the advantage of easily analyzing difficult-to-execute regions of the program, *In class org.eclipse.update.internal.core.ConfiguredSite*

```
if (in == null)
    try {
        in.close();
    } catch (IOException e1) {
    }
```

Figure 1.1: A trivial null pointer bug in Eclipse 3.0.1

such as error handling paths.

The main limitation of code inspections is that they are time-consuming, and therefore expensive, to perform. A more subtle limitation of code inspections is subjectivity: human readers are sometimes influenced by what code looks like it should do, rather than what it actually does. As evidence, consider the code in Figure 1.1 taken from Eclipse 3.0.1 [19]. At first glance, it is obvious that this code is meant to close the input stream `in` if it is non-null. However, the programmer inadvertently used the wrong reference comparison operator (`==` instead of the correct `!=`), resulting in code which closes the input stream only if it *is* null. Although we do not know whether or not this particular code was ever the subject of a code inspection, we do know that this was one of about 30 similar null pointer dereference bugs in the same version of Eclipse.

A *correctness proof* could be considered the ultimate guarantee that a program is free of bugs. However, full correctness proofs are not widely used, for a variety of reasons. The main reason is that a proof would require a complete specification of the program's behavior, and such specifications are extremely difficult to construct. Also, in most software projects requirements evolve over time, resulting in the additional complication of making the specifications track a moving target. Finally, even given complete specifications, constructing a proof is an extremely difficult task. Language features such as finite arithmetic, heap memory allocation, concurrency, and I/O are difficult to model formally.

In addition to testing, code inspections, and proofs, there are many other ways to find bugs and improve software quality. For example, the software process itself can be studied and improved. For example, Extreme Programming [7] practices, such as pair programming, have been widely adopted. However, in this dissertation we are interested in using *static analysis* to find bugs.

1.3 Static Analysis

In *static analysis*, the source code or object code of a program P is fed as input to another program S to determine properties (or likely properties) of P. A static analysis to find bugs would produce as output some number of *warnings*, each warning representing a potential bug in the program.

Using static analysis to find bugs has some advantages over the traditional quality assurance techniques of testing and manual code inspections. Unlike testing, static analysis can easily check hard-to-execute code paths such as error-handling code. Compared to manual code inspection, static analysis is less easily confused by what code appears to do, and is relatively inexpensive to apply. For these reasons, static analysis to find bugs is a very active research area, and increasingly is becoming a standard part of the quality assurance toolbox in real development projects.

The obvious limitation of static analysis is that most nontrivial program properties are undecidable. As a result, any static analysis technique must approximate the behavior of the input program P. For this reason, no static analysis technique can be both *complete* and *sound*. In the literature on using program analysis to find bugs, *sound* generally means "finds every real bug", and *complete* generally means "reports only real bugs."

In this dissertation, we will consider the result of applying a static analysis to a program as producing some number of *warnings*. Each warning describes a potential bug in the program. We refer to warnings which describe genuine software defects as *accurate warnings* or *real bugs*. We refer to warnings which do not describe real defects as *false positives*. Finally, if a real defect exists in a program, but it is not described by any warning, it is a *false negative* or *missed bug*. Using this terminology, a sound analysis is free from false negatives, and a complete analysis is free from false positives.

Deciding how to approximate in a static analysis has important consequences. Both soundness and completeness are desirable properties. In theory, a sound analysis is able to find every real instance of the kind of bugs the analysis is designed to detect. However, it might do this by reporting 1,000 false positives for every accurate warning, making use of the analysis unproductive. Similarly, a complete analysis would only report definite bugs. However, it might find only a very small number of bugs, leaving a large number of false negatives. For these reasons, tools which are neither complete nor sound can serve a valuable role in the software quality assurance process as long as they find a significant number of real bugs without emitting too many false positives.

Another limitation of static analysis, as noted earlier, is that in the absence of specifications it is impossible to know with certainty what software is intended to do.

1.4 Thesis

The thesis for our research is as follows:

> Simple analysis techniques can find an interesting class of serious bugs in production software written in type-safe object-oriented programming languages.

In recent years there much research has focused on static analysis techniques to find bugs in software. Many of these techniques have used sophisticated analysis—inter-procedural analysis, path- and context-sensitive analysis, heap modeling—in order to find "deep" bugs. This work is valuable, and some of these techniques (e.g., [9, 20, 15]) have become useful tools. Given unlimited resources, it would be interesting to fully explore the space of bugs that can be found using static analysis, including deep bugs requiring deep analysis.

However, not all bugs are "deep". Many static analysis tools have been developed to find common bugs (e.g., [45, 26]). However, most of these tools were developed to find relatively low-level errors in unsafe languages such as C and C++. At the time of writing, a significant percentage of mainstream software development is done in higher-level, type-safe languages such as Java [35] and C# [11]. Although some work has been done on analysis techniques to find common bugs in such languages (e.g., [4, 61]), tools based on these techniques are not widely used in practice. For this reason, we were interested in finding out what kinds of bugs can be found in safe object-oriented languages (such as Java) using simple analysis techniques. By systematically exploring the space of such bugs, we felt we could accomplish two main objectives. First, we wanted to gain a better understanding of why bugs occur in the first place. Second, by understanding the low-hanging fruit, we felt we could help inform research on where more sophisticated analysis techniques could be applied most profitably.

Our strategy for exploring the space of easily-detectable bugs was *bug-driven research.* Starting from examples of bugs in real software, we designed and implemented simple analysis techniques which would find occurrences of similar bugs. The result of this effort is a static analysis tool called FindBugs [29]. The term we use for a common, easily-recognizable type of bug is *bug pattern* [1]. In many cases, occurrences of bugs patterns may be effectively identified by looking for indirect signs that a defect might be present, without requiring an exhaustive analysis proving the existence of a defect. Our experience has shown that signs associated with the occurrence of a bug pattern can often be recognized using simple analysis techniques, and that detectors for those patterns find a surprising number of real bugs—hundreds or thousands—in production software artifacts.

1.5 Contributions

Our research makes the following contributions:

1. We describe many common types of bugs affecting Java programs, reasons why they occur, and effective static analysis techniques for identifying occurrences. Several types of bugs we documented were surprising to find in production software.

2. We describe the implementation FindBugs, a static analysis tool to find bugs in Java programs.

3. We evaluate FindBugs on real software artifacts, including both production systems and student programming projects, and show that they find a large number of real bugs with an acceptable rate of false positives. We also measure the false negative rate on student programming projects, showing that for some types of bugs FindBugs finds a significant percentage of all occurrences of those bugs.

4. We discuss experiences from software projects that have used FindBugs, showing that FindBugs finds serious defects, and has been judged useful enough to be a regular part of the development process.

Chapter 2

Bug Patterns

In this chapter we define the general concept of *bug pattern*, and discuss general implementation techniques for automatically recognizing occurrences of bug patterns.

2.1 What is a Bug Pattern?

Many bugs in software are subtle, and would require sophisticated analysis to find. However, a surprising number of bugs in real software are blatant and relatively obvious, such as the null pointer dereference shown in Figure 1.1.

Some simple mistakes are caught at compile time: for example, the Java source to bytecode compiler will catch all syntax and static type errors. However, some bugs (such as the Eclipse null pointer bug) may be thought of as being one step removed from a syntax error. Many such bugs have characteristics that make them relatively easy to identify using static analysis. We refer to these bugs informally as occurrences of *bug patterns*. We use the term "bug pattern" as a way of emphasizing that many bugs are caused by similar mistakes, manifest in similar ways, and may be recognized by looking for the common features they share.

2.1.1 Discovering and Recognizing Bug Patterns

In discussing bug patterns, there are two main questions to answer:

1. How can bug patterns be discovered?

2. What analysis techniques can find occurrences of bug patterns?

We have addressed both questions by conducting *bug-driven* research: we find as many examples of real bugs as possible, determine why they occur, look for easily-recognizable signs indicating their presence, and then develop analysis techniques to find occurrences in real software automatically.

Examples of bugs (and thus ideas for bug patterns) come from many sources: books, articles, student programming projects, bug databases, and our own experience. Any time a bug is identified in some software artifact is an opportunity to discover other, similar bugs. From this perspective, bugs in software are valuable because they can be used to improve the quality of the code in which they occur, and potentially other software as well. As an analogy, consider the role a vaccine plays in building a stronger immune system. A vaccine presents neutralized specimens of a pathogen to the immune system so antibodies can be developed to recognize and eliminate instances of the pathogen in the future. Similarly, once a software defect has been isolated and neutralized, we can develop a static analysis to recognize similar defects in the future.

As a platform for our research on easily-detectable software defects, we have developed a static analysis tool called FindBugs [29]. At the time of writing, we have implemented detectors for 104 different bug patterns in FindBugs. To implement these detectors, we have used a wide variety of analysis techniques. In general, we use the simplest technique that will find and identify some occurrences of the pattern in real software without producing too many false positives. Some of the analysis techniques we use include:

- Inspecting method names, signatures, and inheritance
- Using a sequential scan over bytecode instructions to drive a state machine recognizer—analogous to peephole optimization
- Intra-procedural dataflow analysis [46]
- Flow-insensitive whole-program analysis

2.1.2 Bugs vs. Style

In discussing the use of automatic tools to check code, it is crucial to distinguish *bug checkers* from *style checkers.* A bug checker uses static analysis to find code that violates a specific correctness property, and which may cause the program to misbehave at runtime. A style checker examines code to determine if it contains violations of particular coding style rules. The key value of enforcing coding style rules is that when a consistent style is used throughout a project, it makes it easier for the developers working on the project to understand each other's code, and to avoid unintentional deviations from good practice. Some style rules, such as "do not use fall-through in `switch` statements", and "always use braces around the clauses of an `if` statement", may help prevent certain kinds of bugs. However, violations of those style guidelines are not particularly likely to be bugs. For example, always putting braces around the clauses of an `if` statement might only prevent a bug 1 time out of 100. This is a rule worth following when writing new code, but would not be a particularly effective way to find bugs in existing code.

Tools that focus mainly on checking style, such as PMD [55] and CheckStyle [10], are widely used. However, bug checkers are not nearly as widely used, even though they are more likely to direct developers to specific real bugs in their code. We believe the reason for this disparity is that more work is required to use the output of bug checkers. Style checkers can be extremely accurate in determining whether or not code adheres to a particular set of style rules. Therefore, little or no judgment is needed to interpret their output, and changing code to adhere to style guidelines has a tangible effect on improving the understandability of the code. In contrast, bug checkers, especially ones that use unsound analysis, may produce warnings that are inaccurate or difficult to interpret. In addition, fixing bugs reported by a bug checker requires judgment in order to understand the cause of the bug and to fix it without introducing new bugs.

Another problem with bug checkers is that the percentage of false warnings tends to increase over time, as the real bugs are fixed. Techniques for suppressing false warnings must be used so that developers can concentrate on evaluating and fixing the real problems identified by the bug checker.

We believe that tools based on simple local analysis represent a useful sweet spot in the design space for bug checkers, for several reasons. Aside from being easy to implement, they tend to produce output that is easy for programmers to understand. Because they focus on finding deviations from accepted practice, they tend to be very effective at finding real bugs, even though they do not perform deep analysis. With tuning, they can achieve an acceptably low rate of false warnings. All of these factors contribute to overcoming barriers to wider adoption of bug checkers by developers.

2.2 Catalog of Bug Patterns

In this section, we present an informal taxonomy of bug patterns which we have observed in real programs and implemented in FindBugs. While this collection of patterns is by no means exhaustive, it does shed some light on common reasons why errors occur. Only a brief synopsis of each pattern is presented in this section; for a more complete description of each pattern, see Appendix A.

2.2.1 Classification

We have classified each bug pattern according to three main criteria: the category of error, the bug pattern family, and typical causes.

Categories are *correctness*, *multithreaded correctness*, *performance*, and *malicious code vulnerability*. Correctness is the most general category: these are bugs whose presence would be a serious problem for any kind of program. The other categories describe bugs which might apply only in some situations. For example, the occurrence of a multithreaded correctness pattern can be safely ignored if it occurs in a program which is single-threaded. Performance bugs may only be significant when they occur on certain code paths, such as in tight loops.

Often, bug patterns appear in multiple variations which are worth distinguishing. For example, we have identified two patterns involving suspicious calls to the `equals()` method. We refer to these groups of related patterns as a *family*, which is designated by a short code. For example, the family code for suspicious `equals()` calls is EC, for "Equals Comparison."

We have broken down typical causes of bugs as follows. Note that in the summary tables in this chapter we have used abbreviations for these causes. These abbreviations are given in parentheses after the name of each cause.

Typo (Typo) Many bugs occur because of simple typographical mistakes: for example, using the `==` operator instead of the `!=` operator.

Misunderstood API (API) The runtime library for J2SE 1.5 contains more than 3,700 classes and interfaces defining more than 21,000 public methods (not counting overridden method variants in subtypes). Each class, interface, and method has the potential to be used incorrectly.

Misunderstood language feature (Lang) Although Java is a fairly simple and well-designed language, it has its share of sharp corners.

Misunderstood invariants (Inv) Many mistakes are a result of programmers failing to understand the invariants of a method or class: for example, assuming a parameter will never be null even though callers may pass a null value.

Novice mistake (Novice) Some errors are often seen in code written by novice Java programmers.

Logic error (Logic) This is a catch-all designation for patterns whose root causes did not always fit one of the other classifications.

2.2.2 Summary and Discussion

Tables 2.1 through 2.5 list the bug patterns currently implemented in FindBugs. Table 2.6 summarizes the prevalence of the various causes. Although our taxonomy is informal, and not meant to be a rigorous analysis of why bugs occur, several interesting patterns do emerge.

Stupid mistakes are very common. Of the 104 patterns FindBugs currently recognizes, 35 are bugs frequently caused by typos. This is somewhat surprising, since as programmers we generally expect the compiler to find our lexical and grammatical mistakes. However, a surprising number of typos not only pass the compiler's syntax and type checking, but also appear to be correct on casual inspection. In addition, many of the patterns not explicitly classified as typos are the result of similar oversights; for example, forgetting to initialize a field in a constructor.

API misuse is very common. Misunderstood or misused APIs are the single most prevalent cause in the bug patterns we have implemented; 47 of the 104 patterns are classified as API issues. There are two main reasons why this is so. First, the standard Java runtime library contains an enormous number of public classes and methods, as we noted earlier. Second, many of these APIs have surprisingly subtle or even difficult requirements for correct use; these requirements are not checked by the compiler.

Family	Pattern	Common causes
AM	Empty jar or zip file entry	API, Logic
BC	Impossible cast	Typo, Novice, Logic
BIT	Incompatible bit masks	Logic, Typo
BRSA	Bad ResultSet access	API
CN	Class implements Cloneable, but does not implement clone()	API
CN	Clone method does not call super.clone()	API
Co	Covariant compareTo() method	API
DE	Dropped exception	Logic, API
DLS	Dead local store	Logic, Typo
DMI	Passing invalid constant value for a month	API
Dm	Method calls runFinalizersOnExit()	API
EC	Invoking equals() on an array object	API
EC	Calling equals() on unrelated types	Logic, API
Eq	Covariant equals() method	API
FE	Equality comparison of floating point values	Lang
FI	Explicit invocation of finalizer	Lang
FI	Finalizer does not call super.finalize()	Lang
HE	Equal objects should have equal hashcodes	API
ICAST	Invalid integer shift	Logic, Typo
ICAST	Integer division result cast to double	Logic, Lang
ICAST	Integer value cast to double, then passed to Math.ceil()	Logic, Lang
IL	A container is added to itself	Logic, Typo
IL	Infinite recursive loop	Typo
IM	Integer multiplication of result of integer remainder	Typo, Lang
IMSE	Dubious catching of IllegalMonitorStateException	Novice
IP	A parameter is dead upon entry to method but overwritten	Typo, Novice
It	Iterator next() method can't throw NoSuchElementException	API
J2EE	Store of non serializable object into HttpSession	API
NP	Null pointer dereference in method	Typo, Logic, Inv
NP	Possible null pointer dereference in method	Typo, Logic, Inv
NP	equals() method does not check for null parameter	API
NP	Immediate dereference of the result of readLine()	API
NP	Passing null for unconditionally dereferenced parameter	Inv
NP	Read of unwritten field	Typo
NS	Questionable use of non-short-circuit logic	Typo
Nm	Class defines equal(), not equals()	Typo
Nm	Confusing method names	Typo
Nm	Class defines hashcode(), not hashCode()	Typo
Nm	Class defines tostring(), not toString()	Typo
ODR	Method may fail to close database resource	Logic
OS	Method may fail to close stream	Logic
QF	Complicated, subtle or wrong increment in for-loop	Typo
RC	Suspicious reference comparison	Lang, API
RCN	Redundant comparison of non-null value to null	Inv, Logic
RCN	Redundant comparison of two null values	Inv, Logic
RCN	Nullcheck of value previously dereferenced	Inv, Logic
RE	Invalid syntax for regular expression	Typo, Novice, API
RR	Method ignores results of InputStream.read()	API

Table 2.1: Correctness patterns

Family	Pattern	Common causes
RR	Method ignores results of InputStream.skip()	API
RV	Random value from 0 to 1 coerced to 0	API, Novice
RV	Checking if result of String.indexOf() is positive	API, Typo
RV	Discarding result of readLine() after checking if it is nonnull	Novice, API
RV	Method ignores return value	API, Novice
SA	Self assignment of field	Typo
Se	Instance field in serializable class not serializable	API, Inv
Se	serialVersionUID declared incorrectly	Typo, API
Se	Parent of Serializable class has no void constructor	API
Se	Externalizable class has no void constructor	API
SnVI	Class is Serializable, but doesn't define serialVersionUID	API
UCF	Useless control flow	Typo
UR	Uninitialized read of field in constructor	Logic, Typo
UwF	Field only ever set to null	Logic

Table 2.2: Correctness patterns (continued)

Family	Pattern	Common causes
DC	Double check of field	Lang
Dm	Monitor wait() called on Condition	Typo, API
Dm	A thread was created using the default empty run method	Logic
IS2	Inconsistent synchronization	Logic
JLM	Synchronization performed on java.util.concurrent Lock	Typo, API
LI	Lazy initialization of static field	Lang, Logic
ML	Method synchronizes on an updated field	Typo, Lang
MWN	Mismatched notify()	Typo, Novice, API
MWN	Mismatched wait()	Typo, Novice, API
NN	Naked notify in method	Logic, Lang, API
No	Using notify() rather than notifyAll() in method	Logic, Lang
RS	Class's readObject() method is synchronized	API, Logic
Ru	Invoking run() on a thread	API, Typo
SC	Constructor invokes Thread.start()	Logic, Lang
SP	Method spins on field	Novice, Lang
SWL	Method calls Thread.sleep() with a lock held	Logic
TLW	Wait with two locks held	Logic
UG	Unsynchronized get method, synchronized set method	Lang, Logic, Typo

Table 2.3: Multithreaded correctness patterns

Family	Pattern	Common causes
EI	Returning reference to mutable object	Logic
EI2	Incorporating reference to mutable object	Logic
FI	Finalizer should be protected, not public	API
MS	Storing mutable object into static field	Logic
MS	Static field could be modified by untrusted code	Logic, Typo
MS	Returning mutable array	Logic
MS	Field is a mutable array	Logic
MS	Field is a mutable Hashtable	Logic

Table 2.4: Malicious code patterns

Family	Pattern	Common causes
Dm	Method invokes dubious Boolean constructor	API
Dm	Method allocates a boxed primitive just to call toString()	API
Dm	Explicit garbage collection	API
Dm	Method allocates an object, only to get the class object	Lang, API
Dm	Calling Random.nextDouble() to generate a random integer	API
Dm	Invoking dubious new String(String) constructor	API
Dm	Method invokes toString() method on a String	API
Dm	Method invokes dubious new String() constructor	API
FI	Empty finalizer method	Lang
FI	Finalizer does nothing but call superclass finalizer	Lang
ITA	Method uses toArray() with zero-length array argument	API
SBSC	Method concatenates strings using + in a loop	API, Lang
SIC	Inner class should be made static	Lang, Typo
SS	Unread final instance field	Typo
UrF	Unread field	Typo, Logic
UuF	Unused field	Typo, Logic

Table 2.5: Performance patterns

Category	API	Lang	Inv	Typo	Logic	Novice
Correctness	29	7	7	23	19	7
Multithreaded Correctness	7	8	0	7	10	3
Malicious Code	1	0	0	1	7	0
Performance	10	5	0	4	2	0
All	47	20	7	35	38	10

Table 2.6: Summary of bug pattern causes by category and overall

Java mixes several programming paradigms. Perhaps more than any other mainstream language (other than scripting languages), the Java language and its runtime library freely mix several different programming paradigms. Although nominally object-oriented, Java is not a pure object-based language, which can be seen in features such as primitive types (which are not objects) and static methods. Classes such as `String` and `Integer` are immutable, and employ a nearly-functional style of programming: however, they cannot be compared by value using the standard comparison operators. This inconsistency can cause real problems for programmers, especially novice programmers. The “Method ignores return value” and “Suspicious Reference Comparison” patterns are good examples of the confusion caused by mixed programming paradigms.

Chapter 3

Selected Bug Patterns and Implementation Techniques

In this section we describe in detail some of the bug patterns for which we have implemented detectors in FindBugs. The patterns we have selected for this chapter are those which are especially likely to represent serious errors, are subtle or surprising, or for which we have developed a novel analysis technique.

All of the patterns and implementation techniques we describe here are realized in FindBugs [29], a static analysis tool to find bugs in Java programs. FindBugs works by analyzing Java class files [50] at the bytecode level. Java bytecode is semantically quite close to Java source code, and by making use of source line number tables found in Java class files, the results of analyzing bytecode can be easily mapped back to source code.

3.1 Covariant Equals

Java classes may override the `equals(Object)` method to define a predicate for object equality. This method is used by many of the Java runtime library classes; for example, to implement generic containers.

Programmers sometimes mistakenly use the type of their class `Foo` as the type of the parameter to `equals()`:

```
public boolean equals(Foo obj) {...}
```

```
class Count {
  int value;

  public Count(int value) {
    this.value = value;
  }

  public boolean equals(Count other) {
    return other != null && this.value == other.value;
  }

  public static void main(String[] args) {
    Count c1 = new Count(42);
    Count c2 = new Count(42);

    Object obj = c2;

    System.out.println(c1.equals(c2));  // prints "true"
    System.out.println(c1.equals(obj)); // prints "false"
    System.out.println(obj.equals(c1)); // prints "false"
  }
}
```

Figure 3.1: Example of a class with a covariant equals method

This covariant version of `equals()` does not override the version in the `Object` class, which may lead to unexpected behavior at runtime, especially if the class is used with one of the standard collection classes which expect that the standard `equals(Object)` method is overridden.

This kind of bug is insidious because the code appears to be correct, and will work correctly in some situations. For example, Figure 3.1 shows an example of a class, `Count`, defining a covariant `equals()` method. The method is only called when both the receiver object and the object passed as the argument have the compile-time type `Count`. If either the receiver or argument has the superclass type `Object`, then the default `equals()` method will be called. Since the default implementation compares reference equality, it is likely to yield different results than the covariant `equals()`. Thus, the first time an instance of a class defining a covariant `equals()` method is used in a generic container or other context where `equals(Object)` might be called, the program will quietly perform the wrong computation. Because this kind of error does not generally result in obvious misbehavior, such as a runtime exception, it may elude testing.

Detecting instances of this bug pattern using static analysis simply involves examining the method signatures of a class and its superclasses to see if the class only defines or inherits a covariant version of the `equals()` method.

3.2 Equal Objects Must Have Equal Hash Codes

In order for Java objects to be stored in hash data structures, such as `HashMap` and `HashSet` objects, they must implement both the `equals(Object)` and `hashCode()` methods. Objects which compare as equal must have the same hashcode.

Consider a case where a class overrides `equals()` but not `hashCode()`. The default implementation of `hashCode()` in the `Object` class (which is the ancestor of all Java classes) returns an arbitrary value assigned by the virtual machine. Thus, it is possible for objects of this class to be equal without having the same hashcode. Because these objects would likely hash to different buckets, it would be possible to have two equal objects in the same hash data structure, which violates the semantics of `HashMap` and `HashSet`.

As with covariant equals, this kind of bug is hard to spot through inspection.

There is nothing to "see"; the mistake lies in what is missing. Because the `equals()` method is useful independently of `hashCode()`, it can be difficult for novice Java programmers to understand why they must be defined in a coordinated manner. This illustrates the important role tools can play in educating programmers about subtle language and API semantics issues. (We will return to this issue in Section 4.5.1.)

Automatically verifying that a given class maintains the invariant that equal objects have equal hashcodes would be very difficult, and is undecidable in general. Our approach is to check for the easy cases, such as

- Classes which redefine `equals(Object)` but inherit the default implementation of `hashCode()`

- Classes which redefine `hashCode()` but do not redefine `equals(Object)`

Checking for these cases requires simple inspection of method names and signatures and the inheritance hierarchy.

3.3 Null Pointer Dereference and Redundant Comparison to Null

Exceptions due to dereferencing a null pointer are a very common type of error in Java programs. Some null pointer bugs arise because of null values loaded from the heap or passed from a distant call site: such bugs would require sophisticated analysis to find. However, our experience studying production Java applications and libraries has shown that many null pointer bugs are the result of simple mistakes, such as using the wrong boolean operator. An example of a simple null pointer bug is shown in Figure 3.2. The programmer clearly meant to use the `&&` operator, and

// javax/xml/soap/SOAPPart.java, line 38

```
public String getContentId()
{
   String[] header = getMimeHeader("Content-Id");
   String id = null;
   if( header != null || header.length > 0 )
      id = header[0];
   return id;
}
```

Figure 3.2: An obvious null pointer dereference in JBoss 4.0.0RC1

not the `||` operator, before taking the length of the array `header`. In our work, we have found that simple coding errors like this one are surprisingly common, even in production code.

In this section, we describe the analysis used in FindBugs to determine where null pointer dereferences might occur. The design of this analysis illustrates some interesting examples of trade-offs needed to find real bugs without producing too many false warnings.

3.3.1 Scope

One of the main problems in static analysis is determining which paths through the program are feasible. Even analyzing the possible control flow of a single method is extremely difficult, since the runtime behavior of the method depends on the behavior of called methods and values stored in heap objects. To construct an analysis useful for finding bugs, we must ensure that the issues we report are likely to represent real problems.

Since our null pointer analysis focuses on finding bugs resulting from simple mistakes, we have chosen not to worry about bugs with complex preconditions. Instead, we have adopted a very simple set of criteria for which bugs to report:

1. All null pointer dereferences guaranteed to occur if full statement coverage is achieved are reported as high priority warnings.

2. All null pointer dereferences guaranteed to occur if full branch coverage is achieved are reported as medium priority warnings.

These criteria rely on an implicit specification: that each statement and branch in a program should serve a useful purpose. Programmers generally do not introduce unreachable code or infeasible branches intentionally[1]. If executing any single statement or taking any single branch is guaranteed to cause a null pointer exception, it is a strong indication that the code contains a logic error.

In general, our analysis reports exactly the pointer dereferences described by these criteria, with some additional restrictions designed to reduce false positives due to infeasible exception control flow. The details of the null pointer analysis are described in the remainder of this section.

3.3.2 The Basic Analysis

The null pointer analysis is a forward intra-procedural dataflow analysis performed on a control-flow graph representation of a Java method. The dataflow values are

[1]Assertions are an exception to this rule, since they may add infeasible control flow intended to guard against incorrect code changes in the future. A static analysis may consider assertions to be a special case, and avoid reporting bugs having to do with code reached when an assertion is triggered.

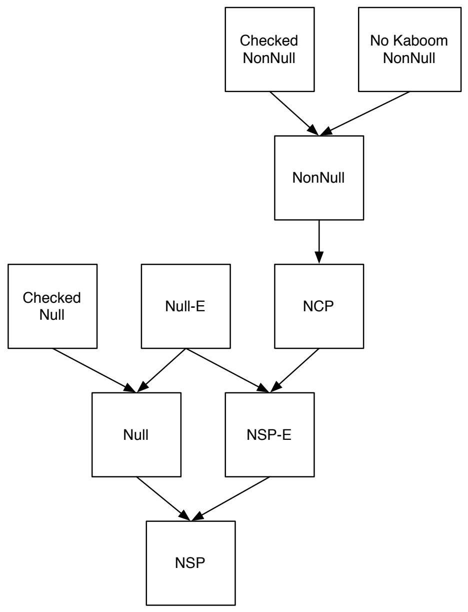


Figure 3.3: Meet-semilattice for dataflow values used in null pointer analysis

Java stack frames containing "slots" representing method parameters, local variables, and stack operands. Each slot contains a single symbolic value indicating whether the value contained in the slot is definitely null, definitely not null, or possibly null. The meet-semilattice of symbolic values is shown in Figure 3.3, and the meaning of each value is described in Table 3.1.

3.3.3 Modeling Values

On method entry, parameter values are assumed to be NCP, Null on a Complex Path. This is a conservative assumption reflecting the fact that the analysis has no *a priori* justification to assume that parameters are either null or non-null. An exception is the reference to the receiver object (`this`), which is set to NonNull in instance methods.

Statements are modeled as shown in Table 3.2. In this table, `p` refers to the

		Warning if ...	
Value	Meaning	Dereferenced	Compared to null
Null	Value definitely null	High	Medium†
Checked Null	Value null due to comparison	High	Medium†
Null-E	Value null on exception path	Medium	Medium†
NonNull	Value definitely non-null	—	Medium†
Checked NonNull	Value non-null due to comparison	—	Medium†
No Kaboom NonNull	Value non-null because it was dereferenced	—	High
NSP	Null on Simple Path	Medium	—
NSP-E	Null on Simple Path due to exception	Low	—
NCP	Null on Complex Path	—	—

†Low priority if the check does not create dead code.

Table 3.1: Symbolic values used in the null pointer analysis

Statement	Value of `p`
`p = null`	Null
`p = this`	NonNull
`p = new ...`	NonNull
`p = "string"`	NonNull
`p = Foo.class`	NonNull
`p = q.x`	NCP
`p = a[i]`	NCP
`p = f()`	NCP

Table 3.2: Modeling statements in the null pointer analysis

value computed by an expression—in other words, the value that results by executing a statement. `q` refers to an arbitrary object reference, `a` refers to an arbitrary array reference, and `Foo` refers to an arbitrary class. In general, the analysis treats unknown values as NCP (Null on Complex Path) unless they are definitely null or definitely non-null.

3.3.4 Control Flow

Values may become non-null by being dereferenced. If a dereference instruction (such as an instance method call or field access) does *not* throw a null pointer exception, then we know that the value was not null. One useful fact to keep track of following a successful dereference was whether or not the value was definitely non-null *before* the dereference. If it was potentially null (NSP or NCP), we represent it with a special "No Kaboom" non-null value, to reflect the fact that a null pointer exception could have been thrown, but was not.

Note that a dereference of a value known to be Null does *not* produce a No Kaboom value; because a null pointer exception is guaranteed, the analysis knows the non-exception edge leading from the dereference is infeasible, and sets the stack frame on that edge to a special "Top" value marking it as dead.

Control joins are modeled in the usual way, by taking the meet in the lattice of the values in corresponding slots. Where possible, branches on comparisons to null are used to gain information about the tested value: we use the direction of the resulting branch to make the value either Null or NonNull on the appropriate control edge.

```
class Foo implements Cloneable {
  HashSet contents;
  ...

  public Object clone() {
    Foo dup = null;
    try {
      dup = super.clone();
    } catch (CloneNotSupportedException e) {
      // Can't happen
    }

    // Make deep copy of contents
    dup.contents = (HashSet)dup.contents.clone();

    return dup;
  }
}
```

Figure 3.4: An infeasible exception handler

Exception paths are handled specially. On entry to an exception handler, all Null values are replaced with Null-E, and all NSP values are replaced with NSP-E. We maintain this distinction because it is common for some call sites targeting methods declaring a checked exception to know *a priori* that the checked exception cannot occur. A typical example involving an infeasible `CloneNotSupportedException` handler is shown in Figure 3.4. By keeping track of values which are only null on an exception path, we can lower the precedence of warnings emitted for dereferences of such values.

3.3.5 Value Numbering

Java bytecode is stack-based, rather than register-based. Unlike register-based code, most Java bytecode instructions consume values by popping them from the stack, and produce results by pushing new values onto the stack. As a result, a reference to the same object may be on the stack several times simultaneously. Whenever we encounter an instruction which gains information about one instance of a duplicated value (usually the top value on the stack), we must ensure that the information is propagated to all other copies of the value in the stack frame. Examples of instructions where this is necessary include dereferences and null comparisons.

FindBugs implements a dataflow analysis to assign each local variable and stack operand a value number. Values with the same value number are guaranteed to be the same at runtime. Therefore, when we encounter an instruction that gains information about whether or not a value is null, such as `ifnull`, we propagate that information to all other local variables and stack operands sharing the same value number.

3.3.6 Infeasible Paths

As noted earlier, infeasible paths are a common source of inaccuracy in static analysis. In the context of an analysis to find null pointer dereferences, the problem is determining when a condition implies that a value will be guaranteed to be null or non-null.

Often, a dereference of a possibly-null value v will be guarded by an explicit comparison of v to null, with the dereference occurring only when v is not null. In such cases, it is easy for the analysis to determine that the dereference of v is protected. A more difficult analysis problem arises when some other condition b guards the dereference of v. If b implies $v \neq$ null then the dereference of v is safe. An example of an indirect null check is shown in Figure 3.5. A naïve analysis might assume that `p` could be null at the call to `p.f()`, resulting in a false warning. However, that is only possible on the infeasible path $2 \rightarrow 3$.

Unfortunately, determining which conditions entail a guarantee that a value is or is not null is a difficult problem (and is undecidable in general). In some cases, the condition checked may originate from outside the scope of the current analysis (such as a parameter passed into the method). Rather than trying to use a sophisticated analysis to untangle the meaning of conditions, we simply assume that *all* conditions are opaque null checks. This is accomplished by changing all NSP (Null on a Simple Path) values to NCP (Null on a Complex Path) on branches.

Because no warnings are reported when NCP values are dereferenced, this technique misses some real bugs. However, it ensures that a very high percentage of warnings that are reported do correspond to real errors. If it is possible to achieve full branch coverage in a method, then every dereference of a NSP is a real, exploitable bug. In the future we may augment the analysis to be smarter about inferring the effects of conditions which are not direct null checks. (For one approach to efficient and precise analysis of program states implied by boolean conditions, see [15].)

3.3.7 Redundant Comparisons

Interestingly, a significant number of null comparisons occurring in real programs are redundant, because the value compared to null is either definitely null or definitely non-null. Sometimes, this is because of defensive programming. A less benign cause of redundant comparisons is that a value was unconditionally dereferenced, and

```
boolean b;

if (p != null)
  b = true;
else
  b = false;

if (b) {
  p.f();
}
```

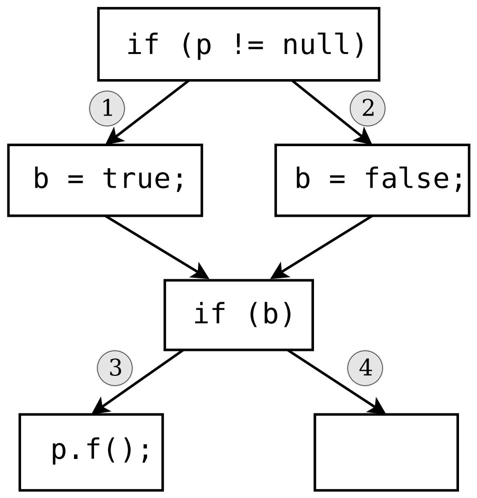


Figure 3.5: Example of an indirect null check

checked against null afterward. This often indicates a real error: if the value really can be null, the comparison should certainly be done before the dereference. (For a general account of using redundant code to find bugs, see [67].)

FindBugs reports a high priority warning for all comparisons of No-Kaboom to null, since this strongly suggests that the value really can be null at the earlier location where it is dereferenced. Other redundant comparisons are considerably less likely to represent real bugs. We use a simple heuristic to determine which of the remaining cases is likely to be worth reporting: if the infeasible branch of a redundant comparison creates a nonempty region of dead code, then we report a medium priority warning. Consider the following code fragment:

```
p = new Object();
q = new Object();
...

if (p != null) { // a defensive null check
  x = p.hashCode();
}
```

```
if (q != null) { // a probable logic error
  y = q.hashCode();
} else {
  doSomethingElse();
}
```

Both null comparisons are redundant, since neither `p` nor `q` can be non-null. The comparison `p != null` is simply defensive, because it does not cause any dead code. The comparison `q != null`, on the other hand, is more likely to indicate programmer confusion: the existence of executable code in the unreachable `else` block suggests a logic error.

In any case, it is important to ensure that the dead code resulting from redundant comparisons does not pollute the results of the analysis. Our analysis marks the frame on the infeasible branch of a redundant comparison as a special "Top" value, which serves as the meet identity element. This effectively makes the dead code invisible to the analysis. In the previous example, this would ensure that `p` and `q` retain the value NonNull, even after their respective null checks.

3.3.8 Assertions

Another form of infeasible control flow which must be considered in a null pointer analysis is exceptions due to failed assertions. Generally, an assertion in Java is a method which takes a boolean argument and throws an exception if the argument is false:

```
// throws exception if p null:
checkAssertion(p != null);
p.f(); // safe
```

We handle these kinds of assertions by simply changing any Null or NSP values to NCP following a call to a method containing the substring “abort”, “assert”, “check”, “error”, or “failed.”

Another form of assertion is a method which throws an exception unconditionally:

```
if (p == null)
  error("p is null"); // throws exception
p.f(); // safe
```

We handle these cases using a simple inter-procedural analysis which identifies methods that throw an exception unconditionally, and at each call site, removing the control edge representing a normal return from those methods.

3.3.9 Finally Blocks

A `finally` block in Java is a region of code associated with a `try` statement which is guaranteed to be executed no matter how control leaves the `try` block. The Java source to bytecode compiler will emit code for a `finally` block either by duplicating it in the generated bytecode, or by emitting a `jsr` subroutine. Two issues arise regarding finally blocks.

The first issue is how to represent `jsr` subroutines in the control flow graph. For FindBugs, we decided to inline them into their call sites. This makes `jsr` and `ret` instructions used to call and return from `jsr` subroutines behave like `goto` instructions as far as the dataflow analysis is concerned. While this could theoretically result in an exponential increase in the size of the resulting control flow graph, we have not observed this in practice.

The second issue is how to handle warnings for code inside `finally` blocks. For most kinds of warnings, including null pointer dereferences, as long as the warning describes a bug feasible for at least one expansion of the block, the warning is valid. However, redundant comparison warnings are only valid if the comparison is redundant for *every* expansion, and is always redundant for the same reason. If the source to bytecode compiler has emitted as a `jsr` subroutine, it is easy to detect the duplication, since each expansion of the subroutine shares the same range of bytecode instructions. However, if the compiler has duplicated the code, the situation is slightly more difficult to detect. In that case, we use the method source line number table to keep track of duplicated code, and only emit redundant comparison warnings if all redundant comparisons for a particular line are in agreement.

3.3.10 Extending the Basic Analysis

In this section, we discuss simple inter-procedural extensions to our basic null pointer analysis.

Unconditionally Dereferenced Parameters

One common source of null pointer errors we have observed in real programs is confusion about whether method parameters may be null. A sure sign that the programmer believes a parameter should be non-null is that it is dereferenced unconditionally. If a caller ever passes a null value for such a parameter, a null pointer exception is guaranteed.

To find methods which unconditionally dereference a parameter, we perform a backward dataflow analysis to compute the set of parameter values guaranteed to be dereferenced on all forward paths. The set at the entry of the method's control flow graph is the set of unconditionally dereferenced parameters. The analysis excludes runtime exception control edges from consideration, unless they are thrown via an explicit `throw` statement. To see why it is necessary to exclude "implicit" runtime exceptions, consider the following method:

```
boolean sameHashCode(Object p, Object q) {
  return p.hashCode() == q.hashCode();
}
```

Although it is possible for the call to `p.hashCode()` to throw a null pointer exception which bypasses the call to `q.hashCode()`, that behavior is not part of the expected behavior of the method. So, our analysis would consider *both* parameters of this method to be dereferenced unconditionally.

Once we have computed the set of methods which unconditionally dereference a parameter, we examine all call sites to find places where a possibly-null value (Null or NSP in the lattice) is passed to a method which might unconditionally dereference it. An obvious difficulty in Java is determining which methods might be called at polymorphic call sites. Our analysis conservatively assumes that unless the type of the receiver object is known exactly it could be any concrete subtype. Some methods, such as `Object.equals()`, have a large number of potential target methods, and could therefore generate a large number of false warnings. To reduce the effect of these false positives, we issue a high or medium priority warning only

Annotation	Context	How checked
`@NonNull`	Parameter	Callers must not pass Null or NSP
	Return value	Method must not return Null or NSP
`@PossiblyNull`	Parameter	Method must not dereference unconditionally
	Return value	Callers must not dereference unconditionally

Table 3.3: Supported annotations for method parameters and return values

for cases where there is a single known target, or when all known targets dereference the parameter unconditionally.

Parameter and Return Value Annotations

A recurring issue in program analysis to find bugs is trying to deduce the *expected* behavior of some piece of code. In null pointer analysis, it is often unclear where the blame lies when an inconsistency is detected. For example, when an unconditional parameter dereference results in a null pointer exception, should we blame the caller or the callee?

Specifications are a simple way for the programmer to make the job of the analysis easier by explicitly marking which values must not be null and which may be null. (LCLint [25] and CQual [31] are other examples of static analysis tools that leverage lightweight annotations to find bugs.) Our analysis can use two kinds of specifications on method parameters and return values: `@NonNull`, which indicates that a value must not be null, and `@PossiblyNull`, which indicates that a value might be null. These specifications are conveyed using Java source annotations [42]. Table 3.3 lists a summary of how these annotations are checked.

`@NonNull` and `@PossiblyNull` annotations are inherited by subclasses. They may be relaxed by subclasses (for example, by changing a `@NonNull` annotation to `@PossiblyNull` on a parameter), but not strengthened. Because annotations are a contract applying to *all* subclasses, they do not suffer imprecision due to the difficulty of resolving target methods at polymorphic call sites. However, they do impose an additional restriction on method callers not to pass a null value or unconditionally dereference a return value, even when they know that to be safe in the calling context. The is a potential source of false warnings. The results of using specifications in student programming projects, including a discussion of false warning rates, are discussed in Chapter 5.

3.4 Wrong Type in Equals Comparison

The contract for the `equals(Object)` method requires that it return `false` if the object passed to it has a different runtime type than that of the receiver object. For this reason, any call to `equals()` which passes an object of a type which cannot possibly match the type of the receiver object is a probable bug, for one of the following reasons:

1. The caller passed the wrong object
2. The class defining the `equals()` method called is implemented incorrectly, and may return `true` for objects of different runtime types

The detector for this pattern is based on a forward dataflow analysis to infer the types of all values of parameters, local variables, and stack operands at each location in an analyzed method. This analysis is similar to the algorithm for verification of

Java class files described in the Java Virtual Machine Specification [50]. The values generated by the analysis represent an upper bound on the real runtime types: the actual runtime type of a value may be a subtype of the type inferred by the analysis. Two kinds of warnings may be produced for a call to `equals()` using receiver type A and argument type B.

If both A and B are class types, FindBugs generates a warning if neither $A \subseteq B$ nor $B \subseteq A$. Because Java classes use single inheritance, this implies that the objects compared by the call to `equals()` cannot possibly have the same runtime type.

If either A and B are interface types, then the detector computes the intersection of all known subtypes of A and B. If no concrete classes are found in the intersection, the detector issues a warning. Note that checking for the existence of subtypes is sound only if FindBugs is aware of all classes that might be available at runtime. Due to language features such as dynamic class loading, this is not always a valid assumption. However, as long as FindBugs has access to the complete class hierarchy that will be used at runtime, this is a reasonable assumption to make.

3.5 Infinite Recursive Loop

Consider the code in Figure 3.6, which is taken from a student programming assignment. The student was obviously confused about the distinction between allocating a new instance of a class using the `new` operator and initializing a new instance in a constructor. At runtime, this code will enter an infinite recursive loop.

```
class WebSpider {
    /** Construct a new WebSpider */
    public WebSpider(boolean isDFS, int limit) {
        WebSpider spider = new WebSpider(isDFS, limit);
    }
    ...
}
```

Figure 3.6: An infinite recursive loop in a student programming project

It is easy to understand why a novice programmer would make this particular mistake. However, could errors like this occur in production code? On the surface, it seems unlikely. No experienced programmer would confuse object allocation and initialization, and any test suite achieving 100% method coverage would be guaranteed to find problems like this one. However, bugs of this type can and do occur in production software. They can occur for several reasons, including:

- Confusing a method with an identically-named field (Figure 3.7)
- Casting a method argument with the intention of dispatching to an overloaded method which does not actually exist (Figure 3.8)
- Failure to delegate, either to another object or to a superclass method (Figure 3.9)

The detector for this bug pattern works by looking for recursive calls which either

1. postdominate the entry point of the method, or
2. pass the method parameters as arguments to a recursive call without first reading and writing at least one field

```
// In java.lang.annotation.AnnotationTypeMismatchException

private final String foundType;

...

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

Figure 3.7: Infinite recursive loop in J2SE version 1.5.0 due to confusing identically-named field and method

```
 // In org.jboss.util.MuInteger, line 229

public int compareTo(int other) {
   return (value < other) ? -1 : (value == other) ? 0 : 1;
}
...
public int compareTo(Object obj) {
   return compareTo((MuInteger)obj); // meant to call compareTo(int)?
}
```

Figure 3.8: Infinite recursive loop in JBoss 4.0.2 due to an attempt to dispatch to an nonexistent overloaded method variant

```
// In org.eclipse.pde.internal.core.plugin.Extensions
// line 27

void load(Extensions srcPluginBase) {
    range= srcPluginBase.range;
    super.load(srcPluginBase);        // delegation
    valid = hasRequiredAttributes();
}
public void load(IPluginBase srcPluginBase) {
    this.load(srcPluginBase);         // failure to delegate?
}
```

Figure 3.9: Infinite recursive loop in Eclipse 3.0 due to failure to delegate

One difficulty in detecting this pattern is determining when the caller and callee are really the same method. For non-virtual method calls, this is simple, since there is only one possible target. For virtual method calls, we issue a warning only in the following cases, where we have high confidence that the call is recursive:

- the callee is known to have the same runtime type as the caller, or
- the caller and callee are dispatched to the same object

Note that the latter condition does not strictly guarantee that the call is recursive, since the object might be an instance of a subclass which overrides the caller. However, we doubt that this case could arise in any competently written program.

3.6 Bad Cast

One of the fundamental techniques used in object-oriented programming is the creation of class hierarchies, in which many separate implementations of some base class or interface exist simultaneously. As long as the methods defined by the base class are sufficiently general, most clients will be able to operate on instances without needing to know their exact subtype. However, in some cases it is necessary for clients to dynamically convert a reference from a base class to a subclass by means of a downcast. Such casts are checked at runtime, and throw a `ClassCastException` if the object is not an instance of the subclass named by the cast.

While the Java source to bytecode compiler prevents some infeasible casts, it allows others to slip through. One type of infeasible downcast not caught by the compiler may happen as the result of an `instanceof` check. The check proves that

// In java.awt.Font.initializeFont(Hashtable)

```
obj = attributes.get(TextAttribute.WIDTH);
if (obj instanceof Integer) {
    width = ((Float)obj).floatValue();
```

Figure 3.10: An infeasible downcast in the released version of the J2SE 1.5.0 runtime library

the tested value is an instance of the named class (or a subclass). However, the compile-time type of the checked variable is not changed, allowing the value to be cast to a provably unrelated type. While we have seen this kind of error occur most often in student programming projects, it can also occur in production software. An example occurring in the released version of the J2SE 1.5.0 core runtime library is shown in Figure 3.10.

The detector for this pattern looks for the following types of errors:

- Infeasible casts
- Infeasible `instanceof` checks
- Upcasts (casts to a supertype)
- Casts of a reference to generic collection interface (e.g., `Map` or `Set`) to an abstract or concrete collection

The first two items (infeasible casts and `instanceof` checks) are almost certainly unintentional and may indicate a serious logic error.

The last two items do not necessarily indicate an error, but often indicate a novice programming mistake. An upcast (cast to a supertype) serves no useful

```
/**
 * Create a new WebPage
 *
 * @param u -
 *            the URL of this WebPage
 */
public WebPage(URL u) {
    Object obj = (Object)u;
    this.webpage = (WebPage)obj;
    this.visited = false;
}
```

Figure 3.11: An infeasible dynamic cast in a student programming project

purpose, since an instance of a subtype can always be used in any context where a supertype is expected. We have sometimes observed students casting a reference to the base `Object` type in order to silence the compiler error which would result from a subsequent cast to an unrelated type. An example is shown in figure 3.11. Casts to abstract or concrete collection classes represent another novice programmer mistake we have observed in student code. Such casts are at best unnecessary, since the collection interface classes contain all of the methods required to operate on instances of collections, and are prone to failing if attempted on a collection object of an unexpected type.

Like the detector for suspicious `equals()` comparisons, the detector for bad casts uses dataflow analysis to compute types for local variables and stack operands. Where appropriate, successful `instanceof` checks are used to make the checked type more precise. Determining which `instanceof` and `checkcast` instructions are guaranteed to either fail or succeed at runtime involves checking the class hierarchy. For example, a `checkcast` instruction is guaranteed to fail if neither class is a subtype of the other. Making this determination requires inspection of the class hierarchy. We assume that all classes which could be used at runtime will be available when the static analysis is performed. Although this is not a sound assumption, it is reasonable in most cases.

3.7 Read Return Should Be Checked

The `java.io.InputStream` class has two `read()` methods which read multiple bytes into a buffer. Because the number of bytes requested may be greater than the number of bytes available, these methods return an integer value indicating how many bytes were actually read.

Programmers sometimes incorrectly assume that these methods always return the requested number of bytes. However, some input streams (e.g., sockets) can return short reads. If the return value from `read()` is ignored, the program may read uninitialized or stale values from the buffer and also lose its place in the input stream.

One way to implement this detector would be to use dataflow analysis to determine whether or not the location where the return value of a call to `read()` is stored is ever used by another instruction. However, a simpler analysis technique works well in practice. The detector for this bug pattern is implemented as a simple linear scan over the bytecode. If a call to a `read()` method taking a byte array is followed immediately by a `POP` bytecode, we emit a warning. As a refinement, if a call to `InputStream.available()` (or one of several other methods which check

```
// GNU Classpath 0.15
// java/util/SimpleTimeZone.java, line 1012

int length = input.readInt();
byte[] byteArray = new byte[length];
input.read(byteArray, 0, length);
if (length >= 4)
  {
    ...
```

Figure 3.12: An example of ignored read return value

the availability of data in the input stream) is seen, we inhibit the emission of any warnings for the next 70 instructions[2]. This eliminates some false positives where the caller knows that the input stream has a certain amount of data available.

In addition to finding calls to `read()` where the return value is ignored, the detector also finds calls to `skip()` where the return value is ignored. Much like `read()`, `skip()` is not guaranteed to skip the exact number of bytes requested.

An example of a bug found by this detector is shown in Figure 3.12. This code occurs in the class's `readObject()` method, which deserializes an instance of the object from a stream. In the example, the number of bytes read is not checked. If the call returns fewer bytes than requested, the object will not be deserialized correctly, and the stream will be out of sync (preventing other objects from being deserialized).

[2]We arrived the value 70 by analyzing the classes in Sun's core Java libraries; there was only a single instance of a call to read() more than 70 instructions past a call to a method which checks the availability of input data.

3.8 Return Value Should Be Checked

This bug pattern is a generalization of the Read Return Should Be Checked pattern. The standard Java libraries define a number of methods whose return value must be checked to be used correctly.

A common form of this pattern is calling a method on an immutable object without checking the result value. For example, once constructed, Java `String` objects do not change value. Methods that transform a `String` value do so by returning a new object. This is often a source of confusion for programmers used to other languages (such as C++) where string objects are mutable, leading to mistakes where the return value of a method call on an immutable object is ignored.

The implementation of the detector for this bug pattern is very similar to that of the Read Return detector. We look for calls to any memory of a certain set of methods followed immediately by `POP` or `POP2` bytecodes. The set of methods we look for includes

- Any `String` method returning a `String`
- `StringBuffer.toString()`
- Any method of `InetAddress`, `BigInteger`, or `BigDecimal`
- `MessageDigest.digest(byte[])`
- The constructor for any subclass of `Thread` or `Throwable`.

This detector is a good example of how bug checkers can help dispel common misconceptions about API semantics.

3.9 Double Checked Locking

Lazy initialization is a common performance optimization used to create singleton objects only as they are needed. In a multithreaded program, some form of synchronization is needed to ensure that the singleton object is created only once, and that the object is always fully initialized before it is used.

A common idiom for lazy initialization of singleton objects is *double checked locking*. Synchronization is performed only if the object has not yet been created, as shown in the following example:

```
static SomeClass field;

static SomeClass createSingleton() {
  if (field == null) {
    synchronized (lock) {
      if (field == null) {
        SomeClass obj = new SomeClass();
        // ...initialize obj...
        field = obj;
      }
    }
  }
  return field;
}
```

The intent of double checked locking is that the overhead of lock acquisition is only incurred if the singleton object is observed as not having been created yet.

Unfortunately, this form of double checked locking is not correct [56]. Although the idiom guarantees that the object is created only once, the Java memory model does not guarantee that the threads that see a non-null field value but do not acquire the lock will see all of the writes used to initialize the object. For example, the JIT compiler may inline the call to the constructor and reorder some of the writes initializing the object so that they occur after the write to the field storing the object instance.

Double checked locking is a good illustration of the gulf between how one might think multithreaded code should behave and the behaviors actually allowed by the language. Most programmers can easily understand how synchronization can be used to guarantee the atomicity of a sequence of operations. However, it is much harder to understand the subtle interaction of compiler and processor optimizations. Memory model issues are challenging even for experts. Double checked locking has been advocated in a number of books and articles, showing that even experts do not always understand the consequences of omitting proper synchronization.

Under the revised Java Memory Model [40], it is possible to fix instances of double checked locking by making the field volatile. The volatile qualifier causes the compiler to insert the necessary optimization and memory barriers needed to ensure that all threads will see a completely initialized object, even if they do not acquire the lock.

Our detector for this bug pattern looks for sequences of bytecode containing an `ifnull` instruction, followed by a `monitorenter` instruction, followed by another `ifnull`, with some small number of intermediate instructions occurring between these sentinels. This implementation catches many instances of double checked locking, with a low false positive rate. We have experimented with more sophisticated detectors for this pattern, but we found that they were not significantly more effective or accurate.

An example of incorrect double checked locking found in JBoss-4.0.0DR3 is

```
// org/jboss/net/axis/server/JBossAuthenticationHandler.java
// Line 178

public void invoke(MessageContext msgContext) throws AxisFault {
   // double check does not work on multiple processors, unfortunately
   if (!isInitialised) {
      synchronized (this) {
         if (!isInitialised) {
            initialise();
         }
      }
   }
   ...
```

Figure 3.13: A double checked locking bug in JBoss-4.0.0DR3

shown in Figure 3.13. This particular example is interesting because not only does the comment indicate that the programmer was aware the idiom is incorrect, the `initialise()` method (not shown) writes a true value to the `isInitialised` field before the object has been completely initialized, meaning that the code would not be correct on a *single* processor system, even if the Java memory model were sequentially consistent.

3.10 Inconsistent Synchronization

When mutable state is accessed by multiple threads, it generally needs to be protected by synchronization. A very common technique in Java is to protect the mutable fields of an object by acquiring a lock on the object itself. A method may be defined with the `synchronized` keyword, in which case a lock on the receiver object is obtained for the scope of the method. Or, if finer grained synchronization is desired, a `synchronized(this)` block may be used to acquire the lock within a

```
// java.lang, StringBuffer.java, line 825

public int lastIndexOf(String str)
{
  return lastIndexOf(str, count - str.count);
}
```

Figure 3.14: An atomicity bug in GNU Classpath 0.08

block scope. Classes whose instances are intended to be thread safe should generally only access shared fields while the instance lock is held. Unsynchronized field accesses often are race conditions that can lead to incorrect behavior at runtime. We refer to unsynchronized accesses in classes intended to be thread safe as *inconsistent synchronization*. An example is shown in Figure 3.14. The `count` field is read without synchronization and then passed to a synchronized method. Because the value may be out of date, an `ArrayIndexOutOfBoundsException` exception can result.

To detect inconsistent synchronization, the FindBugs tool tracks the scope of locked objects[3]. For every instance field access, the tool records whether or not a lock is held on the instance through which the field is accessed. Fields which are not consistently locked are reported as potential bugs.

We use a variety of heuristics to reduce false positives. Field accesses in object lifecycle methods, such as constructors and finalizers, are ignored, because it is unlikely that the object is visible to multiple threads in those methods. We ignore

[3]The analysis is intra-procedural, with the addition that calls to non-public methods within a class are analyzed, and non-public methods called only from locked contexts are considered to be synchronized as well.

public fields, on the assumption users must be responsible for guarding synchronization of such fields. Volatile fields are also ignored, because under the Java memory model [40], reads and writes of volatile fields can be used to enforce visibility and ordering guarantees between threads. Similarly, final fields are ignored, since they are largely thread safe (the only exception being cases where objects are made visible to other threads before construction is complete).

3.10.1 Limitations of the Inconsistent Synchronization Detector

While it is effective at finding many concurrency bugs, the inconsistent synchronization detector has some important limitations. First, programs that are free of race conditions may still have atomicity bugs. A naïve approach to synchronization is to make every method of a class synchronized. However, successive calls to methods in such a class will not occur atomically unless a lock on the receiver object is explicitly held in a scope surrounding both calls. Another limitation is that the detector only works when shared fields are synchronized by locking the object instance. Although this is a common technique in Java, it is also common to use explicit lock objects. Modifying the detector to handle arbitrary lock objects would require more sophisticated analysis, including some form of heap analysis.

3.11 Wait Not in Loop

Java monitors support notify and wait operations to allow threads to wait for a condition related to the state of a shared object. For example, in a multithreaded blocking queue class, the `dequeue()` method would wait for the queue to become nonempty, and the `enqueue()` method would perform a notify to wake up any threads waiting for the queue to become nonempty.

Often, a single Java monitor is used for multiple conditions. For such classes, the correct idiom is to surround the call to `wait()` with a loop which repeatedly checks the condition. Without the loop, the thread would not know whether the condition is actually true when the call to `wait()` returns.

The condition can fail to be true for several reasons:

- The monitor is being used for multiple conditions, so the condition set by the notifying thread may not be the one the waiting thread is waiting for.
- In between the notification and the return from wait, another thread obtains the lock and changes the condition. For example, a thread might be waiting for a queue to become non-empty. A thread inserts a new element into the queue and notifies the waiting thread. Before the waiting thread acquires the lock, another thread removes the element from the queue.
- The specification for the `wait()` method allows it to spuriously return for no reason. This can arise due to special handling needed for the interaction of interrupts and waiting, and because underlying operating system synchronization primitives used by the JVM, such as POSIX threads [38], allow spurious wakeups.

The detector for this bug pattern examines bytecode for a call to `wait()` which is not in close proximity to the target of a backwards branch (i.e., a loop head).

```
if (!book.isReady()) {
  DebugInfo.println("book not ready");

  synchronized (book) {
    DebugInfo.println("waiting for book");
    book.wait();
  }
  ...
}
```

Figure 3.15: An unconditional wait bug in an early version of the International Children's Digital Library

3.12 Unconditional Wait

The Unconditional Wait pattern is a special case of Wait Not In Loop. In this bug pattern, a wait is performed immediately (and unconditionally) upon entering a synchronized block. Often, this indicates that the programmer did not include the test for the waited-for condition as part of the scope of the lock, which could lead to a missed notification. An example of this bug pattern is shown in Figure 3.15. If the book object becomes ready after `isReady()` is called and before the lock is acquired, the notification could be missed and the thread could block forever.

3.13 Two Lock Wait

Signaling between threads is usually handled by using the `wait()`, `notify()`, and `notifyAll()` methods. When `wait()` is invoked, the thread invoking wait must hold a lock on the object on which wait is invoked, and the lock on that object is released while the thread is waiting. However, locks on other objects are not released. This can cause poor performance, and can cause deadlock if they thread that is trying to perform a notify needs to acquire that lock.

The detector for this pattern performs an intra-procedural analysis to find the scope of locks, and emits a warning whenever a method holds multiple locks when wait is invoked. While occurrences of this pattern are not always bugs, they are worth inspecting closely to ensure that no deadlocks can arise.

3.14 Dead Local Store

In general, there is no reason to store a value in a local variable unless that value will be used later. Stores whose values are never loaded are *dead stores*. They may be found by performing a backward dataflow analysis to find, at the location of a store to a local variable, whether its value will be used on any forward path.

A dead store is not necessarily a bug. From our inspection of dead stores occurring in production code, we have found that most occurrences do not indicate an actual bug. However, there are two cases which are at least somewhat likely to be errors:

1. A method parameter immediately overwritten upon method entry

2. An integer increment overwritten by another store

These cases are considered correctness issues. We classify all other dead local stores as style issues.

Dead local stores are interesting because they strongly suggest that a computation is being performed, but its result is never used. These cases are worth inspecting closely because they may reveal a logic error, or may identify code which has become convoluted through modification and is in need of restructuring.

Chapter 4

Experiences Analyzing Production Software

Assessing the success of static bug-finding tools is challenging. In general, we can never know how many real bugs exist in a software artifact. In the absence of formal requirements and specifications, it is impossible to precisely specify what may be considered a bug in a particular software artifact. In addition, not every bug is equally important. Some bugs may prevent the correct functioning of the software, while other may be mere annoyances. As noted in the Chapter 1, there are very real costs associated with fixing a bug. For this reason, economic concerns may result in some defects being left unfixed.

While it would be nice for a static bug checker to guarantee that software is free from (some kinds of) bugs, the ability to find at least some real bugs is useful. If a tool allows a developer to find a bug that she otherwise would not have found, that is a positive outcome. Thus, one measure we can use to judge the effectiveness of a static bug checker is how many of the bugs it finds are judged by developers to be worth fixing.

A less direct but perhaps more important measure of success for any software engineering tool is the extent to which it is adopted by real software developers. Regular use of a static analysis tool strongly suggests that the tool finds issues which are judged to be important by developers. Because we have made FindBugs available as open-source software, we have had ample opportunity to communicate with real software developers and to learn of their experiences using the tool.

In this chapter, we investigate the effectiveness of FindBugs at finding bugs in production software. Section 4.1 describes a study in which we inspected warnings produced by FindBugs on several Java applications and libraries in order to determine the false positive rate for selected bug detectors. Section 4.2 examines the inconsistent synchronization detector in more detail, and investigates the general question of whether data races in Java programs are accidental or deliberate. Section 4.3 lists bugs we have reported directly to software maintainers which have been confirmed as worth fixing. Sections 4.4 and 4.5 discuss adoption of FindBugs by other organizations.

4.1 Empirical Evaluation of Production Applications and Libraries

It is easy to apply our bug pattern detectors to software. However, evaluating whether the warnings generated correspond to errors that warrant fixing is a manual, time-consuming and subjective process. We have applied our best effort to fairly classify many of the warnings generated by FindBugs for several real applications and libraries. Each warning is classified in one of the following ways:

- Some bug pattern detectors are very accurate, but determining whether the situation detected warrants a fix is a judgment call. For example, we can easily and accurately tell whether a class contains a static field that can be modified by untrusted code. However, human judgment is needed to determine whether that class will ever run in an environment where it can be accessed by untrusted code. We did not try to judge whether the results of such detectors warrant fixing, but simply report the warnings generated.

- Some the bug detectors admit false positives, and report warnings in cases where the situation described by the warning does not, in fact occur. Such warnings are classified as *false positives.*

- The warning may reflect a violation of good programming practice but be unlikely to cause problems in practice. For example, many incorrect synchronization warnings correspond to data races that are real but highly unlikely to cause problems in practice. Such warnings are classified as *mostly harmless* bugs.

- And then there are the cases where the warning is accurate and in our judgment reflects a serious bug that warrants fixing. Such warnings are classified as *serious.*

In this section, we report on our manual evaluation of the warnings produced by FindBugs version 0.9.2[1] for several warning categories on the following applications and libraries:

- GNU Classpath, version 0.08

- rt.jar from Sun JDK 1.5.0, build 59

[1]We used the `-effort:max` command line option to enable extra analysis, such as pruning infeasible exception paths, which improve the accuracy of some bug detectors.

	classpath-0.08				rt.jar 1.5.0 build 59			
			mostly				mostly	
code	warnings	serious	harmless	false pos	warnings	serious	harmless	false pos
BC	0	—	—	—	9	100%	0%	0%
DC	1	100%	0%	0%	88	100%	0%	0%
EC	2	100%	0%	0%	9	100%	0%	0%
IL	2	100%	0%	0%	4	100%	0%	0%
IS2	40	62%	17%	20%	109	45%	45%	8%
NP	14	78%	0%	21%	67	64%	0%	35%
NS	0	—	—	—	12	25%	66%	8%
OS	6	50%	0%	50%	13	15%	0%	84%
RCN	3	100%	0%	0%	40	60%	2%	37%
RR	9	100%	0%	0%	12	91%	0%	8%
RV	11	45%	0%	54%	8	62%	0%	37%
UR	3	66%	0%	33%	4	100%	0%	0%
UW	2	0%	0%	100%	6	50%	0%	50%
Wa	3	0%	0%	100%	8	37%	0%	62%

Table 4.1: False positive rates for selected bug pattern detectors on GNU Classpath and Sun J2SE libraries

	eclipse-3.0				drjava-stable-20040326			
			mostly				mostly	
code	warnings	serious	harmless	false pos	warnings	serious	harmless	false pos
DC	88	100%	0%	0%	0	—	—	—
EC	15	73%	0%	26%	0	—	—	—
IL	3	66%	0%	33%	0	—	—	—
IS2	61	67%	21%	11%	2	0%	0%	100%
NP	107	65%	4%	29%	0	—	—	—
NS	14	78%	21%	0%	0	—	—	—
OS	26	46%	0%	53%	4	100%	0%	0%
RCN	61	49%	29%	21%	0	—	—	—
RR	39	38%	0%	61%	0	—	—	—
RV	13	76%	0%	23%	0	—	—	—
TLW	2	0%	0%	100%	1	0%	0%	100%
UCF	1	100%	0%	0%	0	—	—	—
UR	4	50%	50%	0%	1	0%	100%	0%
UW	7	28%	0%	71%	3	100%	0%	0%
Wa	12	25%	0%	75%	3	100%	0%	0%

Table 4.2: False positive rates for selected bug pattern detectors on Eclipse and DrJava

	jboss-4.0.0RC1				jedit-4.2pre15			
			mostly				mostly	
code	warnings	serious	harmless	false pos	warnings	serious	harmless	false pos
DC	3	100%	0%	0%	0	—	—	—
EC	4	100%	0%	0%	0	—	—	—
IS2	25	28%	24%	48%	3	33%	33%	33%
NP	23	82%	4%	13%	3	33%	0%	66%
NS	0	—	—	—	1	0%	0%	100%
OS	10	60%	0%	40%	4	75%	0%	25%
RCN	10	40%	10%	50%	1	0%	0%	100%
RR	8	62%	25%	12%	5	100%	0%	0%
RV	3	33%	0%	66%	0	—	—	—
UR	2	50%	0%	50%	1	0%	0%	100%
UW	4	50%	25%	25%	1	100%	0%	0%
Wa	4	0%	0%	100%	2	50%	0%	50%

Table 4.3: False positive rates for selected bug pattern detectors on JBoss and jEdit

- Eclipse, version 3.0
- DrJava, version stable-20040326
- JBoss, version 4.0.0RC1
- jEdit, version 4.2pre15

GNU Classpath [33] is an open source implementation of the core Java runtime libraries. rt.jar is Sun's implementation of the APIs for J2SE [39]. Eclipse [19] and DrJava [18] are popular open source Java integrated development environments. JBoss [43] is a popular Java application server. jEdit [44] is a programmer's text editor. All of the applications and libraries we used in our experiments, with the possible exception of GNU Classpath, are commercial-grade software products with large user communities.

None of the analyses implemented in FindBugs is particularly expensive to perform. On a 1.8 GHz Pentium 4 Xeon system with 1 GB of memory, FindBugs

Application	Eq	HE	MS	Se	DE	CN
classpath-0.08	2	14	39	14	2	27
rt.jar 1.5.0 build 59	9	55	259	207	89	73
eclipse-3.0	3	170	1,000	49	23	20
drjava-stable-20040326		9	45	37	5	4
jboss-4.0.0RC1	1	18	227	44	10	22
jedit-4.2pre15		6	53	5	1	1

Table 4.4: Warning counts for selected other detectors

took no more than 90 minutes to run all of the bug pattern detectors on any of the applications we analyzed. To give a sense of the raw speed of the analysis, the version of rt.jar we analyzed contains 13,083 classes, is about 40 MB in size, and required approximately one hour to analyze. The maximum amount of memory required to perform the analyses was approximately 700 MB. We have not attempted to tune FindBugs for performance or memory consumption.

4.1.1 Empirical Evaluation

Tables 4.1 through 4.3 shows our evaluation of the accuracy of the detectors for which there are clear criteria for deciding whether or not the reports represent real bugs. All of the detectors evaluated found at least one bug pattern occurrence which we classified as a real bug.

It is interesting to note that the accuracy of the detectors varied significantly by application. For example, the detector for the RR pattern was very accurate for most applications, but was less successful in finding genuine bugs in Eclipse. The reason is that most of the warnings in Eclipse were for uses of a custom input stream class for which the `read()` methods are guaranteed to return the number of bytes requested.

Our target for bug detectors admitting false positives was that at least 50% of reported bugs should be genuine. In general, we were fairly close to meeting this target. Only the UW and Wa detectors were significantly less accurate. However, given the small number of warnings they produced and the potential difficulty of debugging timing-related thread bugs, we feel that they performed adequately. We also found that these detectors were much more successful in finding errors in code written in undergraduate courses, which illustrates the usefulness of tools in steering novices towards correct use of difficult language features and APIs.

It is worth emphasizing that all of these applications and libraries (with the possible exception of GNU Classpath) have been extensively tested, and are used in production environments. The fact we were able to uncover so many bugs in these applications makes a very strong argument for the need for automatic bug checking tools in the development process. Static analysis tools do not require test cases, and do not have the kind of preconceptions about what code is "supposed" to do that human observers have. For these reasons, they usefully complement traditional quality assurance practices.

Ultimately, no technique short of a formal correctness proof will eliminate every bug. From our evaluation of FindBugs on real applications, we conclude that simple static tools like bug pattern detectors find an important class of bugs that would otherwise go undetected.

Application	KLOC	FindBugs	PMD
classpath-0.08	457	724	4,079
rt.jar 1.5.0 build 59	1,183	3,314	17,133
eclipse-3.0	2,237	4,227	25,227
drjava-stable-20040326	109	293	645
jboss-4.0.0RC1	989	1,188	13,521
jedit-4.2pre15	140	191	1,113

Table 4.5: Application sizes and total number of warnings generated by FindBugs and PMD

4.1.2 Other Detectors

Table 4.4 lists results for some of our bug detectors for which we did not perform manual examinations. These detectors are fairly to extremely accurate at detecting whether software exhibits a particular feature (such as violating the hashcode/equals contract, or having static fields that could be mutated by untrusted code). However, it is sometimes a difficult and very subjective judgment as to whether or not the reported feature is a bug that warrants fixing in each particular instance. We will simply note that in many cases, these reports represent instances of poor style or design.

4.1.3 Comparison of FindBugs with PMD

In Table 4.5, we list the number of lines of source code for each benchmark application[2], the total number of high and medium priority warnings generated by FindBugs version 0.9.2, and the number of warnings generated by PMD version 1.9 [55][3]. In general, FindBugs produces a much lower number of warnings than

[2]Note that the figure for rt.jar is low because not all of its source code is available in the standard public distribution.

[3]For PMD, we used the suggested rule sets: basic, unusedcode, imports, and favorites.

PMD when used in the default configuration. Undoubtedly, PMD finds a significant number of bugs in the benchmark applications: however, they are hidden in the sheer volume of output produced.

We do not claim this comparison shows that FindBugs is “better” than PMD, or vice versa. Rather, the two tools focus on different aspects of software quality. Tools like PMD are extremely valuable on enforcing consistent coding style guidelines, and making code easier to understand by developers. Tools like FindBugs help uncover errors, while largely ignoring style issues. Therefore, PMD and FindBugs complement each other, and neither is a good substitute for the other. (A paper by Rutar et. al. [58] compares a number of static analysis tools for Java, including FindBugs and PMD.)

4.1.4 How Many Bugs Remain?

Static analysis will not find every bug. However, we would like to have some idea of what percentage of all bugs are found by static analysis.

Citing recent results [16] from the Software Engineering Institute, a report by the Security Across the Software Development Lifecycle Task Force [66] estimates the number of defects in typical production software at 1–7 per 1,000 lines of code. For build 59 of the J2SE runtime library, we manually confirmed 256 of the warnings emitted by FindBugs accurately identify serious defects.

Since we do not have the full source code for the library, we estimated the number of lines of source code based on inspection of the compiled class files. For class files with line number information, we can directly measure the number of lines in the original source files. We can use this information to compute the ratio of executable code size (bytecode instructions) to source lines. This ratio can then be used to estimate the number of lines of code in classes without source line information, based on the size of the bytecode. Our estimate for total lines of code in the library was 5,733,596. Assuming 1–7 defects per 1,000 lines of code, there should be somewhere between 5,700 to 40,100 total bugs in the library. Based on this estimate, FindBugs has found somewhere between .6% and 4.4% of all remaining bugs, and possibly more, since we did not inspect and classify all of the warnings.

There are too many sources of uncertainty for this estimate to be trusted. We really don't know how many actual defects remain in this library. However, it seems likely that FindBugs has made a significant dent in finding the remaining bugs. In Chapter 5, we present results from a study of student programming projects in which we can make a more confident estimate of the number of bugs not found by static analysis.

4.2 Evaluation of Data Races in Production Applications

Data races are a very common form of error in multithreaded programs, and one of the most difficult to find by testing. A bug caused by a data race may happen so infrequently that it takes weeks to find. In addition, adding tracing code to help find the error may alter the timing of the program just enough that the bug no longer occurs. For these reasons, finding data race bugs before they enter a production system is extremely important.

In this section, we evaluate inconsistent synchronization warnings produced by FindBugs for several multithreaded applications and libraries. In particular, we examine the assumption that data races are likely to occur because programmers occasionally forget to use synchronization.

4.2.1 Determining Which Fields Should Be Protected by Locks

Recall that the inconsistent synchronization detector works by inferring which objects are meant to be thread-safe by looking for occurrences of the common Java idiom where an object is locked in order to guard accesses to its fields by multiple threads.

Initially, we assumed that shared fields of objects intended by programmers to be thread-safe would generally be synchronized consistently, and that bugs would usually be the result of oversight by the programmer. For example, a programmer might add a public method to a thread-safe class, but forget to make the method synchronized. Under this assumption, we used the frequency of unsynchronized accesses to prioritize the warnings generated for inconsistently synchronized fields. We hypothesized that fields with 25% or fewer statically unsynchronized accesses (but at least one unsynchronized access) were probably intended to be synchronized, so we assigned those cases medium or high priority. We assumed that fields with 25-50% statically unsynchronized accesses were probably only incidentally synchronized, and not intentionally synchronized. Therefore, we assigned those cases low priority.

To evaluate the appropriateness of our ranking heuristic, we manually categorized inconsistent synchronization warnings for several applications and libraries. We used the same classification scheme as the study described in Section 4.1, classifying inconsistent synchronization warnings as Serious, Mostly Harmless, or False.

Our decisions were based on manual inspection of the of the code identified by each warning. While our judgment is fallible, we tried to err on the side of classifying warnings as false or harmless if we could not see a scenario that would lead to unintended behavior.

We then studied the number of serious, harmless, and false warnings that would be reported by the tool for varying cutoff values for the minimum percentage of unsynchronized field accesses. For example, for a cutoff value of 75%, only fields whose accesses were synchronized at least 75% of the time would be reported. By graphing the number of warnings in these categories, we were able to evaluate the validity of the hypothesis that most of the serious bugs would have a high percentage of synchronized accesses. We combined the harmless and false categories because together they represent the set of warnings we believed would not be of interest to developers. Figure 4.1 shows these graphs for several applications and libraries. The applications are two implementations of the J2SE core libraries (rt-1.5-b42 and classpath-0.08), a open source Java application server (jboss-4.0.0DR3), and an open source CORBA ORB (jacorb-2.1). Our hypothesis that fields with lower (but nonzero) percentages of unsynchronized accesses would be more likely to be errors was found to be incorrect.

Contrary to our expectations, the graphs show that the likelihood of an inconsistently synchronized field being a serious bug was not strongly related to the

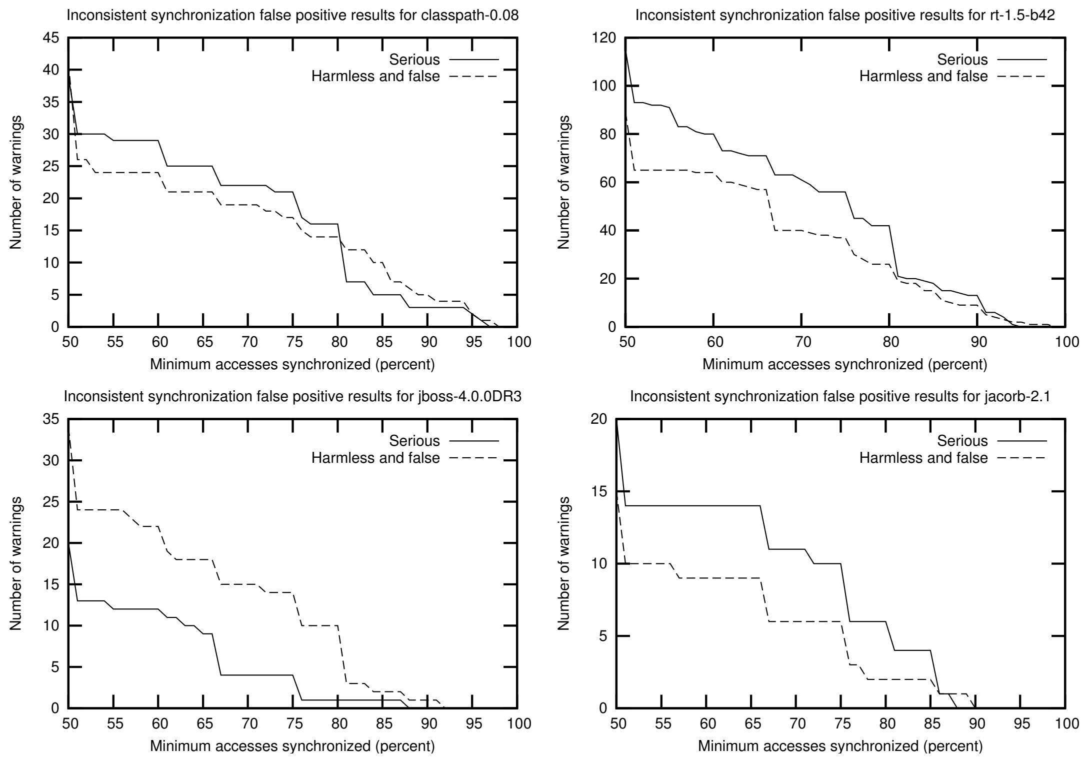


Figure 4.1: Serious bugs and false and harmless warnings for varying values for minimum percentage of synchronized accesses

percentage of synchronized accesses for the range of cutoff values we examined. In other words, the inconsistent synchronization bugs we found were not generally the result of the programmer simply forgetting to synchronize a particular field access or method. Instead, we found that the lack of synchronization was almost always intentional—the programmer had deliberately used a race condition to communicate between threads. The data suggests that we should try even lower cutoff values (below 50%), since many genuine bugs were found for fields synchronized only 50% of the time. The message to take away here is that lack of synchronization is not exceptional; for many classes, it is the norm.

4.3 Confirmed Bugs

We have filed official bug reports for some of the bugs found by FindBugs in the core J2SE runtime library [39]. While direct adoption by developers is perhaps a better indication of the usefulness of a static analysis tool, filing bug reports is also a useful validation.

The reports we have filed which have been acknowledged to be bugs by Sun include:

- Bug 6179014, an infinite recursive loop (shown in Figure 3.7)
- Bugs 6179614, 6179620, 6252367, 6265746, 6266406, 6282820, 6288271, and 6288277, also infinite recursive loops
- Bug 6289605, a null pointer exception

- Bug 5027753, a redundant comparison to null
- Bug 6266406, which reports a number of defects in the classes implementing JNDI, the Java Native Debugging Interface
- Bug 5023830, a class which defines `equals()`, but not `hashCode()`
- Bug 5027748, use of unnecessary `String` constructors in the `java.util.regex` classes
- Bug 6202042, creation of an object which was never used
- Bug 6228736, a data race in `ByteArrayOutputStream`

We have also filed several bug reports against GNU Classpath [33]. Bugs 13457 (ignored method return values) and 13455 (load and dereference of a value from an unwritten field) were acknowledged to be genuine, but had already been fixed in the development source code repository. Bug 13456, an ignored return value from `String.trim()`, was acknowledged and fixed. Bug 13458, a redundant null comparison, has been assigned to a developer, but not fixed. 12 other reports are still pending, although we have very high confidence that these are serious problems and will eventually be addressed.

4.4 Adoption

Since we made FindBugs publicly available under an open source license, it has been used by many of individuals and organizations.

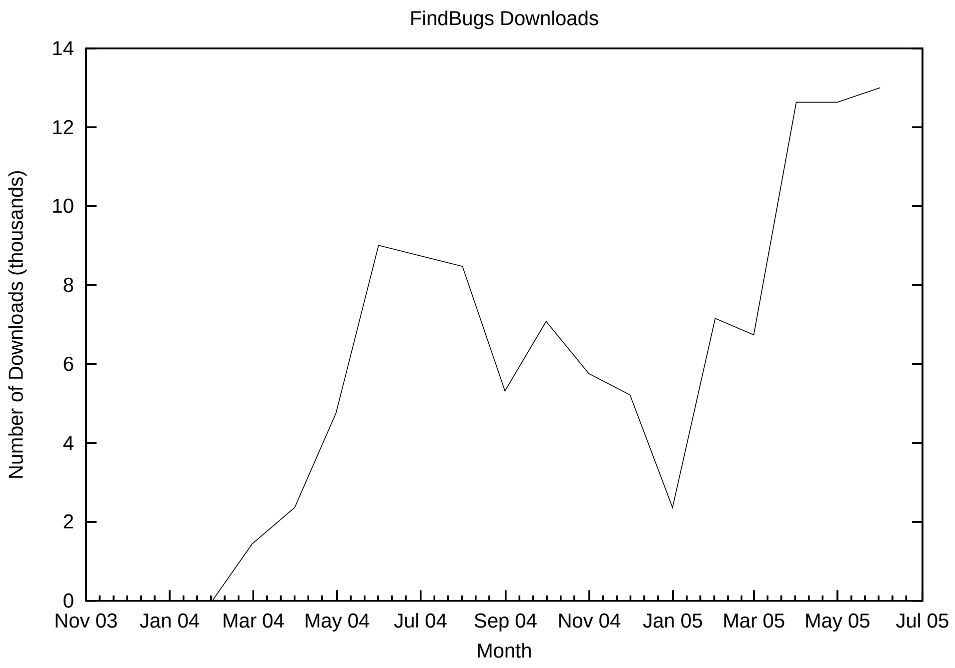


Figure 4.2: FindBugs downloads

4.4.1 Downloads and Web Hits

FindBugs development is hosted at SourceForge [62], which provides free web hosting for open-source projects. One of the services SourceForge provides is activity statistics, including numbers of file downloads and web page hits over time. This information is a useful gauge of the level of interest generated by a project, and may serve as a rough indication of the number of active users.

Figures 4.2 and 4.3 show the number of downloads and web page hits for the FindBugs project from December 2003 (when the project first moved to SourceForge) to the present. There were 14 official FindBugs releases during this period. Interest in FindBugs has been relatively strong; as of June 2005, FindBugs downloads average about 400 per day. While not every developer who downloads Find-

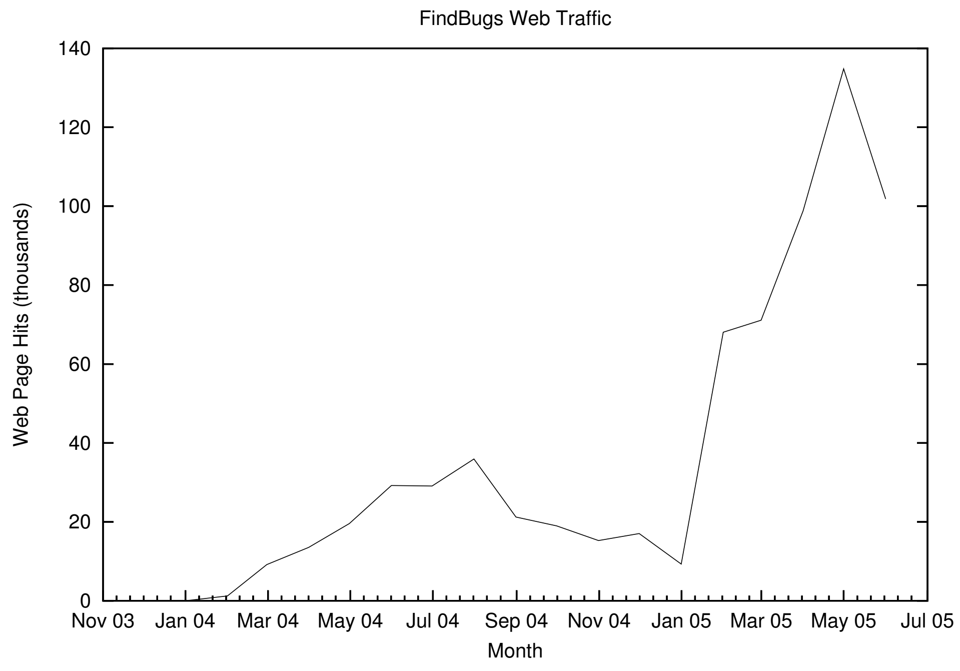


Figure 4.3: FindBugs web page hits

Bugs will become a regular user, we believe that its continuing popularity is evidence that many developers find it to be a valuable tool.

4.4.2 Adoption by Open Source Projects

Open source projects serve a very useful role in the evaluation of software engineering tools, because the development process is conducted in the open. In particular, code repositories, bug databases, and other sources of information about the software engineering practices used by these projects is available at the click of the mouse.

To see if FindBugs has been adopted by any open source projects, we simply entered the term "findbugs" into the Google web search engine [34]. Many of hits are HTML reports produced by FindBugs. By checking the website hosting the report, we found several examples of open source projects which appear to have used FindBugs as part of the development process. Some examples include:

- The Jakarta Commons project, a collection of reusable Java components (`http://jakarta.apache.org/commons`)
- Middlegen, a tool to generate object persistence and GUI code from a database schema (`http://boss.bekk.no/boss/middlegen`)
- ArgoUML, a UML-based object-oriented modeling application (`http://argouml.tigris.org`)
- JaxoDraw, a tool for drawing Feynman diagrams (`http://eee.uv.es/~JaxoDraw/`)
- SAT4J, a SAT solver library for Java (`http://www.sat4j.org`)
- bddbddb, an implementation of Datalog using Binary Decision Diagrams to represent relations (`http://bddbddb.sourceforge.net`)

It is somewhat difficult to know whether these reports indicate that FindBugs was used only once, or whether it is used regularly. From closer inspection of the project website, it is clear that the ArgoUML project runs FindBugs regularly as part of their build process. The projects running FindBugs using the Maven [52] project management system, which include Middlegen, bddbddb, and JaxoDraw, are likely to be running FindBugs regularly, because FindBugs is not a standard part of Maven, indicating that the developers must have thought it was worth the effort to install and configure separately. The extent to which the developers of these projects find the output of FindBugs useful is hard to determine, although from casual inspection of the reports available online, it appears that FindBugs is finding some interesting issues. For example, the report for ArgoUML lists 1 null pointer dereference and 164 redundant null comparisons.

We have also discovered projects using FindBugs though more indirect means. For example, the source distribution for the Berkeley DB Java Edition (`http://www.sleepycat.com/products/je.shtml`) contains a file called `FindBugsExclude.xml`, which is a filter for excluding false warnings from being reported by FindBugs.

4.5 User Experiences

While raw data on downloads and web page hits provides some evidence of a tool's usefulness, it does not offer any insight on how the tool is used in a real development environment. Because FindBugs has an active user community, we have had many opportunities to communicate with software developers to learn more about how FindBugs fits into their development process, and what kinds of bugs it finds.

This section summarizes feedback about FindBugs we have received from users. These reports are not meant to be rigorous; in particular, we cannot say that the developers who responded are a representative sample of Java developers or FindBugs users. However, their responses do provide a glimpse of how a static analysis tool can be useful in a real software development process.

4.5.1 An Informal Survey

In June 2005, we sent an email message to the main FindBugs discussion mailing list asking for feedback from users about how they were applying FindBugs in their development projects. In particular, we asked users to comment on the following issues:

1. What kind of software FindBugs was applied to
2. How often FindBugs was run
3. At what stage of development FindBugs was most useful
4. What kinds of bugs FindBugs has found, and whether they were interesting or important
5. What techniques were used to handle false positives

10 users responded to the survey. Most users who responded were developing server or middleware applications. This is not too surprising, considering the popularity of Java in this domain.

We were somewhat surprised to find that most respondents run FindBugs frequently, as part of active development (4/10), or as part of a nightly or weekly build (3/10). The other respondents use FindBugs only when incorporating major changes, or as part of pre-release testing. The fact that these developers are using the tool relatively frequently suggests that the information the tool produces is deemed worthwhile, and that false warnings are not a serious problem.

All of the developers who responded found FindBugs to be most useful during active development, because it allowed them to find some bugs early in the development process. Several developers also found the tool useful for reviewing existing codebases. These responses confirm our general view of the role of static analysis tools as a first line of defense against programming errors.

We received a variety of responses regarding the kinds of bugs found by FindBugs, and their importance. All of the respondents confirmed that FindBugs had found real bugs in their software. Most (8/10) had found at least some serious bugs in their code. One user who works as both a software developer and a consultant noted that FindBugs had found very little of interest on his own code, but had found numerous problems in his clients' code. Among the most interesting kinds of problems identified by FindBugs were null pointer dereferences, suspicious reference comparisons, and thread synchronization errors.

Interestingly, 5 of the 10 respondents reported that they were not using any false warning suppression strategy, and were willing to review all FindBugs warnings on a regular basis. This suggests the current false warning suppression heuristics implemented in FindBugs are reasonably effective, and that for most code FindBugs does not report an unmanageable number of warnings. The other 5 respondents used filtering (by detector, class, or method) to suppress false warnings. We were somewhat surprised to find that two developers were willing to modify code to suppress false warnings. This suggests that some false warnings, while not actually identifying an exploitable bug, may still highlight sufficiently dubious code to merit a cleanup. This points out an interesting advantage of using simple analysis techniques: code which confuses the static analysis is also likely to confuse developers trying to understand it. One developer wrote:

> Interestingly, really nasty bugs are found by detectors that don't exactly look for the problem that are found because the detector found something tangentially, and by digging around a little, a big problem surfaces. Even reported bugs that we "don't care about" tend to point up other, more serious problems.

This phenomenon also lends support to the idea that static analysis tools can help draw attention to particularly buggy parts of the code, since defects tend to cluster [47].

In addition to responding to the specific questions we solicited, several respondents also provided more general feedback. A common theme in these comments was that FindBugs was a good way to learn about some of the more obscure areas of Java. For example:

> One of the really neat things is your bug descriptions, which serve as a learning tool. I've been doing Java for years and some of the things you detect are totally new to me.

Also:

> FindBugs is also a fantastic way to learn more about Java. People are learning all the time while using FindBugs, often discussing what it is reporting about and what it means to us. Over time, these issues become

ingrained into the developer and there are less repetitive reports as time goes by.

4.5.2 A Geographic Information System Application

One FindBugs user is a team leader in an organization developing a Geographic Information System application. The application comprises over 200 packages, 2,000 classes, and 400,000 lines of code.

The first time FindBugs was applied to this code base, it reported around 3,000 warnings. The developers on the project spent about 1 week addressing these issues, eventually getting the total number of warnings down below 1,000. This shows that, for this application, a significant number of the warnings issued by FindBugs were judged by the developers to be worth fixing. The project leader notes that they are still fixing some of the remaining warnings. Some of the specific types of errors reported by FindBugs that the developers have fixed are hashcode/equals problems, serialization issues, unchecked method return values, unused fields and constants, mutable static data, and null pointer dereferences.

Two specific errors caught by FindBugs illustrate how static analysis tools can be useful for finding subtle bugs in code that appears to be correct upon first inspection. For example, the developers found the following problem stemming from an incorrect boolean operator:

```
if ((tagged == null) && (tagged.length < rev))
```

The second error resulted from confusion between an instance field and an identically named method parameter:

```
public void setData(String keyName, String valName,
  HashMap hashMap)
{
  if (hashMap != null)
    this.hashMap = hashMap;
  else
    this.hashMap = new HashMap(true);

  if (hashMap.size() > 0) {
    ...
}
```

Both of these examples *look* like they ought to work, making it hard to find them by human inspection.

Finally, the lead developer notes that FindBugs has been helpful in large part because of its ease of application; it can be run on the entire application in a few minutes.

4.5.3 A Financial Application

Another user has applied FindBugs to a large (350K lines of code) financial application. He notes that upon first running FindBugs on the application, it produced 300 warnings. Of those, the development team considered about 50 to be real bugs. Many of the remaining warnings are false positives, some of which the team suppresses using a pattern-matching filter.

Figure 4.4 shows a graph of the number of warnings produced by FindBugs, CheckStyle [10], and PMD [55] on the application, over a period of about 5 months. This graph illustrates the distinction between bug and style checkers. Both PMD and CheckStyle, which focus mainly on style issues, produce a far larger number of

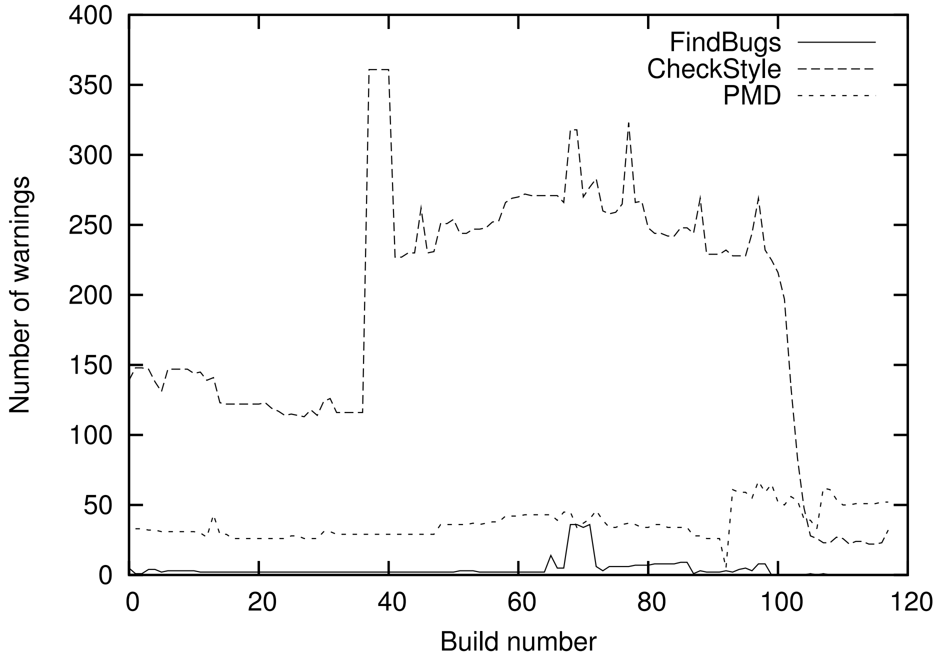


Figure 4.4: Number of warnings produced by FindBugs, CheckStyle, and PMD on a large financial application

warnings than FindBugs. This illustrates the importance of the distinction between bug and style checkers. The fact that the style checkers produced a relatively high number of warnings that were not fixed demonstrates that their role is not to find specific bugs, but to check conformance to existing project standards practices. Tools such as FindBugs can make a valuable addition to style checkers, since they find a significant number of real bugs while producing only a moderate number of overall warnings.

Chapter 5

Experiences Analyzing Student Programming Projects

In this chapter, we discuss using FindBugs to analyze programs written by students in an introductory programming course. Student projects are an interesting target for static analysis for two reasons.

First, by analyzing student projects we can find out what kinds of coding errors students are likely to make. We know from experience that many students find introductory programming courses to be a difficult experience. Without experience to guide them, students struggling to get past a particular stumbling block can become caught up in unproductive "Brownian motion" programming, where they make a series of random changes in the hope of making the project work. This kind of experience can be very discouraging, and may have a negative effect on student retention, especially for students without significant prior programming experience. We surmised that at least some errors typically made by students might be detectable using static analysis, and that by targeting FindBugs to find some of these errors, we might be able to improve students' experience in the course.

Second, student projects are a very useful source of data for evaluating and improving the accuracy of static analysis. Code written by professional programmers often is complex, vaguely specified, and generally lacks an objective way to measure whether or not the code is implemented correctly. In contrast, student projects in introductory programming courses tend to be simple, well-specified, and most importantly, often have a test suite to measure how well the implementation matches the specification. A comprehensive test suite allows us to do something that is extremely difficult to do for arbitrary production software artifacts: determine what bugs are *not* found by static analysis.

By analyzing static analysis warnings and test failures for student projects, we can make progress toward two important objectives:

1. Finding new bug patterns, by examining test failures not predicted by any static analysis warning

2. Improving analysis effectiveness and accuracy by identifying and addressing the causes of false negatives and false positives

5.1 The Marmoset Project

In order to better understand bugs in student code, we developed a project snapshot, submission, and testing system called Marmoset [63, 64]. Like many similar systems, Marmoset allows students to submit versions of their projects to a central server, which automatically tests them against a suite of unit tests and stores the results in a database. However, it differs from previous systems in two ways.

First, it employs a novel mechanism for providing feedback about the results of unit tests. Students can request to see the names of passed unit tests, along with the names of the first two failed unit tests, subject to the availability of *release tokens.* Each student has a small number of tokens available. Once used, a token regenerates after a fixed period of time. For the courses in which we have used Marmoset, we have given students 3 release tokens, each of which regenerates 24 hours after being used. This system encourages students to start work early (since the release tokens can only be used a fixed number of times per day), and also provides an incentive for students to think carefully about their work before using a token.

The second novel feature of Marmoset is that it collects fine-grained snapshots of student projects as they work. Each time a student saves a file, her work is automatically committed to a CVS [14] repository. Over time this creates a detailed history of each student's work. Most of the code changes captured by Marmoset are small: 70% add or change no more than 4 lines of code. A summary of the number of code snapshots and the sizes of code deltas from one snapshot to the next from one semester of an introductory Java programming course is shown in Figure 5.1. In this course, Marmoset captured 33,015 unique snapshots from 73 students over the course of 8 programming projects. These snapshots are an ideal laboratory for studying bugs: they record the work of many programmers using many different programming styles and idioms, and record the code changes made during active development, allowing us to see how bugs are introduced and fixed.

5.2 Discovering New Bug Patterns

One important use of the student snapshot data has been to discover new bug patterns. A simple and effective way to discover new bug patterns is to look at unit tests which fail because of a runtime exception. When we examine all snapshots in

Lines Added or Changed	Number of Commits	Total
1	12,873	39%
2	5,484	56%
3-4	4,726	70%
5-8	3,608	81%
9-16	2,503	88%
17-32	1,229	92%
33-64	612	94%
65+	352	100%

Figure 5.1: Summary of delta sizes for code snapshots captured in CMSC 132 (Object-Oriented Programming II) in Fall 2004

which a particular type of runtime exception is thrown, odds are good that we will find that many of those exceptions occurred for similar reasons: a candidate for a bug pattern.

Table 5.1 shows the number of student code snapshots where at least one unit test failed with a runtime exception for the Fall 2004 and Spring 2005 semesters of CMSC 132. Null pointer exceptions were the most frequent problem. We found that a surprising number of tests failed because of `StackOverflowError` and `ClassCastException`. By examining snapshots where these exceptions occurred, we were able to generalize common causes. This process led to the development of detectors for Infinite Recursive Loops (Section 3.5) and Bad Casts (Section 3.6).

Although both the IL and BC patterns are common in code written by beginning programmers, we have also found many occurrences of these patterns in production code. This confirms our observation that simple, dumb mistakes can be found in almost all software.

The exception data indicates that many bugs in student code are due to an

Exception type	Number of snapshots
NullPointerException	10,684
ClassCastException	4,348
IndexOutOfBoundsException	3,094
OutOfMemoryError	2,605
StringIndexOutOfBoundsException	2,124
ArrayIndexOutOfBoundsException	2,084
StackOverflowError	1,949
NoSuchElementException	1,928
IllegalArgumentException	1,877
IOException	1,683

Table 5.1: Most frequently occurring runtime exceptions in student code snapshots

out-of-bounds index used to read an element from an array, string, or list. While these bugs are certainly worth investigating, they may be unlikely to have common causes which could be generalized as a bug pattern.

5.3 Experimental Setup

In order to measure the effectiveness of FindBugs on student code, we examined student code snapshots from two programming projects:

- Web Spider: crawl web pages in depth-first or breadth-first order
- Search Tree: implement a balanced binary tree

We annotated both projects with `@NonNull` and `@PossiblyNull` annotations for parameters and return values. (The meaning of these annotations is described in Section 3.3.10.) For example, the `Spider.crawl()` method of the Web Spider project takes a `URL` object as a parameter. We annotated the parameter as `@NonNull`, since `crawl()` is required to use the `URL` object. Another example is the `Spider.getNextPageToCrawl()` method, which returns null if there are no more pages to crawl. Therefore, the return value of this method was annotated as `@PossiblyNull`. Adding annotations allowed us to measure their effectiveness at finding null pointer bugs (beyond those found by our basic intra-procedural analysis.)

In addition to null pointer bugs, we also looked at stack overflow and dynamic type cast bugs, both of which manifest as easily recognizable runtime exceptions, have detectors implemented in FindBugs, and pinpoint a precise source line where a runtime exception is expected if the warning is accurate.

For both projects, we measured the rate of false negatives and false positives. False negatives are snapshots where a runtime exception occurs without any warning predicting the exception. False positives are warnings that do not predict any runtime exception.

The main question in measuring false negatives and false positives is which snapshots and warnings to count. There are two issues to consider in answering this question. First, we cannot easily evaluate the accuracy of a warning if the statement it identifies is never executed. To resolve this issue, we collected code coverage data for each unit test, and excluded all warnings never covered by any test case from consideration. The second issue is slightly more subtle. Because Marmoset collects fine-grained snapshots, it tends to record a single warning or runtime exception many times. If we consider all snapshots and warnings, we tend to overcount those that persist through many successive versions. This will overcount warnings that are false positives, and may undercount exceptions that are false negatives. To solve this problem, we only counted warnings that are either removed in the subsequent snapshot, or are present in the final snapshot, as potential false positives. Likewise, for potential false negatives we only counted the first in a chronological series of snapshots where a particular runtime exception occurred.

After selecting the subset of snapshots and warnings to count, we calculated false negatives and false positives as follows:

$$\text{False negatives} = \frac{\text{Snapshots with exception but no warning}}{\text{Snapshots with exception}}$$

$$\text{False positives} = \frac{\text{Warnings with no exception thrown}}{\text{Warnings}}$$

5.4 Discussion of Results

The results of the experiment are shown in Figures 5.2 through 5.5.

Warnings issued by the strictly intra-procedural null pointer analysis (Figure 5.2) were very accurate, with a 2% false positive rate, but predicted only a small percentage of the null pointer exceptions, with a 70% false negative rate for the Web Spider project and a 98% false negative rate for the Search Tree project. Fortunately, with the addition of annotations (Figure 5.3), the situation improves greatly. The analysis finds more than 50% of the null pointer bugs in the Search Tree project, and nearly 80% of the null pointer bugs in the Web Spider project. The higher percentage of bugs found comes at the cost of increasing the false positive rate. However, the observed rate of 10%–21% is still very acceptable.

The detectors for infinite recursive loops and bad dynamic casts (Figures 5.4

Project	Snapshots	With warning	Observed false neg. %
Search Tree	71	1	98
Web Spider	162	47	70

Project	Warnings	With exception	Observed false pos. %
Search Tree	2	2	0
Web Spider	77	75	2

Table 5.2: Observed false negative and false positive rates for null pointer warnings in student projects (without annotations)

and 5.5) were generally not as effective as the null pointer bug detector. The infinite loop detector did find almost 90% of the stack overflow exceptions in the Web Spider project. However, this was due mainly to the fact that most of those bugs were caused by many students misinterpreting an API comment in the same way (as shown in Figure 3.11). The resulting bugs were easily detectable. The detector was not as effective at finding stack overflows in the Search Tree project. For both projects, the warnings that were issued were very accurate. The bad cast detector found 8% and 32% of the class cast exceptions for the Search Tree and Web Spider projects, respectively, with a reasonably low rate of false positives.

5.5 Reasons for False Negatives and False Positives

As we classified false negatives and false positives, we tried to identify ways in which the static analysis could be made more accurate. In keeping with our philosophy of using the simplest analysis techniques possible, we only enumerated cases where a simple technique or heuristic could be effective, and labeled the other cases as

Project	Snapshots	With warning	Observed false neg. %
Search Tree	71	38	46
Web Spider	162	127	21

Project	Warnings	With exception	Observed false pos. %
Search Tree	40	36	10
Web Spider	129	101	21

Table 5.3: Observed false negative and false positive rates for null pointer warnings in student projects (with annotations)

Project	Snapshots	With warning	Observed false neg. %
Search Tree	46	7	84
Web Spider	18	16	11

Project	Warnings	With exception	Observed false pos. %
Search Tree	7	7	0
Web Spider	16	16	0

Table 5.4: Observed false negative and false positive rates for infinite recursive loop warnings in student projects

Project	Snapshots	With warning	Observed false neg. %
Search Tree	28	2	92
Web Spider	50	16	68

Project	Warnings	With exception	Observed false pos. %
Search Tree	2	2	0
Web Spider	17	11	35

Table 5.5: Observed false negative and false positive rates for dynamic cast warnings in student projects

Reason	False neg. occurrences	False pos. occurrences
Complex program state or calling context	45	25
Value loaded from Map	10	2
Incomplete object initialization	11	0
Infeasible branch	0	3
Invalid null check using compareTo()	1	0
Implementation more robust than annotation	0	2

Table 5.6: Reasons for null pointer false negatives and false positives

Reason	False neg. occurrences
Complex program state or calling context	23
No modified data read on path to recursive call site	4
No heap writes on path to recursive call site	14

Table 5.7: Reasons for infinite recursive loop false negatives

"complex."

Figures 5.6 through 5.8 list the causes of false negatives and false positives for the snapshots we inspected. While these classifications are informal, they do point out some relatively easy ways to improve the accuracy of the underlying analyses. The remainder of this section briefly describes each source of inaccuracy we identified.

Complex program state or calling context. This is a catch-all category for false negatives or false positives which would require detailed information about the program state or calling context to correct. In other words, these are cases where we do not envision a simple analysis or heuristic being effective at avoiding the inaccuracy.

Value loaded from Map. These are false negatives or false positives result-

Reason	False neg. occurrences	False pos. occurrences
Complex program state or calling context	3	0
Stronger type analysis needed (field types)	21	0
Non-Comparable object added to SortedSet	4	0
Stronger type analysis needed (method return types)	9	6
Implicit type parameter of collection is violated	21	0
Using wrong map iterator	2	0

Table 5.8: Reasons for dynamic cast false negatives and false positives

ing from a call to the `get()` method on a `Map` object. False negative arise because the program dereferences the return value in cases where the `Map` does not contain the requested entry. False positives arise due to annotations: a call to a method whose return value is declared `@PossiblyNull` is dereferenced immediately because the caller knows the method is implemented by a map lookup of a key known to be in the map. It is possible we might be able to construct a static analysis to determine when a calling context will use a key guaranteed to definitely be in the map, or guaranteed not to be in the map, and issue or suppress a warning as appropriate.

Infeasible Branch. Infeasible control flow is a common source of inaccuracy in program analysis. One kind of false positive we noticed several times in the search tree project was code of the following form appearing in a method whose return value was declared `@NonNull`:

```
int value = ...
Object result = null;

if (value < 0) {
  result = non-null value
} else if (value > 0) {
  result = non-null value
} else if (value == 0) {
  result = non-null value
}
return result;
```

The final `if` statement is guaranteed to evaluate to a true value due to the range of the variable `value`. Because the `else` branch in infeasible, `result` is guaranteed to be non-null when the `return` statement is reached. We could eliminate this particular form of false positive using a value range analysis.

Incomplete object initialization. We noticed several instances of null pointer exceptions caused by loading a value from a field not initialized by the object's constructor. In these cases, the uninitialized field is assigned a non-null value by at least one public method. However, if the field is loaded and dereferenced before any method which initializes the field is called, then a null pointer exception will occur. These cases could be found by looking for classes with public constructors that fail to completely initialize all of the object's fields, and public methods which dereference possibly-uninitialized fields without a check. Careful evaluation would be needed to ensure that a detector taking this approach did not produce too many false positives.

Invalid null check using compareTo(). In one case, a student tried to check whether or not an object reference was null by calling the `compareTo()` method on it, passing a null value as the argument. The `Comparable` interface that defines this method does not require it to handle a null argument, so a null pointer exception is very likely in this case. In general, a large number of classes and interfaces in the standard Java library could have their parameters or return values annotated with

`@NonNull` and `@PossiblyNull` to catch bugs like this one.

Implementation more robust than annotation. Some students implemented methods which were more robust than required by the project specifications. For example, although the `SearchTreeMap.put()` method in the Search Tree project was specified as having a `@NonNull` key, some students coded this method to allow (and ignore) null keys, and tested this functionality in their own test cases. Relaxing the annotation of the method would eliminate these false positives.

No heap writes on path to recursive call site. If no heap writes occur on the path from the entry point of a method to a recursive call which passes all method parameters as arguments, it is safe to assume that an infinite recursive loop will be entered. Our detector for infinite recursive loops can detect this situation as long as all heap reads and writes are local to the method; however, if a method call is seen, it makes the conservative assumption that any heap location could be read or written, and thus the recursive invocation might take a different path. In the Search Tree project, we noticed several cases of false negatives due to method calls between the method entry and recursive call site where the called methods were "read-only." For example, calling the `compareTo()` method to compare two objects typically does not modify heap state. Therefore, we could eliminate some of these false negatives by encoding knowledge of methods which are read-only by convention.

No modified data read on path to recursive call site. Another source of false negatives we noticed in the Search Tree project were methods which performed logically write-only operations on an object passed to the method, but did not invoke any methods which would reveal changes to that object on the path to a recursive call where all of the method parameters were passed as arguments. This suggests that we might be able to catch some of these cases by expanding the detector's definition of reads and writes to encompass knowledge about "logical" read and write operations on objects, such as instances of container classes.

Implicit type parameter of collection is violated. Prior to the introduction of generic types into Java [41], generic containers were written using `Object` references as the element type for collections. In theory, it is possible to store unrelated object types in a container, since all objects are subtypes of the base `Object` class. However, in practice all containers have implicit *type parameters* specifying what kinds of types are allowed in the container. Because these parameters are implicit, they are prone to being violated if the programmer is confused about what kind of objects a container actually stores, and either tries to cast an object obtained from the container to the wrong type, or puts the wrong type of object in the container. This was the largest source of missed dynamic cast bugs. Every one of the false negatives in this category would have been found at compile time if generic types were used.

Non-Comparable object added to SortedSet. We noticed several instances of an object not implementing the `Comparable` interface being added to a `SortedSet`. Unless the set uses an explicit comparator object, this will result in a `ClassCastException` when the implicit comparator tries to cast the elements to `Comparable`. (An analogous situation can arise for keys used with `SortedMap` objects.) To catch such cases, our analysis would need to identify `SortedSet` objects not having an explicit comparator, and check elements passed to their `add()` method to ensure they implement `Comparable`.

Using wrong map iterator. Some students attempted to iterate through either the keys or values of a `Map` object using an `entrySet()`'s iterator, which returns `Map.Entry` objects. We could extend our analysis to check the uses of iterator objects returned from `entrySet().iterator()` to ensure that they are cast to `Map.Entry`, and not some other type. Note that these cases would be caught at compile time using Java generics.

Stronger type inference needed (method return types). Our detector for suspicious dynamic casts issues a warning when a generic collection interface, such as `List` or `Map`, is cast to an abstract or concrete collection class type, such as `AbstractSet` or `HashMap`. Such casts are unnecessary, violate encapsulation, and generally indicate confusion or malice on the programmer's part. However, if the reference really is of the correct type, these casts are harmless. Some of the false positives generated by FindBugs could be eliminated using more powerful type inference. For example, a method declared to return `List` might be guaranteed to return `LinkedList`, allowing the more precise type to be known at call sites. However, suppressing these false positives is almost certainly a bad idea, because a method return type defines a contract between the caller and the callee. If a method is declared to return `List`, then it is perfectly valid to return any subtype of `List`. Implementations that violate this encapsulation are very likely to break when the program is modified and the method is changed to return a different subtype.

Stronger type inference needed (field types). We noticed several dynamic cast bugs not caught by FindBugs where a value was loaded from a field and immediately cast to a subtype of the field's declared type. The analysis could potentially keep track of the concrete types stored in fields, and issue a warning if a cast was used when it was likely that any of several subtypes could be stored in the field. However, it is likely that some or most such warnings would be false positives, because the code could have knowledge of what type was stored in the field based on the program state. Since fields are usually private to the implementation of a class, breaking encapsulation is not a concern; so a false positive referring to a value loaded from a field involved in a dynamic cast would be less acceptable than a false positive referring to a value returned from a method.

Chapter 6

Experiences Studying Static Analysis Warnings Over Time

As code is modified and software evolves, static analysis results will also change. We would like to be able to understand changes in static analysis results in order to determine several important phenomena. By looking at when warnings come into existence and when they disappear, we can gain insight on which warnings represent bugs, and therefore where bugs are being fixed in the software. We can also look for warnings which persist through many versions of the software, which are likely to be false positives.

Being able to compare warnings through multiple versions is also important for false positive suppression. As developers inspect warnings and classify them as serious or false positives, we can carry these classifications forward to new versions of the software. In this way, developers can avoid inspecting the same false positives over and over again.

6.1 Comparing Warnings Across Versions

Software artifacts are not static. Most software is developed in an environment of constantly changing requirements and specifications. As features are added and changed, and as bugs are fixed, the code changes continuously. Static analysis tools must accommodate this change by being able to determine when static analysis warnings are “the same” in two different versions of the software artifact. We refer to this problem as the *warning comparison* problem.

The obvious difficulty in comparing warnings from different versions of the same software artifact is that because the underlying code may be different, the details of the warnings, such as class names, method names, and source line numbers, may be different, even though the underlying cause of the warnings is the same. For this reason, a warning comparison procedure must have some degree of fuzziness, allowing warnings from different versions of the code to match even though some underlying details may have changed. However, if the comparison procedure is too fuzzy, it risks being unable to distinguish warnings which are actually different.

6.1.1 Warning Comparators

A warning comparison procedure, or *warning comparator*, allows us to perform set operations on collections of warnings from different software versions. A warning comparator may be viewed as a function that takes two warnings from different software versions and returns true if they are equivalent, or false if they are not equivalent. The notion of “equivalence” is inherently vague. In general, it is meant to capture the notion that the warnings describe the same code features in both versions, and were issued for the same reason. However, because some of the relevant code may have changed between the two versions, the features of the warnings may be slightly different.

FindBugs supports two warning comparators. Both comparators work by comparing warnings based on their *attributes*. (Warning attributes and the data model for warnings are described in Section B.1.)

The *exact* warning comparator compares warnings based on the bug pattern type, and all class, method, field, and source line attributes, all of which must match exactly in order for the compared warnings to be considered equivalent. Source lines within methods are compared as offsets from the beginning of the method, rather than as absolute line numbers within the source file. The idea is to allow code to be added or removed earlier in the file without affecting the ability to match warnings inside the method, as long as the method has not changed. When the exact warning comparator judges two warnings to be equivalent, it is very likely that the warnings really are the same. However, small changes in line numbers within a method may cause the warning to fail to match some warnings which are equivalent.

The *fuzzy* warning comparator matches bug pattern type and class, method, and field attributes only. It ignores source line attributes. For this reason, it may consider some warnings equivalent when they are not. However, this is only possible for warnings in the same method. Given the relatively small number of warnings emitted per method, the overall accuracy of the fuzzy comparator is reasonably good—we have measured it as more than 90% accurate when comparing warnings between versions of the core J2SE libraries. Unlike the exact comparator, it tolerates changes in source line numbers within methods.

6.1.2 Set Operations

The important operations on sets of warnings W_n and W_{n+1} representing the results of applying static analysis to software versions n and $n + 1$ are:

1. Added warnings: $W_{n+1} - W_n$

2. Removed warnings: $W_n - W_{n+1}$

3. Retained warnings: $W_n \cap W_{n+1}$

All of these operations can be reduced to the problem of matching equivalent warnings in W_n and W_{n+1}. After all equivalent warnings are matched, the unmatched warnings in W_n are in the removed set, and unmatched warnings in W_{n+1} are in the added set.

FindBugs matches warnings incrementally by starting with the exact comparator, and moving to increasingly fuzzy comparators. Each comparator is used to greedily match as many equivalent warnings as possible. If a warning in W_n matches multiple warnings in W_{n+1}, one of them is picked arbitrarily. The process terminates when all warnings in either W_n or W_{n+1} are matched with an equivalent warning, or when there are no more comparators. This approach has the advantage that the exact comparator will be used to match warnings in methods that have not been modified, while less accurate comparators will only be used for warnings in code that has been changed.

6.1.3 Handling Moved or Renamed Classes

As code evolves, classes may be renamed, or moved from one package to another. This makes comparing classes from different software versions somewhat complicated. For the warnings comparators used in FindBugs we have implemented a simple technique which can detect when classes are moved to a different package. Before comparing warnings from different software versions, the tool notes “removed” classes, which classes exist only in the first version, and ”added” classes, which classes exist only in the second version. It then looks among the added and removed classes for pairs of classes which differ only by package name. These classes are then considered to be the same when matching warnings.

We have not yet implemented a technique to detect renamed classes. That problem is somewhat harder; without class names to guide the tool, it is difficult to know which pairs of classes it should consider equivalent. We have experimented with a similarity metric based on class features which would be likely to be unchanged after a renaming, such as field names and types, with some initial success.

6.2 A Case Study

One of our primary motivations for developing warning matching techniques was as a tool to investigate whether or not the warnings produced by FindBugs identify bugs that are actually fixed by the developers of the software. A common criticism of work on static analysis to find bugs is that the bugs found are often trivial, and that fixing them will not necessarily result in a noticeable improvement in software quality. This criticism is very difficult to address, since for most software artifacts we do not know how many bugs exist overall, or where the “important” bugs are. In our study of student programming projects (Chapter 5) we were able to partially address this concern by measuring the false negative rate for certain kinds of bugs, based on matching warnings and unit test failures. However, most production software artifacts do not have a sufficiently comprehensive test suite for this kind of experiment. Nonetheless, we can show some evidence that our static analysis tool finds important bugs by showing that developers are willing to fix at least some of the bugs it reports.

As a case study, we looked at several versions of the Berkeley DB, Java Edition [8]. This project is an interesting example for several reasons. First, based on the existence of a false positive suppression filter, we discovered that FindBugs had been used as part of the development process. The software is widely used, and because it is an embedded database supporting concurrency and transactions, freedom from defects is an important requirement. Our goal was to use our warning matching techniques to find evidence that bugs reported by FindBugs were being fixed.

We looked at four versions of the software:

1. Version 1.3.0: a beta release

2. Version 1.5.0: first production release (61 KLOC)

3. Version 1.7.1: FindBugs used for this and later versions

4. Version 2.0.24: current release (77 KLOC)

For each version of the software, we ran FindBugs and used our warning matching algorithm to find removed, retained, and added warnings. Table 6.1 shows the results of this analysis.

Our hypothesis was that we would see a significant number of warnings removed at the point in the development history where FindBugs was used for the

	Warnings			
Version	Added	Removed	Retained	Active
1.3.0	0	0	0	88
1.5.0	3	8	79	87
1.7.1	6	51	36	42
2.0.42	1	15	60	34

Table 6.1: FindBugs warnings in Berkeley DB, Java Edition

first time. This hypothesis was confirmed, as more than half of the warnings reported in version 1.5.0 were removed in version 1.7.1. This is strong confirmation that the issues reported by FindBugs were deemed to be worth fixing by the developers. Over the versions of the software we examined, we observed 74 removed warnings. Among the warnings removed were:

- 12 Inconsistent Synchronization warnings
- 3 Double Checked Locking warnings
- 10 Hashcode/Equals warnings
- 5 Redundant Null Check warnings
- 5 Dead Local Stores
- 1 Read Return warning

The presence of warnings related to possible synchronization bugs is significant, since the software supports concurrency.

Overall, this case study was a useful validation of the usefulness of FindBugs in a real software development project, and of the usefulness of our warning matching techniques. In the future, we plan to study the effects of using FindBugs on the development of larger systems. Our goals include characterizing what bugs can be found with static analysis compared to other techniques, and improving the accuracy of static analysis by identifying long-lived warnings likely to be false positives.

Chapter 7

Related Work

In this chapter, we describe related work on static analysis to find bugs in software.

7.1 Static Analysis

Static analysis techniques are used to extract properties from a program by analyzing its code. Because nontrivial properties of programs are undecidable in general, the results of static analysis are usually approximate. Unlike forms of analysis relying on program execution, such as testing and program instrumentation, static analysis can infer properties about all possible executions of a program.

7.1.1 General Static Analysis Techniques

One of the first general frameworks for static analysis was *dataflow analysis*, introduced by Kildall [46]. Dataflow analysis tracks dataflow facts at points in the control flow graph of a procedure. The dataflow facts form a lattice. Given a dataflow fact for the entry to a control flow graph, the dataflow analysis algorithm computes facts at all points within the graph. The effects of basic blocks are computed using *transfer functions*, which summarize the effects of instructions in basic blocks. A meet function is used to combine dataflow facts at control join points. The analysis iterates by propagating dataflow facts through basic blocks and along control edges until a fixpoint is reached. Kildall proved that given a fixed height lattice and monotone transfer functions, dataflow analysis is guaranteed to terminate and produce correct results. The results are an approximation of the real behavior of the procedure. Dataflow analysis is relatively inexpensive to perform (usually linear in the size of the control flow graph), and almost any analysis property can be formulated as a dataflow problem. However, the loss of information at join points in dataflow analysis results in lost precision. In our work on FindBugs, we have used dataflow analysis extensively to find properties such as the types of variables, which variables may contain a null value, which program statements are covered by locks, and many others.

Abstract interpretation, introduced by Cousot and Cousot [12], was developed to provide a firmer theoretical foundation for dataflow analysis. An *abstraction function* maps the infinite domain of program objects into a finite domain of abstract objects. The effects of program operations are modeled using the abstract domain. A *concretization function* maps the abstract objects back into the program domain (with some loss of precision). Dataflow analysis can be considered a special case of abstract interpretation.

ASTLOG [13] is a language for specifying patterns to match the abstract syntax trees of C and C++ programs. Unlike dataflow analysis and abstract interpretation, this kind of analysis does not directly model the semantics of program operations, focusing entirely on syntactic structures. Even given this limitation, ASTLOG has been used successfully to find bugs and performance problems, and is the basis of the PREfast tool used extensively within Microsoft to find bugs.

ASTLOG is an interesting data point in the space of static analysis techniques to find bugs, because it shows that simple pattern-matching approaches can be very effective at finding interesting program features, such as probable bugs. The bug detectors in FindBugs implemented as a sequential scan over bytecode instructions using a state-machine recognizer are similar to the approach used in ASTLOG in many respects. The main difference is that we have not codified a language to describe these recognizers, relying instead on a visitor-based implementation strategy.

Liu et. al. [51] describe an algorithm for matching regular queries on nodes in a control flow graph. Nodes may be matched against *constructors*, which represent the meaning of the node. For example, the constructor **def** might indicate that the node defines the value of a variable. Nodes may also be matched against *symbols*, which represent variables or literal values referenced by the node. For example, the expression $\mathbf{def}(a)$ would represent a definition of a variable to be referred to as a in the query. All occurrences of a particular symbol in the query must match the same underlying variable or constant. Constructors and symbols may be formed into regular expressions with wildcards, repetition, and negation. Queries are executed at a given start node $v0$, and the results of the query are pairs (v, θ) where v is a node such that a sequence of symbols from $v0$ to v matches the query, and θ is a map of symbols used in the query to variables and constants matched by those symbols. Variations of the algorithm are presented for both existential and universal queries, which are analogous to *may* and *must* formulations of dataflow analysis. The authors present a number of standard dataflow analyses expressed as parametric path queries, as well as formulations of checkers for correctness properties such as lock ordering and correct ordering of file I/O operations. One limitation of parametric path queries is the reliance on matching symbol names to variable names in order to express data dependences; there is no simple way to extend this matching to heap objects. However, because many interesting bugs may be found through local analysis, parametric path queries could serve a very useful purpose for concisely expressing local correctness properties. In the future we intend to explore the use of regular path queries as a replacement for the ad-hoc visitor-based bug detectors in FindBugs. We believe that they would provide a concise and flexible way to express recognizers for many kinds of bug patterns, and might allow users to add domain-specific bug detectors more easily.

7.1.2 Static Analysis to Find Bugs

The literature concerning static analysis techniques to find bugs is very extensive. This section only attempts to summarize a few of the most relevant works.

Lint [45], the ancestor of all program checkers, was designed to find common C programming errors. Due to the limited CPU and memory resources available at the time of its implementation, the analysis performed by lint is simplistic. For example, lint checks for uses of uninitialized local variables by seeing if they are assigned earlier in the source file than their first use, rather than using a more sophisticated analysis (such as dataflow analysis). Due to the imprecision of lint's analysis, special annotation comments may be added to the program's source code to suppress false warnings. Many checks performed by lint are now built into modern C compilers. FindBugs very much falls within the category of Lint-like program analysis tools. However, because of the much greater computing power available today, it uses more powerful analysis techniques, such as dataflow analysis. Because FindBugs analyzes Java rather than C, it focuses less on low-level bugs, such as memory allocation and dereferencing errors, and more on higher level concerns, such as API use.

LCLint [26] is a static analysis tool to detect errors in C programs. For programs with no specifications, LCLint works much like the original lint tool, warning about common C programming mistakes. For programs with specifications, LCLint checks the code for consistency with the specifications. Examples of properties checked by LCLint include incorrect use of boolean values, abstraction violations (such as incorrect use of a "private" element of a struct), and violations of temporal properties (such as using a field before it is initialized). Subsequent work extended LCLint to check for memory errors [25], such as failure to free allocated memory and use of uninitialized memory, and security exploits [27], such as buffer overruns and format string vulnerabilities. The analysis performed by LCLint is neither sound nor complete. However, explicit specifications help to make the analysis more precise in cases where properties cannot be inferred by analysis alone. The null pointer annotations in FindBugs (described in Section 3.3.10) were inspired by LCLint, and our study of bugs in student programming projects confirm the value of specifications in finding bugs.

Warlock [65] is a lint-like static analysis tool to find race conditions in C programs. Each source file of a program is analyzed to determine which mutex locks are used to protect variable accesses. Variables not protected by a consistent lock (or set of locks) are flagged. The analysis is inter-procedural; functions with no callers are assumed to be call graph roots. Imprecision in the analysis is mitigated through the use of annotations, which can be used to describe the locks intended to protect particular variables, which sections of code are multithreaded, which functions might be called through a function pointer, etc. The detector for inconsistent synchronization in FindBugs uses a somewhat similar approach; however, it relies on the common Java idiom of locking an object in order to protect its fields, and does not handle the more general case where any object might be used to protect a particular memory location.

PREfix [9] applies static analysis to find common dynamic errors in C and C++ programs. Examples of errors detected by PREfix are invalid pointer accesses, leaked memory, use of freed memory, and use of an invalid resource (such as a file or window handle). The analysis performed by PREfix works by symbolically executing a large number of paths through the program, tracking the state of variables and memory. PREfix works inter-procedurally, starting at the call graph leaves and working towards the roots. It analyzes functions by evaluating a large number of paths through their control flow graphs. To a significant degree, the values of expressions are tracked using predicates, to avoid analyzing infeasible paths. As functions are analyzed, models are built for later use when calls to those functions are encountered. Constraints are added to functions to detect situations such as passing a null pointer to a function which dereferences it unconditionally. PREfix was able to find significant numbers of errors in large software systems with a low (25% or less) false positive rate. The authors noted that a large amount of implementation effort was required to present detected errors to the user in a meaningful way. PREfix is a good example of a “deep” analysis to find bugs. In our work on FindBugs, we have focused largely on bugs which can be found using local analysis. One of the surprising conclusions of our work is that a significant number of serious bugs can be found *without* deep analysis.

MC [20, 36] (*Meta-Compilation*) is an inter-procedural static analysis system for analyzing C and C++ programs. The novel aspect of MC is the use of a language, called Metal, to describe the properties checked by the analysis. Metal is based on a syntactic pattern matching technique similar to ASTLOG. Syntactic patterns are used to describe functions and data objects of interest to the analysis. The MC system uses state machines to describe temporal correctness properties. For example, a lock object can be in unlocked or locked states. The analysis creates new state machine instances as interesting objects are encountered. Syntactic patterns specify when state transitions should occur; for example, calling a lock function changes the state of the lock. Some state transitions represent bugs, such as locking a non-recursive lock object already in the locked state. The analysis also has the notion of global state to describe global properties such as whether or not interrupts are enabled. The analysis is inter-procedural, starting at the call graph roots and working towards the leaves. Aggressive caching and memoization of function effects are used to make the analysis more efficient, at the expense of accuracy and correctness. Within functions, the analysis is path sensitive, using knowledge of variable states to avoid analyzing false paths. MC is similar to PREfix in many respects. Its main innovation is providing a generic framework for expressing and checking correctness properties. Our work on FindBugs focuses on a somewhat different class of bugs, although there are areas of overlap.

RacerX [21] is a static analysis system to find deadlocks and data races in multithreaded software. The techniques employed by RacerX are very similar to those used in Warlock; lock sets are tracked at all program points. Variables not accessed with a consistent lock set are potential data races. In order for the analyzed system to be free of deadlocks, an acquisition of a lock l implies an ordering constraint that all locks in the current lock set must be acquired before l. Based on all observed constraints, RacerX computes the transitive closure and emits potential deadlock warnings for any detected cycles. The analysis used is very similar to MC; however, variables are not tracked to eliminate false paths. To reduce the impact of false positives due to imprecision in the analysis, the authors employ a wide variety of heuristics. One novel technique is *unlockset analysis*, which computes locks known *not* to be held using a backwards analysis. Unlocksets are used to increase confidence in the accuracy of locksets; when the unlockset disagrees with the lockset, the most likely explanation is that an analysis error wrongly concluded that a lock was held. RacerX also uses heuristics to rank the warnings produced by detected data races. For example, unsynchronized variable accesses or unlocked function calls are ranked higher if they are the first or last statement in a critical section, since those accesses are most likely to be intentionally protected by the lock, and not merely coincidentally locked. The authors used RacerX to find a number of deadlocks and data races in Linux, FreeBSD, and a commercial OS kernel. FindBugs also has detectors for data races and deadlocks. However, they detect a more limited subset of cases. For example, FindBugs only detects data races related to the common idiom of locking an object to protect access to its fields. Deadlock detection in FindBugs is limited to identifying locations where a wait is performed with two locks held. What is interesting in FindBugs is that simple analysis techniques can reveal serious bugs.

7.2 Ranking Techniques

Almost all useful bug checkers produce some false warnings. If the ratio of false warnings to real bugs found is too high, using the tool may be unproductive. One way to mitigate the impact of false warnings is to use ad-hoc heuristics to identify and suppress probable false warnings. A more general way of dealing with false warnings is for the tool to produce a *ranking* in which the warnings likely to be genuine are ranked higher than the warnings likely to be false. Even if the ratio of false warnings is very high, the existence of a good ranking allows the tool to be used productively. Warnings are inspected in order of their rank, with the warnings most likely to be real bugs inspected first. The inspection continues until the false positive ratio becomes too high.

Z-ranking [48] is a statistical technique proposed by Kremenek and Engler for ranking static analysis warnings. The technique works in the context of static bug checkers which analyze many possible paths through a program, which means that each program location will generally be checked many times. Z-ranking is based on the intuition that programmers usually write correct code, so at any given program location, the correctness property being checked will usually hold. Thus, if a property seldom or never checks as correct at a given location, it is likely that the violation is the result of an invalid static analysis decision rather than a real bug. The authors evaluated Z-ranking on several bug checkers, and showed it to be up to 10 times more effective than random ranking at distinguishing real errors from false warnings. An interesting limitation of Z-ranking is that because of the "no success" hypothesis, it treats genuine bugs which result in failed checks on all program paths as false positives. The authors argue that testing should uncover such bugs, since they would manifest on any path to the location.

FeedbackRank [47] is another ranking technique proposed by Kremenek and Engler, which works by exploiting the locality of real bugs and false positives. The technique is based on the observation that both real bugs and false warnings tend to cluster. For example, real bugs may cluster because a programmer unaware of a correctness requirement violated it repeatedly while working on a particular function or module. False warnings might cluster because an incorrect static analysis decision resulted in cascading errors. FeedbackRank divides warnings into populations by locality: in the paper, the authors used functions, modules, and leaf directories (packages) as the basis for assigning warnings to populations. As the user classifies warnings as genuine or false, a Bayesian network models the influence of membership in the different kinds of populations on the likelihood that a warning is false. A novel feature of FeedbackRank is that the model is updated after each user classification; in this way, the model adapts to the particular bug distribution patterns of the code artifact being analyzed. The authors found that FeedbackRank performed between 2–8 times better than random ranking at distinguishing real bugs from false warnings. In addition, the authors show evidence that as an evolving code artifact is repeatedly analyzed and the new static analysis warnings ranked, the model should reach a steady state where the ranking effectiveness is maximized. A nice feature of FeedbackRank is that unlike Z-ranking, it does not depend in any way on the static analysis used to produce the warnings. Instead, it relies only on correlations among the warnings in different populations. The authors note that the locality is only one possible way to divide warnings into populations, and that other sources of correlation could be exploited.

We have not yet explored general ranking techniques in FindBugs. Instead, we have used ad-hoc techniques to suppress likely false positives.

7.3 Verification Techniques

The ultimate goal of techniques to ensure software quality is to guarantee the absence of bugs. Research has made significant progress towards this goal, although complete proofs of correctness are generally too expensive to apply to large systems. Partial verification techniques have been used successfully to show that programs have particular correctness properties.

Verification techniques are, in our opinion, complimentary to unsound bug checking tools like FindBugs. Verification of some program properties is a desirable goal. However, full verification of all correctness properties is not feasible in general, especially given the absence of precise specifications for most programs.

Axiomatic proof techniques [37] were prosed by Hoare as a means of establishing the correctness of programs. The axioms used are similar to those in arithmetic, with the important caveat that computer arithmetic is finite. Based on the syntactic forms of program statements, schemata may be applied to form axioms formally defining the meaning of those statements. An axiom defining a program statement S takes the form $P\{S\}Q$, where P is the assertion of the statement's preconditions and Q is the assertion of the statement's postconditions. Deductive rules allow introduction of new theorems based on those already proved; in this way, reasoning about the meaning of a single statement can be extended to a sequence of statements, loops, procedures, and eventually the entire program. Such proofs do not guarantee termination; instead, they guarantee a correct result in the event that the program does terminate. Hoare notes that the complexity of the schemata defining the meanings of program statements is directly related to the complexity of the underlying language. For this reason, axiomatic proof techniques would be difficult to apply to a language in which statements have side effects other than local assignment.

Extended Static Checking (ESC) [17, 49] is a system which uses formal proof techniques to find bugs in software. Its analysis is based on specifications, which take the form of program annotations. A variety of properties are checked, including procedure preconditions and postconditions, object invariants, temporal correctness properties of object interfaces, out of bounds array accesses, and incorrect synchronization. An automatic theorem prover is used to check the program's specifications against its implementation. The analysis is designed to be modular: invariants are specified for the interface to a particular class or module which can then be checked for all uses of the interface. In many respects, ESC can be seen as a verification technique, rather than a static analysis. However, ESC has several sources of imprecision which lead to unsoundness. For example, loop iterations are only analyzed to a finite depth. The approach we have taken in FindBugs contrasts strongly with ESC. ESC attempts to prove a variety of correctness properties over the entire program. In essence, ESC considers the program incorrect until proved correct. FindBugs attempts to identify code where an error is likely to exist—considering code to be correct unless there is compelling evidence to the contrary. We believe that this distinction is a very important factor in adoption for static analysis tools. Because errors are rare, programmers are not likely to embrace a tool that creates extra work in the common case.

CQual [31, 60, 32] uses *type qualifiers* as a means to annotate C programs to check particular correctness properties. Type qualifiers add properties to the existing programming language types; for example, a "tainted" qualifier might indicate data from an untrusted source that must be sanitized before being passed to a trusted sink. Qualified types have a subtype relationship with plain types. For example, for a type T, $T \subseteq \text{tainted } T$. CQual provides efficient automatic inference of type qualifiers based on constraints generated by program statements. Because types are familiar to programmers, and are efficient to check, type qualifiers represent a practical lightweight verification technique based on specifications. However, type qualifiers do not have the expressive power of full axiomatic semantics. The authors of CQual used it to infer possible locations where `const` qualifiers could be added

[31], to find format string vulnerabilities [60], and to find locking bugs in the Linux kernel [32].

ESP [15] is a system for partial verification of software. Rather than proving that the program is entirely correct, it checks that a single temporal correctness property holds. ESP uses a novel technique to avoid losing information at control joins by discarding states that do not affect the property being checked. This gives ESP path sensitivity without (in most cases) an exponential increase in the number of states that need to be tracked. The ESP algorithm is sound but not complete; it always rejects programs which do not have the property being verified, but may fail to prove the correctness of programs which are actually correct. The authors used ESP to prove that all file handles allocated in the GNU C compiler are used safely. In the future we may explore adopting a similar approach for making some of the intra-procedural analysis performed by FindBugs path-sensitive.

Model checking is a technique for exploring reachable states in a concurrent system for errors. It was initially developed for verification of hardware systems; recent research has applied it to software systems. The SLAM project [6, 5] applies model checking to *boolean abstractions* of programs. For predicates in a C program, boolean predicates are abstracted such that if a property is true of the boolean program, it will also be true of the original C program. The boolean abstraction of the original program is done incrementally; if the desired property cannot be proved, new predicates are added and the boolean program checked again. The authors used SLAM to verify that several Windows NT device drivers were free of certain kinds of locking errors.

7.4 Extraction of Specifications from Programs

Static analysis techniques to find bugs must be based on some notion of what distinguishes correct program behavior from incorrect behavior. Sometimes, *a priori* knowledge of language semantics are sufficient to suggest properties to check for. However, bug finding techniques benefit significantly from specifications of expected behavior, as can be seen in the number of bug patterns implemented in FindBugs related to incorrect use of APIs, and the number of serious occurrences of those patterns in production software. Recent work has developed techniques for automatically inferring specifications from software.

The authors of the MC system developed a technique called *belief inference* [22] for deriving system-specific rules. Belief inference is based on the observation that almost all program statements have a set of beliefs about their preconditions and postconditions. For example, dereferencing a pointer implies the belief that the pointer is non-null. Using belief inference, the authors of MC were able to develop two general strategies for automatically inferring errors. First, *consistency checking* works by flagging pairs of operations whose beliefs contradict. One of the conflicting operations must be an error, even if it is not known which one is incorrect. *Statistical analysis* works by generating a number of hypotheses and checking the code for evidence to support or refute the hypotheses. An example hypothesis is "variable v is protected by lock l." A statistical technique, *z-ranking*, is used to rank the confidence that a particular violation of a hypothesis is an error. Through checkers based on belief inference techniques, the authors were able to find hundreds of bugs in Linux and OpenBSD, some of which were violations of hypotheses about program correctness that the authors did not know in advance.

Daikon [24, 23] is a system for inferring likely invariants from the dynamic behavior of a program. At procedure entry and exit points, and (optionally) loop heads, Daikon infers candidate invariants. These may take the form of expressions of equality, inequality, or relations between program variables and constants or other variables, sequence ordering, sequence relations, or sequence membership. As the program executes, the values of program variables are recorded. As a post-processing step, candidate invariants are checked, and if falsified, removed from consideration. The invariants inferred by Daikon can be used as a first step in constructing a correctness proof using axiomatic techniques. The authors used Daikon to generate specifications to be checked by ESC/Java [53, 54]. For several small- and medium-sized Java programs, Daikon was able to infer about 90% of the invariants needed for the programs to pass the checker without warnings. This approach is noteworthy in that it uses the dynamic behavior of a program to infer a significant number of the specifications and invariants needed to verify aspects of its correctness.

Houdini [30] is an "annotation assistant" for ESC/Java. Like Daikon, it proposes candidate invariants at certain program points. Rather than using dynamic behavior to eliminate false invariants, it uses ESC/Java's theorem prover to prove or refute the invariants. Based on the theorem prover's output, refuted annotations are removed and the theorem prover run again. The process terminates when all of the annotations added by Houdini are accepted by the theorem prover. Compared to Daikon, Houdini has the advantage of not requiring a test suite in order to work.

The authors found that Houdini significantly reduced the amount of effort needed to annotate a program to successfully pass through ESC/Java with no warnings.

Specification mining [2] is a technique for extracting temporal correctness properties from traces recording uses of an API or abstract data type. The specifications produced could be used to find bugs using model checking or static analysis. Traces are produced by instrumenting the functions or methods implementing the API or ADT to record the name of the call and the identities of data objects. Specification mining requires that the specifications for interaction with the API or ADT have the form of a regular language. Since in general many APIs do not have this property, specification mining extracts *interaction scenarios*, which are subtraces which are regular. In addition to representing protocol states, the scenarios also convey data dependences based on the objects participating in the interaction. The interaction scenarios are fed to a probabilistic finite state automaton learner, which constructs PFSAs from the scenarios. The authors used specification mining to successfully extract protocol state machines for Unix socket calls and the X Window System selection API.

Chapter 8

Conclusions

In our work, we have made progress toward characterizing the software defects that can be found using simple static analysis techniques in programs written in type-safe object-oriented languages, such as Java. Our hypothesis that many bugs share common features and arise from similar causes has proved to be a productive way to explore the space of easily-detectable defects. By starting from real examples of bugs, and then researching analysis techniques to find them, we have developed a static analysis tool, FindBugs, which has found a large number of bugs in production software artifacts. We have conducted empirical studies to measure the effectiveness and accuracy of FindBugs on both production software and student programming projects. For student projects, we have been able to determine which bugs are *not* found by static analysis, and discovered many possible ways to improve the effectiveness and accuracy of the underlying analysis. FindBugs has been widely adopted by commercial organizations and open source projects. Given this success, it is clear that simple, bug-driven static analysis can play a useful role in maintaining and improving software quality.

At a higher level, we have gained a better understanding of why bugs occur in the first place, and have documented many common causes of errors. These insights suggest possible language or API improvements. In addition, characterizing defects that can be found using simple analysis techniques provides a context for work on more sophisticated analysis techniques.

We have several ideas for future work. We would like to continue our study of student programming projects, in order to find out if they contain more bugs that would be easy targets for static analysis. We would also like to investigate application-specific bugs patterns. Our work so far has focused on generic bug patterns that could affect any software. However, all software contains internal APIs and invariants which are not checked for correct use. Bugs arising from misuse of internal invariants would be a good target for static analysis. Given the success of null pointer annotations at finding bugs in student projects, we feel it would be worth investigating other uses of annotations. Another topic for future work is studying the benefits and costs of using more sophisticated analysis. Many commercial static analysis tools are advertised as performing deep, inter-procedural analysis. It would be interesting to see how many additional bugs they find. Finally, we would like to fully integrate FindBugs into the development process of a large software project. This would allow us to find out with greater confidence which bugs found by static analysis are judged to be worth fixing. It might also shed some light on the relative effectiveness of static analysis at finding bugs, as opposed to other techniques, such as testing. This would allow us to gain a better understanding of the economic costs and benefits of using static analysis to find bugs.

Appendix A

Description of Bug Patterns

This appendix describes each bug pattern listed in Section 2.2.

A.1 Correctness Patterns

AM: Empty jar or zip file entry. A loop creates Jar or Zip file entries without writing any data to those entries.

BC: Impossible cast. A cast is guaranteed to fail, or an `instanceof` check is guaranteed to evaluate to false. This bug is surprisingly common in code written by novice programmers.

BIT: Incompatible bit masks. An expression involving a comparison of values computed using bitwise arithmetic is guaranteed to evaluate to a boolean constant. If the boolean is used for control flow, dead code may result. Such errors may indicate that the wrong constant and/or operators were used in the expression.

BRSA: Bad ResultSet access. A value is fetched from or stored to a `ResultSet` object using an index value of 0. Because `ResultSet` values are indexed starting at one, an exception will be thrown.

CN: Class implements Cloneable, but does not implement clone(). Classes which implement the `Cloneable` interface should override the `clone()` method in order to make it visible to callers.

CN: Clone method does not call super.clone(). All `clone()` methods should delegate to their parent class's `clone()` method in order to create the new object. This serves two purposes: it guarantees that an object of the correct runtime type will be created, and it also guarantees that superclass fields will be initialized correctly.

Co: Covariant compareTo() method. Classes which implement the `Comparable` interface must define a `compareTo()` method which takes an argument of type `Object`. Classes which implement a covariant version of `compareTo()` cannot be called through the `Comparable` interface.

DE: Dropped exception. A method with an empty exception handler may unintentionally discard exceptions which should be re-thrown, logged, or reported to the user.

DLS: Dead local store. A value is computed and stored in a local variable, but the stored value is never used. If the stored value is an object, it may be the case that the object was intended to be used in some way, but due to an oversight was not used.

DMI: Passing invalid constant value for a month. Methods which take an integer representing a month (such as the `set()` methods of `java.util.Calendar`) require the value to be in the range 0..11, rather than the more usual range 1..12.

Dm: Method calls runFinalizersOnExit(). Calling runFinalizersOnExit() (in either the `System` or `Runtime` classes) may cause finalizers to be executed on live objects, resulting in state corruption, deadlocks, and many other problems.

EC: Invoking equals() on an array object. Array objects inherit the `equals()`

method from the base `Object` class, which works by checking reference equality. Programmers may incorrectly expect `equals()` to compare array contents.

EC: Calling equals() on unrelated types. The contract for the `equals()` method states that it should return `false` if the types of the objects compared do not match exactly. If the argument to `equals()` is an object which can be statically determined to be of a different type, then it should be guaranteed to return false at runtime.

Eq: Covariant equals() method. A covariant `equals()` method does not override the base `Object.equals()` method, and thus cannot be called from container classes and other contexts where the object is referred to by a supertype reference.

FE: Equality comparison of floating point values. Code that compares floating point values using the `==` or `!=` operators may produce unexpected results at runtime because these operators are inherently imprecise. Most comparisons of floating point values should be done using the `<`, `>`, `<=`, and `>=` operators.

FI: Explicit invocation of finalizer. Finalizer methods should not be called directly; they should only be executed by the virtual machine.

FI: Finalizer does not call super.finalize(). Subclasses should ensure that superclass finalizer actions are performed by calling `super.finalize()` in their `finalize()` methods.

HE: Equal objects should have equal hashcodes. Classes must ensure that objects that compare as equal using the `equals()` method must have the same hash code. Otherwise, they will not work correctly in `Hashtable` and `HashMap` objects.

ICAST: Invalid integer shift. An integer shift should be within the range 0..31.

ICAST: Integer division result cast to double. In integer division, the result is truncated. If the result of an integer division is cast to double, this suggests that the lost precision was meant to be retained.

ICAST: Integer value cast to double, then passed to Math.ceil(). Because an integer value has no fractional part, the call to `Math.ceil()` is effectively a no-op. This suggests that the integer value should have been a float or double.

IL: A container is added to itself. Adding a reference to a container object to itself will result in a stack overflow if the `hashCode()` method is called on the container, since the hash code of a container depends on the hash code of its elements.

IL: Infinite recursive loop. A method invokes itself unconditionally, or invokes itself recursively passing the original parameters as arguments. This will result in a stack overflow exception.

IM: Integer multiplication of result of integer remainder. Multiplying the result of an integer remainder by an integer constant may indicate operator precedence confusion.

IMSE: Dubious catching of IllegalMonitorStateException. A correct locking discipline should always ensure that monitor operations (such as `wait()` and `notify()`) are only performed when the thread holds a lock on the monitor. Any time code which explicitly catches `IllegalMonitorStateException` is seen, it is almost certain to be compensating for an error elsewhere, and may indicate a general misunderstanding of the semantics of `wait()` and `notify()`.

IP: A parameter is dead upon entry to method but overwritten. While it is arguably legitimate to conditionally assign a value to a parameter, it is poor style to write to a parameter whose original value is never read. Often, such cases are unintentional, and indicate an error.

It: Iterator next() method can't throw NoSuchElementException. The `next()` method of an `Iterator` class cannot throw `NoSuchElementException`, which it is required to do if there are no more elements to return. This may mean that the iterator object can walk off the end of a data structure.

J2EE: Store of non serializable object into HttpSession. An object which does not implement `Serializable` is stored in an `HttpSession`. If the session is passivated, the object will not be stored.

NP: Null pointer dereference in method. A variable containing the null value is dereferenced, which results in a `NullPointerException` being thrown.

NP: Possible null pointer dereference in method. A variable which may contain a null value is dereferenced. If full branch coverage of the method is possible, a `NullPointerException` can be thrown.

NP: equals() method does not check for null parameter. An `equals()` method dereferences its parameter unconditionally. However, the contract defined by `Object.equals()` requires all `equals()` methods to return `false` if passed a null value.

NP: Immediate dereference of the result of readLine(). The `readLine()` method in `BufferedReader` returns `null` when there are no more lines to read. Dereferencing the return value of `readLine()` without first checking to ensure that it is not null could result in a `NullPointerException`.

NP: Passing null for unconditionally dereferenced parameter. A null value is passed to a method which will dereference it unconditionally, resulting in a `NullPointerException` at runtime.

NP: Read of unwritten field. A value is loaded from a field and dereferenced, even though no value is ever written to that field. This will result in a `NullPointerException`.

NS: Questionable use of non-short-circuit logic. A non-short-circuiting bitwise `&` or `|` operator is used with boolean values, rather than integer values. Often, the programmer intended to use a short-circuiting boolean operator (`&&` or `||`).

Nm: Class defines equal(), not equals(). Defining a method called “equal” is usually a failed attempt to override `equals()`.

Nm: Confusing method names. Two method names differ only by capitalization. This may indicate a failed attempt to override a method in a superclass or interface.

Nm: Class defines hashcode(), not hashCode(). An all-lower-case `hashcode()` method does not override `Object.hashCode()`.

Nm: Class defines tostring(), not toString(). An all-lower-case `tostring()` method does not override `Object.toString()`.

ODR: Method may fail to close database resource. A method creates a database resource, such as a `ResultSet`, but does not close it on all paths out of the method. This may degrade application performance, or cause the application to run out of handles to database resources.

OS: Method may fail to close stream. A method opens an input or output stream, but does not close it on all paths out of the method. This may cause the application to run out of file handles, and if the stream is buffered, could result in data not to be written.

QF: Complicated, subtle or wrong increment in for-loop. The variable incremented by a for loop is different than the variable initialized and checked by the loop. This may cause the loop to fail to terminate.

RC: Suspicious reference comparison. Two references of a String, Integer, or similar class are compared by reference equality. Usually, instances of these classes should be compared using the `equals()` method, which compares the contents of the objects for equality.

RCN: Redundant comparison of non-null value to null. A value known to be non-null is compared to null. Although sometimes this is just defensive programming, it might indicate more serious confusion on the part of the programmer.

RCN: Redundant comparison of two null values. A value known to be null is compared to null. Although sometimes this is just defensive programming, it might indicate more serious confusion on the part of the programmer.

RCN: Nullcheck of value previously dereferenced. A value which is not known to be definitely non-null is dereferenced and then checked for null. This often indicates that the code was modified incorrectly by inserting a dereference before a null check. If it is possible for a value to be null, it should be checked *before* the dereference.

RE: Invalid syntax for regular expression. A `Pattern` object is created from a constant string which does not encode a valid regular expression, resulting in a `PatternSyntaxException` being thrown at runtime.

RR: Method ignores results of InputStream.read(). A `read()` method is called on an input stream, passed a byte array, but the return value is not checked. The call may return fewer bytes than requested, and if the caller does not check the return value, it may get out of sync with the stream, and uninitialized data in the byte array may be used.

RR: Method ignores results of InputStream.skip(). Like `read()`, the `skip()` method of `InputStream` classes may return fewer bytes than requested, and needs to be checked by the caller.

RV: Random value from 0 to 1 coerced to 0. When a random floating point value in the range 0..1 is coerced to an integer, the result is certainly 0. This may indicate that the `Random.nextInt(n)` method should have been called.

RV: Checking if result of String.indexOf() is positive. The `indexOf()` methods of the `String` class return a negative number when a matching substring is not found. Checking to see if this value is positive does not handle the case where a match is found at the beginning of the string.

RV: Discarding result of readLine() after checking if it is nonnull. After checking that the value returned from `readLine()` is non-null, the value is not used.

RV: Method ignores return value. Many methods in the core Java libraries only serve a useful purpose if their return value is checked. For example, all of the methods which transform a `String` object return a new object as a result, leaving the original unmodified. Failing to check the return value of such methods often means that the programmer did not understand this subtlety.

SA: Self assignment of field. A value is read from a field, and then immediately assigned to the same field. This almost certainly indicates a typo.

Se: Instance field in serializable class not serializable. A class is declared as `Serializable`, but one or more of its instance fields are not of a type guaranteed to be serializable. If a non-serializable object is stored in one of these fields, the object cannot be properly serialized.

Se: serialVersionUID declared incorrectly. A `serialVersionUID` field either is not of type `long`, is not static, or is not public. As such, it does not actually declare a serial version UID for purposes of serialization.

Se: Parent of Serializable class has no void constructor. A class implements the `Serializable` interface and its superclass does not. When such an object is deserialized, the fields of the superclass need to be initialized by invoking the void constructor of the superclass. Since the superclass does not have one, serialization and deserialization will fail at runtime.

Se: Externalizable class has no void constructor. A class implements the `Externalizable` interface, but does not define a void constructor. When externalizable objects are deserialized, they first need to be constructed by invoking the void constructor. Since this class does not have one, serialization and deserialization will fail at runtime.

SnVI: Class is Serializable, but doesn't define serialVersionUID. A class implementing `Serializable` does not define a serial version UID, so the virtual machine will create an implicit UID as a hash of the fields and methods of the class. However, because this hash includes synthetic fields and methods, its value depends on the source to bytecode compiler used, creating compatibility problems if the class is recompiled.

UCF: Useless control flow. The targets of both the `if` and `else` branches of an `if` statement are the same. A common typo (putting a semicolon after the guard of an `if` statement) is often the cause.

UR: Uninitialized read of field in constructor. An instance field is read in a constructor before any value has been assigned to the field; this will result in a `null` value being read. This may mean that the programmer forgot to assign a value to field, possibly due to typo.

UwF: Field only ever set to null. All writes to a field set its value to `null`. This may indicate that a constructor or initialization method was supposed to initialize the field, but did not.

A.2 Multithreaded Correctness Patterns

DC: Double check of field. Programmers often use the double-checked locking pattern [59] to lazily initialize objects, in order to avoid acquiring a lock every time the object is accessed. However, because of compiler and processor optimizations allowed by the Java Memory Model [40], the object may not be seen as fully initialized.

Dm: Monitor wait() called on Condition. The `wait()` method is called on a `java.util.concurrent.Condition` object. The programmer probably meant to call `await()`.

Dm: A thread was created using the default empty run method. A `Thread` object is created, but no `Runnable` object is passed to the thread, nor is an overloaded `run()` method defined. Therefore, the thread will do nothing when it is started.

IS2: Inconsistent synchronization. A field of a class is sometimes, but not always, accessed without a lock held on the object. This suggests that the programmer meant instances of the class to be safe for access by multiple threads (using the common idiom that a lock is held on the object while its mutable state is accessed); however, some accesses of the field were inadvertently left unsynchronized. As a result, the code may contain data races.

JLM: Synchronization performed on java.util.concurrent Lock. Ordinary monitor synchronization is performed on a `java.util.concurrent.Lock` object. It is likely that the programmer was confused, since the purpose of `Lock` objects is to allow explicit calls to `lock()` and `unlock()`.

LI: Lazy initialization of static field. A static field is initialized lazily. If the code is reachable by multiple threads at runtime, they may see an incompletely initialized object, because no synchronization is performed.

ML: Method synchronizes on an updated field. A method synchronizes on an object referenced from a field which is written more than once. This is unlikely to have useful semantics, since different threads may be synchronizing on different objects.

MWN: Mismatched notify(). `Object.notify()` is called on an object which is not locked. An `IllegalMonitorStateException` will be thrown.

MWN: Mismatched wait(). `Object.wait()` is called on an object which is not locked. An `IllegalMonitorStateException` will be thrown.

NN: Naked notify in method. A call to `notify()` or `notifyAll()` was made without any (apparent) accompanying modification to mutable object state. In general, calling a notify method on a monitor is done because some condition another thread is waiting for has become true. However, for the condition to be meaningful, it must involve a heap object that is visible to both threads.

No: Using notify() rather than notifyAll() in method. A method calls `notify()` rather than `notifyAll()`. Java monitors are often used for multiple conditions. Calling `notify()` only wakes up one thread, meaning that the thread woken up might not be the one waiting for the condition that the caller just satisfied.

RS: Class's readObject() method is synchronized. A serializable class defines a `readObject()` which is synchronized. By definition, an object created by deserialization is only reachable by one thread, and thus there is no need for readObject() to be synchronized. If the `readObject()` method itself is causing the object to become visible to another thread, that is an example of very dubious coding style.

Ru: Invoking run() on a thread. This method explicitly invokes `run()` on an object. In general, classes implement the `Runnable` interface because they are going to have their `run()` method invoked in a new thread, in which case `Thread.start()` is the right method to call.

SC: Constructor invokes Thread.start(). A constructor starts a thread. This is likely to be wrong if the class is ever extended/subclassed, since the thread will be started before the subclass constructor is started.

SP: Method spins on field. This method spins in a loop which reads a field. The compiler may legally hoist the read out of the loop, turning the code into an infinite loop.

SWL: Method calls Thread.sleep() with a lock held. This method calls `Thread.sleep()` with a lock held. This may result in very poor performance and scalability, since other threads may be waiting to acquire the lock.

TLW: Wait with two locks held. Waiting on a monitor while two locks are held may cause deadlock. Performing a wait only releases the lock on the object being waited on, not any other locks.

UG: Unsynchronized get method, synchronized set method. This class contains similarly-named get and set methods where the set method is synchronized and the get method is not. This may result in incorrect behavior at runtime, as callers of the get method will not necessarily see a consistent state for the object.

A.3 Malicious Code Patterns

EI: Returning reference to mutable object. Returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object. If instances are accessed by untrusted code, unchecked changes to the mutable object could compromise security or other important properties.

EI2: Incorporating reference to mutable object. A method stores a reference to an externally mutable object into the internal representation of the object. If instances are accessed by untrusted code, unchecked changes to the mutable object could compromise security or other important properties.

FI: Finalizer should be protected, not public. A class's finalize() method should have protected access, not public. Untrusted code could corrupt the state of the program by calling the finalizer method directly.

MS: Storing mutable object into static field. A method stores a reference to an externally mutable object into a static field. Unchecked changes to the mutable object could compromise security or other important properties.

MS: Static field could be modified by untrusted code. A mutable static field could be changed by malicious code or by accident from another package. Often, this situation arises because of a typo; the field was meant to be declared as final, but was not.

MS: Returning mutable array. A public static method returns a reference to an array that is part of the static state of the class. Any code that calls this method can freely modify the underlying array.

MS: Field is a mutable array. A final static field references an array and can be accessed by malicious code or by accident from another package. This code can freely modify the contents of the array.

MS: Field is a mutable Hashtable. A final static field references a `Hashtable` and can be accessed by malicious code or by accident from another package. This code can freely modify the contents of the `Hashtable`.

A.4 Performance Patterns

Dm: Method invokes dubious Boolean constructor. A method crates a new instance of a `Boolean` object. This is wasteful because `Boolean` is an immutable class. A better approach is to use the singleton `Boolean.TRUE` or `Boolean.FALSE` objects.

Dm: Method allocates a boxed primitive just to call toString(). A new boxed primitive object (`Integer`, `Boolean`, etc.) is created for the sole purpose of calling its `toString` method. Each of these classes contains a static version of `toString()` which converts a primitive value to a string directly.

Dm: Explicit garbage collection. A method invokes `System.gc()`, which causes an immediate garbage collection. Because a full garbage collection can be expensive, such calls can significantly harm performance, especially if they occur in a loop.

Dm: Method allocates an object, only to get the class object. A new object is constructed for the sole purpose of getting its runtime `Class` object. The `Class` object can be obtained more efficiently by accessing the `class` static pseudo-field.

Dm: Calling Random.nextDouble() to generate a random integer. Calling Random.nextDouble(), multiplying it by a constant, and casting the result to an `int` is less efficient that simply calling `Random.nextInt()`.

Dm: Invoking dubious new String(String) constructor. A method creates a new `String` object by invoking the constructor which takes another `String`. Since strings are immutable, creating a new identical `String` object serves no purpose, and waste time and memory.

Dm: Method invokes toString() method on a String. Calling `toString()` on a `String` object is a redundant operation.

Dm: Method invokes dubious new String() constructor. Creating a new empty `String` using the default constructor is unnecessary, and wastes memory; using the empty string constant (`""`) is a better idea, and is more clear.

FI: Empty finalizer method. A class defines an empty finalizer method. Because it has no effect, it should be deleted.

FI: Finalizer does nothing but call superclass finalizer. A class defines a finalizer method which does nothing but calls its parent class's finalizer. Because it has no effect, it should be deleted.

ITA: Method uses toArray() with zero-length array argument. Because the Java collection classes carry no type information about their contents, the `toArray()` method defined for collection classes must be passed an argument of the correct runtime array type. If this array is not large enough to hold the entire contents of the collection, a new array must be created. Passing a zero-length array to `toArray()` forces the creation of a new array object.

SBSC: Method concatenates strings using + in a loop. A string value is built in a loop by repeated concatenation using the `+` operator. This will likely require the creation of a new `StringBuffer` or `StringBuilder` on every loop iteration. In addition to the object-creation overhead, the buffer copying involved creates quadratic complexity instead of linear (as would be the case if `append()` is called on a single `StringBuffer` or `StringBuilder`.)

SIC: Inner class should be made static. An inner class does not use its embedded reference to the object which created it. This reference makes the instances of the class larger, and may keep the reference to the creator object alive longer than necessary. If possible, the class should be made static.

SS: Unread final instance field. A final instance field is never read because its value is a constant at compile time. It should be made static to avoid taking up space in the object.

UrF: Unread field. A field is never read, and should be removed from the class.

UuF: Unused field. A field is never read or written, and should be removed from the class.

Appendix B

Deployment Issues

In order to make the greatest impact on software quality, software engineering tools must be used as an ongoing part of the development process, so that bugs can be found as early as possible. For a static analysis tool to fit into the development process, several issues must be addressed:

- Usability
- Integration with existing tools and environments
- Handling false positives

B.1 Usability

Software developers have an enormous number of software engineering tools to choose from. SourceForge [62] hosts over 14,000 open-source projects classified in the “Software Development” category, and these compete with a wide variety of commercial tools. Developers are also notoriously overworked, and have little time to research and evaluate new tools. Therefore, in order to have a chance at being adopted, any new tool must offer an clear and immediate benefit.

We have not conducted a formal usability study for FindBugs. However, based on our own experience and feedback from users, we know of several factors which are important when a new static analysis tool is considered for adoption.

The tool should require minimal setup and configuration. Users have little patience for complex installation or configuration procedures. In particular, users should not have to compile the tool from source code (which is not uncommon for open-source software), nor should they be required to edit configuration files or environment variables in order for the software to run. While some configurability is beneficial, it is important that the default settings for the tool are appropriate for most users.

We have made sure that each FindBugs release can be executed immediately after its distribution archive is unpacked. Running FindBugs is as simple as launching the tool's GUI interface, choosing a set of Java class files, archives, or directories to analyze, clicking a button, and reviewing the output. We have received positive feedback about initial ease of use from users. One user writes:

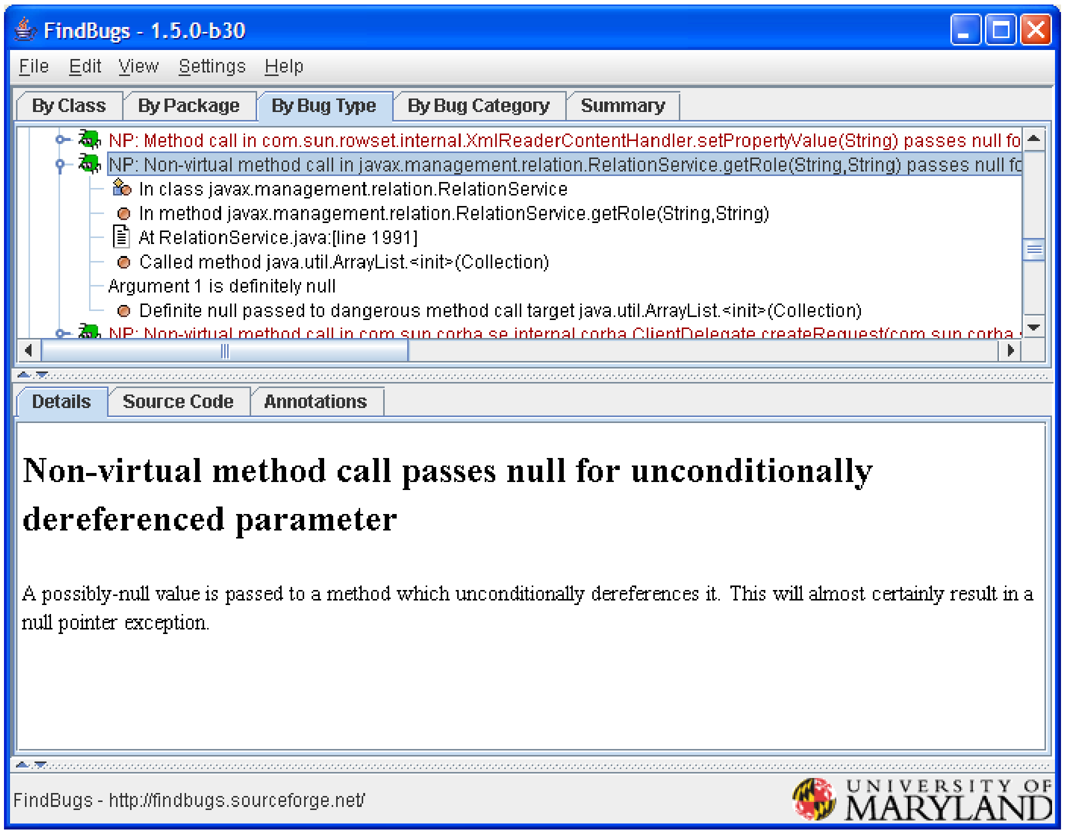
> I gave it a whirl on my company's product and found some interesting stuff. I like how easy it is to use, I just pointed it at the .jar file and also where the sources live and voila—free code review.

The information presented to the user must be clear and comprehensive. A useful static analysis tool cannot simply assert the existence of a bug; it must also provide enough detail to explain which parts of the code are incorrect, and why. For some bugs, this may be fairly simple: for example, "There is a null pointer dereference in Foo.java at line 200." For more complex or subtle bugs, such as race conditions, it is important to provide an explanation of the bug means, and to provide contextual details describing which parts of the code appear to be faulty.

FindBugs has a rich data model for warnings. The core of a warning is composed of a the bug pattern type and a warning priority. The bug pattern type specifies the bug pattern of which the code identified by the warning is a probable occurrence. Each bug pattern type has a one or two paragraph description, which is shown in the FindBugs GUI when a warning of that type is selected. The warning priority indicates the severity of the warning and the confidence that the warning represents a real bug. A high priority warning indicates high likelihood of a serious bug. Medium priority warnings are also likely to be bugs, but may not represent as severe a problem as high priority warnings. Low priority warnings are included for the sake of completeness, but generally do not indicate real errors. (By default, low priority warnings are not displayed, meaning users must explicitly request to see them.)

Each warning is decorated by a number of *warning attributes* which provide context information about the bug. Types of warning attributes include:

- Class: a class or interface
- Method: a method in a class or interface
- Field: a field of a class
- Source line: a range of one or more source lines
- Integer value



NP: Method call in com.sun.rowset.internal.XmlReaderContentHandler.setPropertyValue(String) passes null fo
NP: Non-virtual method call in javax.management.relation.RelationService.getRole(String,String) passes null fo
In class javax.management.relation.RelationService
In method javax.management.relation.RelationService.getRole(String,String)
At RelationService.java:[line 1991]
Called method java.util.ArrayList.<init>(Collection)
Argument 1 is definitely null
Definite null passed to dangerous method call target java.util.ArrayList.<init>(Collection)
NP: Non-virtual method call in com.sun.corba.se.internal.corba.ClientDelegate.createRequest(com.sun.corba

Non-virtual method call passes null for unconditionally dereferenced parameter

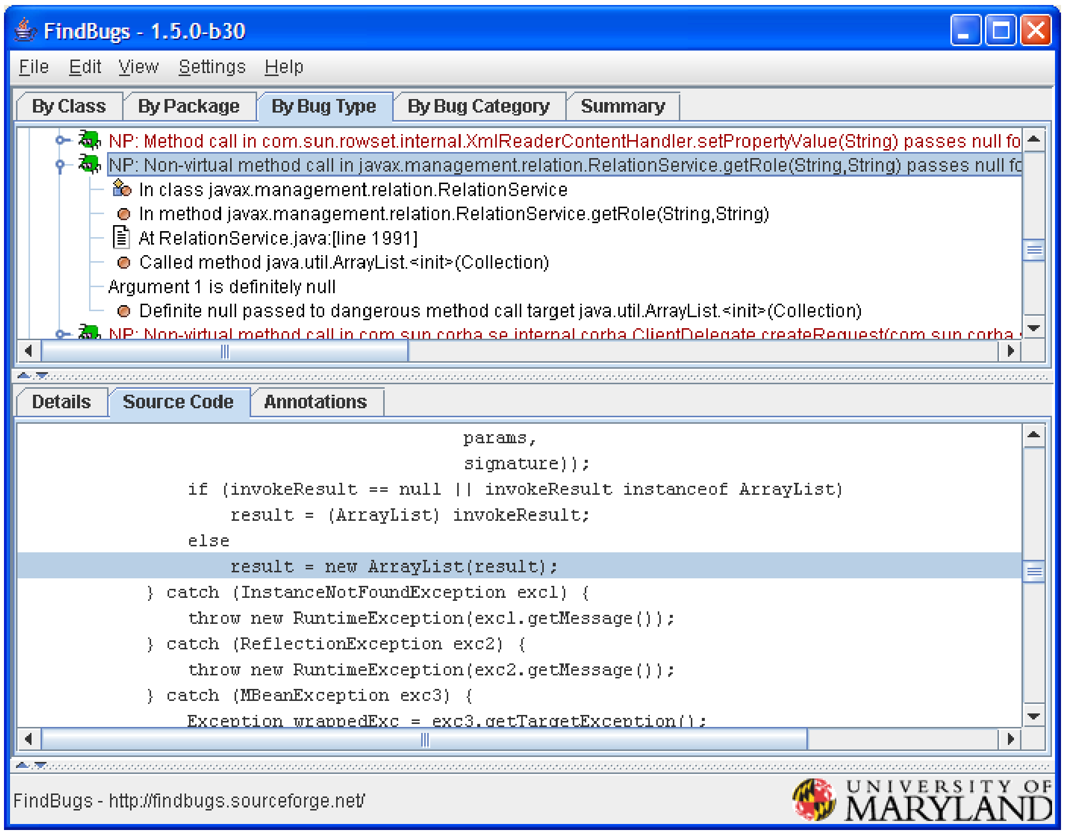
A possibly-null value is passed to a method which unconditionally dereferences it. This will almost certainly result in a null pointer exception.

FindBugs - http://findbugs.sourceforge.net/

Figure B.1: Screen shot of FindBugs GUI showing expansion of warning attributes

All warnings specify a class attribute. Warnings which identify likely bugs in executable code always have a method attribute, indicating the method in which the bug occurs, and also a source line attribute, indicating the affected source lines. (Internally, source line attributes also record the bytecode offsets of the affected instructions.)

Each warning attribute has a *role*, which may be used to elaborate the meaning of the specified code feature or analysis fact. For example, method attributes may have the role “called”, which indicates that the warning involves a call to the method. This allows warnings to distinguish between code features specifying the *location* of a warning and code features which are merely *referenced* by the warning.



NP: Method call in com.sun.rowset.internal.XmlReaderContentHandler.setPropertyValue(String) passes null fo
NP: Non-virtual method call in javax.management.relation.RelationService.getRole(String,String) passes null fo
In class javax.management.relation.RelationService
In method javax.management.relation.RelationService.getRole(String,String)
At RelationService.java:[line 1991]
Called method java.util.ArrayList.<init>(Collection)
Argument 1 is definitely null
Definite null passed to dangerous method call target java.util.ArrayList.<init>(Collection)
NP: Non-virtual method call in com.sun.corba.se.internal.corba.ClientDelegate.createRequest(com.sun.corba.

```
                                             params,
                                             signature));
        if (invokeResult == null || invokeResult instanceof ArrayList)
            result = (ArrayList) invokeResult;
        else
            result = new ArrayList(result);
    } catch (InstanceNotFoundException exc1) {
        throw new RuntimeException(exc1.getMessage());
    } catch (ReflectionException exc2) {
        throw new RuntimeException(exc2.getMessage());
    } catch (MBeanException exc3) {
        Exception wrappedExc = exc3.getTargetException();
```

FindBugs - http://findbugs.sourceforge.net/

Figure B.2: Screen shot of FindBugs GUI showing source code for warning

The FindBugs GUI displays warnings as elements in a tree view. The text of a warning item is a brief synopsis of the warning. Each warning may be expanded to display its attributes. Figures B.1 and B.2 show a warning and its attributes being displayed in the FindBugs GUI. The warning shown describes a possible inter-procedural null pointer dereference. The warning contains attributes specifying the class, method, and source lines of a method call site where a possibly null value is passed to a method which will unconditionally dereference it. The "Details" window shows a brief description of the bug, and the "Source Code" window displays the relevant source code. Another method attribute specifies which method is called at the call site: the user may click on this attribute to show the source code for the called method. An integer value attribute specifies which argument passed to the method is possibly null at the call site.

Users must be able to find the information they are most interested in. Static analysis tools must provide an effective way for the user to find the subset of warnings they are most interested in. In the FindBugs GUI, we have implemented navigation by class, package, bug pattern, and bug category. Users may also filter warnings by category and priority. These navigation features allow developers to quickly focus on particular warning types or code regions.

B.2 Tool Integration

Another important consideration for static analysis tools is integration with other development tools and environments. Of particular importance is making the tool easy to run from the developer's build system, so it can be integrated as a regular part of the development process.

FindBugs may be plugged into a development environment in several ways. First, it supports a batch mode interface, which may be invoked from a script. It also integrates with the ubiquitous Ant [3] build tool, and with the Maven [52] project management system. These interfaces make it easy to analyze code with FindBugs on demand or as part of a nightly or weekly build.

In addition to providing a convenient way to run a static analysis tool, it is critical to provide a way to store the warnings in a persistent form. FindBugs supports several output formats. In addition to a plain text format, FindBugs can emit warnings as XML [68]. The XML-based format fully preserves all information associated with each warning, including all of its attributes. This format may be read and written by the GUI interface: this makes it possible to save the results of static analysis for later inspection.

While XML is useful as generic format for storing structured data, it is not particularly well-suited as a presentation format to be read by humans. FindBugs includes an XSL stylesheet for converting the XML format to human-readable HTML. This allows FindBugs warnings to be viewed in any web browser.

In addition to batch mode tools, FindBugs also provides a plugin for the Eclipse Integrated Development Environment [19]. While the Eclipse plugin is not as full-featured as the other interfaces supported by FindBugs, it performs analysis continuously as source code is modified. This allows developers to see possible errors as soon as they are introduced.

B.3 Handling False Positives

As developers fix problems identified by static analysis, over time an increasingly large percentage of the warnings will be false positives. In order to avoid pestering the user with these warnings every time the tool is run, the tool must offer a way to suppress them.

It is worth noting that suppression of uninteresting warnings is an issue for all static analysis tools, and not just those which admit false warnings. While it is possible to engineer a static analysis so that all warnings reported are real bugs (according to some definition of correctness), in practice not all bugs will be considered worth fixing by developers.

A very simple approach to suppressing false positives, which we have implemented in FindBugs, is to provide developers a means of determining which warnings are new after code changes. Assuming that it is possible to precisely determine when a warning is "the same" from one version of the code to the next, limiting inspection to newly-introduced warnings ensures that developers will only need to inspect any warning once.

In practice, it is not always possible to determine when two warnings in different versions correspond to the same issue. This imprecision may cause some warnings to be wrongly classified as new, requiring developers to inspect them more than once over the lifetime of the software. It may also result in some warnings to be wrongly classified as old, resulting in warnings which are never inspected. However, as long as the matching technique is sufficiently accurate overall, this approach can be a simple and effective way of handling false positives.

We have also implemented source-level annotations [42] for the suppression of false warnings. Because these annotations are part of the source code, they have the advantage of being completely unambiguous. Once in place, they will continue to suppress the specified warnings even if the annotated class or method is renamed or modified. However, this advantage is also a limitation: if the code is modified, the suppressed warning or warnings may no longer exist, making the annotation unnecessary. The tool may assist the user in this situation by reporting suppression attributes which no longer apply to any warnings. Another problem with the source annotation approach is that annotations can only apply at the granularity of an entire method or class: this may result in some valid warnings being suppressed.

BIBLIOGRAPHY

[1] Eric Allen. *Bug Patterns In Java.* APress, 2002.

[2] Glenn Ammons, Rastislav Bodk, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16. ACM Press, 2002.

[3] Apache ant. `http://ant.apache.org`, 2005.

[4] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded Java. In *Proceedings of the 13th Australian Software Engineering Conference*, pages 68–75, August 2001.

[5] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[6] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *Lecture Notes in Computer Science*, 2031:268+, 2001.

[7] Kent Beck. *Extreme Programming Explained.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[8] Sleepycat Software: Products: Berkeley DB, Java Edition. `http://www.sleepycat.com/products/je.shtml`, 2005.

[9] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30:775–802, 2000.

[10] Checkstyle. `http://checkstyle.sourceforge.net`, 2005.

[11] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001.

[12] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[13] Roger F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the Usenix Conference on Domain-Specific Languages*, October 1997.

[14] CVS. `http://www.cvshome.org`, 2005.

[15] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68. ACM Press, 2002.

[16] Noopur Davis and Julia Mullaney. The team software process (TSP) in practice: A summary of recent results. Technical Report CMU/SEI-2003-TR-014, Carnegie Mellon Software Engineering Institute, September 2003.

[17] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Task Force Report59, Compaq Systems Research Center, December 1998.

[18] DrJava. `http://www.drjava.org`, 2005.

[19] Eclipse. `http://www.eclipse.org`, 2005.

[20] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.

[21] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252. ACM Press, 2003.

[22] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 57–72. ACM Press, 2001.

[23] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[24] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, June 2000.

[25] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53. ACM Press, 1996.

[26] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[27] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[28] Michael Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 13(3):182–211, 1976.

[29] Findbugs. `http://findbugs.sourceforge.net`, 2005.

[30] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. *Lecture Notes in Computer Science*, 2021:500+, 2001.

[31] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

[32] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.

[33] GNU Classpath. `http://www.gnu.org/software/classpath`, 2005.

[34] Google. `http://www.google.com`, 2005.

[35] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[36] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, June 2002.

[37] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[38] Institute of Electrical and Electronic Engineers, Inc. Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York City, New York, USA, 1995. also ISO/IEC 9945-1:1990b.

[39] Java(tm) 2 Platform, Standard Edition. `http://java.sun.com/j2se`, 2005.

[40] Java Specification Request (JSR) 133. Java memory model and thread specification revision. `http://jcp.org/jsr/detail/133.jsp`, 2004.

[41] Java Specification Request (JSR) 14. Add generic types to the Java programming language. `http://jcp.org/jsr/detail/14.jsp`, 2004.

[42] Java Specification Request (JSR) 175. A metadata facility for the Java (tm) programming language. `http://jcp.org/jsr/detail/175.jsp`, 2004.

[43] JBoss. `http://www.jboss.org`, 2005.

[44] jEdit. `http://www.jedit.org`, 2005.

[45] S. Johnson. Lint, a C program checker, Unix programmer's manual, AT&T Bell Laboratories, 1978.

[46] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[47] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *Proceedings of the SIGSOFT Foundations of Software Engineering Conference*, October 2004.

[48] Ted Kremenek and Dawson R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In Radhia Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 295–315. Springer, 2003.

[49] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Report 2000-002, Compaq Systems Research Center, October 2002.

[50] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[51] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 219–230. ACM Press, 2004.

[52] Apache maven. `http://maven.apache.org`, 2005.

[53] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, July 2001.

[54] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 232–242, July 2002.

[55] Pmd. `http://pmd.sourceforge.net`, 2005.

[56] William Pugh. The double checked locking is broken declaration. `http://www.cs.umd.edu/users/pugh/java/memoryModel/DoubleCheckedLocking.html`, July 2000.

[57] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, National Institute of Standards & Technology, 2002.

[58] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *Proceedings of the International Symposium on Software Reliability Engineering*, Saint-Malo, France, November 2004.

[59] Douglas Schmidt and Tim Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *3rd Annual Pattern Languages of Program Design Conference*, 1996.

[60] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[61] Smalllint. `http://st-www.cs.uiuc.edu/users/brant/Refactory/Lint.html`, 2005.

[62] Sourceforge.net. `http://sourceforge.net`, 2005.

[63] Jaime Spacco, David Hovemeyer, and William Pugh. An Eclipse-based course project snapshot and submission system. In *3rd Eclipse Technology Exchange Workshop (eTX)*, Vancouver, BC, October 24, 2004.

[64] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. Software repository mining with Marmoset: An automated programming project snap-

shot and testing system. In *Proceedings of the Mining Software Repositories Workshop (MSR 2005)*, St. Louis, Missouri, USA, May 2005.

[65] Nicholas Sterling. WARLOCK: A static data race analysis tool. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–106, January 1993.

[66] Security Across the Software Development Lifecycle Task Force. Improving security across the software development lifecycle. Task Force Report, National Cyber Security Partnership, April 2004.

[67] Yichen Xie and Dawson Engler. Using redundancies to find errors. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, November 2002.

[68] Extensible markup language (XML). `http://www.w3.org/XML`, 2005.