Finding Bugs is Easy [*]

David Hovemeyer and William Pugh
Dept. of Computer Science, University of Maryland
College Park, Maryland 20742 USA
{daveho,pugh}@cs.umd.edu

ABSTRACT

Many techniques have been developed over the years to automatically find bugs in software. Often, these techniques rely on formal methods and sophisticated program analysis. While these techniques are valuable, they can be difficult to apply, and they aren't always effective in finding real bugs.

Bug patterns are code idioms that are often errors. We have implemented automatic detectors for a variety of bug patterns found in Java programs. In this paper, we describe how we have used bug pattern detectors to find serious bugs in several widely used Java applications and libraries. We have found that the effort required to implement a bug pattern detector tends to be low, and that even extremely simple detectors find bugs in real applications.

From our experience applying bug pattern detectors to real programs, we have drawn several interesting conclusions. First, we have found that even well tested code written by experts contains a surprising number of obvious bugs. Second, Java (and similar languages) have many language features and APIs which are prone to misuse. Finally, that simple automatic techniques can be effective at countering the impact of both ordinary mistakes and misunderstood language features.

1. INTRODUCTION

Few people who develop or use software will need to be convinced that bugs are a serious problem. Automatic techniques and tools for finding bugs offer tremendous promise for improving software quality. In recent years, much research has investigated automatic techniques to find bugs in software (e.g., [11, 24, 28, 46, 6, 9, 44, 29, 22]). Some of the techniques proposed in this research require sophisticated program analysis. This research is valuable, and many interesting and useful analysis techniques have been proposed as a result. However, these techniques have generally not found their way into widespread use.

In our work, we have investigated simple static analysis techniques for finding bugs based on the notion of *bug patterns*. A bug pattern is a code idiom that is likely to be an error. Occurrences of bug patterns are places where code does not follow usual correct practice in the use of a language feature or library API. Automatic detectors for many bug patterns can be implemented using relatively simple static analysis techniques. In many ways, using static analysis to find occurrences of bug patterns is like an automated code inspection. We have implemented a number of automatic bug pattern detectors in a tool called FindBugs.

In this paper, we will describe some of the bug patterns our tool looks for, and present examples of bugs that our tool has found in several widely used applications and libraries. We hope that the obvious and embarrassing nature of these bugs will convince you of the need for wider adoption of automatic bug finding tools. We also present empirical results which classify (for selected bug patterns) the percentage of warnings reported by the tool that indicate real bugs. We will argue that although some of the bug pattern detectors we evaluated produce false warnings, they produce enough genuine examples of bugs to make the tool useful in practice.

We have two primary motivations in this work. The first is to raise awareness of the large number of easily-detectable bugs that are not caught by traditional quality assurance techniques: we believe that developers who employ rigorous quality assurance practices will be surprised at the number of bugs that elude unit testing and code inspections. The second is to suggest possibilities for future research; specifically, that much more work needs to be done on finding ways to integrate automatic bug finding techniques into the development process in a way that best directs developers towards the bugs that are most profitable to fix, while not distracting them with irrelevant or inaccurate warnings.

The structure of the rest of the paper is as follows. In Section 2, we discuss the general problem of finding bugs in software, and the strengths and weaknesses of various approaches. In Section 3, we describe the implementation of FindBugs, our bug checker for Java based on bug patterns. In Section 4, we describe some of the bug pattern detectors implemented in FindBugs, and show some examples of bugs they found in several widely used applications and libraries. In Section 5, we evaluate the effectiveness of our bug pattern detectors on several real programs. In Section 6, we offer observations from our experiences and some of our users' experiences putting bug pattern detection into practice. In Section 7 we offer some conclusions, and describe possibilities for future work. In Section 8, we describe related work.

2. TECHNIQUES FOR FINDING BUGS

There are many possible ways to find bugs in software. Code inspections can be very effective at finding bugs, but

have the obvious disadvantage of requiring intensive manual effort to apply. In addition, human observers are vulnerable to being influenced by what a piece of code is *intended* to do. Automatic techniques have the advantage of relative objectivity.

Dynamic techniques, such as testing and assertions, rely on the runtime behavior of a program. This can be both an advantage and a limitation. The advantage is that dynamic techniques, by definition, do not consider infeasible paths in a program. The disadvantage is that they are generally limited to finding bugs in the program paths that are actually executed. Anyone who has tried to construct a test suite for a program of significant complexity knows the difficulty of achieving high statement or branch coverage. In addition, the time required to run a full suite of tests can be too great to run frequently.

In contrast to dynamic techniques, static techniques can explore abstractions of all possible program behaviors, and thus are not limited by the quality of test cases in order to be effective. Static techniques range in their complexity and their ability to identify or eliminate bugs. The most effective (and complex) static technique for eliminating bugs is a *formal proof* of correctness. While the existence of a correctness proof is perhaps the best guarantee that the program does not contain bugs, the difficulty in constructing such a proof is prohibitive for most programs. *Partial verification* techniques have been proposed. These techniques prove that some desired property of a program holds for all possible executions. Such techniques may be complete or incomplete; if incomplete, then the analysis may be unable to prove that the desired property holds for some correct programs. Finally, *unsound* techniques can identify "probable" bugs, but may miss some real bugs and also may emit some inaccurate warnings.

2.1 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. Some style rules, such as "always put constants on the left hand side of comparisons", and "don't use assignment statements in the guard of an if statement", may help prevent certain kinds of bugs. However, violations of those style guidelines are not particularly likely to be bugs.

Another way to think about the distinction between style checkers and bug checkers is that violations of style guidelines only cause problems for the developers working on the software. Warnings produced by a bug checker may represent bugs that will cause problems for the users of the software.

Tools that focus mainly on checking style, such as PMD [38] 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 bug patterns 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 don't 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.

3. THE FINDBUGS TOOL

In this section we briefly describe the implementation of the FindBugs tool. FindBugs is open source, and source code, binaries, and documentation may be downloaded from `http://findbugs.sourceforge.net`.

Currently, FindBugs contains detectors for about 50 bug patterns. All of the bug pattern detectors are implemented using BCEL [7], an open source bytecode analysis and instrumentation library. The detectors are implemented using the Visitor design pattern; each detector visits each class of the analyzed library or application.

The implementation strategies used by the detectors can be divided into several rough categories:

- **Class structure and inheritance hierarchy only**. Some of the detectors simply look at the structure of the analyzed classes without looking at the code.

- **Linear code scan**. These detectors make a linear scan through the bytecode for the methods of analyzed classes, using the visited instructions to drive a state machine. These detectors do not make use of complete control flow information; however, heuristics (such as identifying the targets of branch instructions) can be effective in approximating control flow.

- **Control sensitive**. These detectors make use of an accurate control flow graph for analyzed methods.

- **Dataflow**. The most complicated detectors use dataflow analysis to take both control and data flow into account. An example is the null pointer dereference detector.

None of the detectors make use of analysis techniques more sophisticated than what might be taught in an undergraduate compiler course. The detectors that use dataflow analysis are the most complex; however, we have implemented a framework for dataflow analysis which moves most
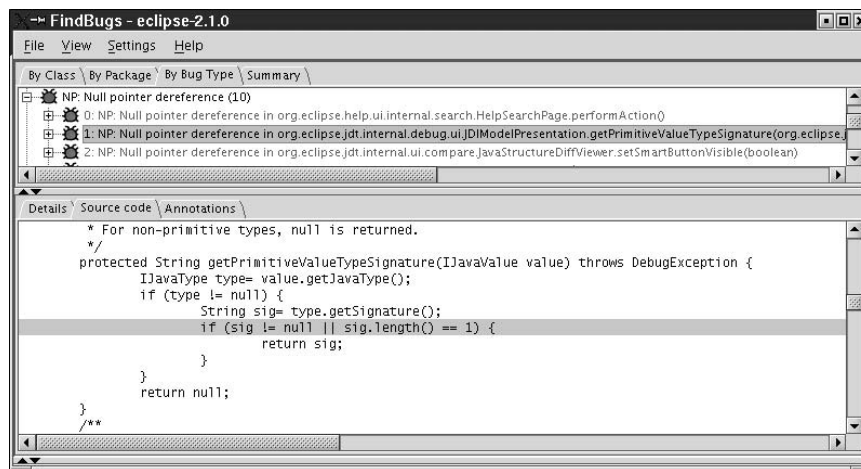
Figure 1: Screenshot of FindBugs.

of the complexity out of the detectors themselves. The most complex detector has about 1000 lines of Java source code (including blank lines and comments). Almost half of the detectors have less than 100 lines of source code.

We have implemented several front ends to FindBugs:

- A simple batch application that generates text reports, one line per warning.

- A batch application that generates XML reports.

- An interactive tool that can be used for browsing warnings and associated source code, and annotating the warnings. The interactive tool can read and write the XML reports. A screenshot of the interactive tool is shown in Figure 1.

We have also developed several tools for manipulating the XML reports. These tools allow us to perform actions such as comparing the difference in warnings generated on two different versions of an application.

FindBugs users have contributed two additional front-ends: a plugin that integrates FindBugs into the Eclipse [18] IDE, and a task for running FindBugs from the Apache Ant [2] build tool.

4. BUG PATTERN DETECTORS

For space reasons, we will only discuss a handful of the bug pattern detectors implemented in FindBugs. Each bug pattern is identified by a short "code", which will be used in the evaluation in Section 5. A table summarizing the bug patterns and detectors described in this paper is shown in Figure 2.

Broadly speaking, each detector falls into one or more of the following categories:

- Single-threaded correctness issue

- Thread/synchronization correctness issue

- Performance issue

- Security and vulnerability to malicious untrusted code

Code	Description
CN	Cloneable Not Implemented Correctly
DC	Double Checked Locking
DE	Dropped Exception
EC	Suspicious Equals Comparison
Eq	Bad Covariant Definition of Equals
HE	Equal Objects Must Have Equal Hashcodes
IS2	Inconsistent Synchronization
MS	Static Field Modifiable By Untrusted Code
NP	Null Pointer Dereference
NS	Non-Short-Circuit Boolean Operator
OS	Open Stream
RCN	Redundant Comparison to Null
RR	Read Return Should Be Checked
RV	Return Value Should Be Checked
Se	Non-serializable Serializable Class
UR	Uninitialized Read In Constructor
UW	Unconditional Wait
Wa	Wait Not In Loop

Figure 2: Summary of selected bug patterns.

```
// jdk1.5.0, build 59
// javax.sql.rowset.spi,
// SyncFactory.java, line 325

if(syncFactory == null){
    synchronized(SyncFactory.class) {
        if(syncFactory == null){
            syncFactory = new SyncFactory();
        } //end if
    } //end synchronized block
} //end if
```

Figure 3: Example of double checked locking.

4.1 Cloneable Not Implemented Correctly (CN)

This pattern checks for whether a class implements the Cloneable interface correctly. The most common violation is to not call `super.clone()`, but rather allocate a new object by invoking a constructor. This means that the class can never be correctly subclassed (and the warning is silenced if the class is final), since calling clone on a subtype will not return an instance of that subtype[1].

4.2 Double Checked Locking (DC)

Double checked locking is a design pattern intended for thread safe lazy initialization [42]. An example of double checked locking is shown in Figure 3.

Unfortunately, the double checked locking pattern assumes a sequentially consistent memory model, which isn't true in any major programming language. In Figure 3 it is possible that the writes initializing the `SyncFactory` object and the write to the `syncFactory` field could be reordered (either by the compiler or the processor).

It is now accepted that the double checked locking pattern is broken [39] as originally stated. In Java, it is possible to fix the double checked locking pattern simply by making the checked field volatile.

4.3 Dropped Exception (DE)

This detector looks for a try-catch block where the catch block is empty and the exception is slightly discarded. This often indicates a situation where the programmer believes the exception cannot occur. However, if the exception *does* occur, silently ignoring the exception can create incorrect anomalous behavior that could be very hard to track down.

4.4 Suspicious Equals Comparison (EC)

This detector uses intraprocedural dataflow analysis to determine when two objects of types known to be incomparable are compared using the `equals()` method. Such comparisons should always return false, and typically arise because the wrong objects are being compared.

4.5 Bad Covariant Definition of Equals (Eq)

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()`:

---

[1]Having all `clone()` methods call `super.clone()` ensure that they all delegate object creation to `Object.clone()`, which automatically creates a new object of the correct class.

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

This covariant version of `equals()` does not override the version in the `Object` class, which may lead to unexpected behavior at runtime, especially is 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 it looks correct, and in circumstances where the class is accessed through references of the class type (rather than a supertype), it will work correctly. However, the first time it is used in a container, mysterious behavior will result. For these reasons, this type of bug can elude testing and code inspections.

Detecting instances of this bug pattern simply involves examining the method signatures of a class and its superclasses.

4.6 Equal Objects Must Have Equal Hashcodes (HE)

In order for Java objects to be stored in `HashMap`s and `HashSet`s, 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.

Automatically verifying that a given class maintains the invariant that equal objects have equal hashcodes would be very difficult. 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 analysis of method signatures and the class hierarchy.

4.7 Inconsistent Synchronization (IS2)

Many Java classes, both in runtime libraries and applications, are designed to be safe when accessed by multiple threads. The most common way to ensure thread safety is to synchronize on the receiver object ("`this`") when accessing or modifying shared state.

A common category of mistakes in implementing thread safe objects is to allow access to mutable fields without synchronization. The detector for this bug pattern looks for such errors by analyzing accesses to fields to determine which accesses are made while the object's lock is held.

Fields which are sometimes accessed with the lock held and sometimes without are candidate instances of this bug pattern. We use heuristics to reduce the number of false positives:

- Public fields are ignored

- Volatile fields are ignored

- Fields that are never read without a lock are ignored

- Access in methods unlikely to be reached when the object is accessible to multiple threads (e.g., constructors) are ignored

- Accesses in nonpublic methods called only from contexts where the lock is held are treated as locked

We count the number of reads and writes of a field performed with and without a lock. We report that a field is inconsistently synchronized if there are some unsynchronized accesses to a field and

$$2(\text{RU} + 2\text{WU}) \leq (\text{RL} + 2\text{WL}) \text{ and } \text{RU} + \text{WU} \leq \text{RL} + \text{WL}$$

where RU and WU are unlocked reads and writes, and RL and WL are locked reads and writes. This formula captures both the idea that most of the accesses should be performed while a lock is held, and that looking at whether a lock is held when a write is performed is often a better indication that the synchronization guarding the field was intentional rather than incidental.

The detector for inconsistent synchronization is one of the more complicated that we have implemented. It uses dataflow analysis to determine where locks are held and to determine which objects are locked, since the analysis relies on being able to determine when a lock is held on the reference through which a field is accessed. We perform redundant load elimination to determine when a value loaded from a field is likely to be the same value as that of an earlier load of the same field; without this analysis, the detector would treat all of the references to x in the following code as different objects:

```
// x is a field
synchronized (x) {
  ... = x.f;
  x.g = ...
}
```

The detector uses a whole program analysis, since fields may be accessed from classes other than the class in which they are defined.

The inconsistent synchronization detector is interesting because inferring whether or not a class is intended to be thread-safe in the absence of specifications is difficult. The detector relies on the fact that synchronizing on the `this` reference is a common idiom in Java, and uses heuristics to make an educated guess about whether synchronization is intended or incidental.

An example of inconsistent synchronization is seen in Figure 4. This method is in the `Vector` class of the GNU Classpath library, version 0.08. The `Vector` class is specified as being thread safe. Most of the accesses to the class's fields are protected by synchronizing on the object's `this` reference. In the code shown, the `elementCount` field is accessed

```
// GNU classpath 0.08,
// java.util,
// Vector.java, line 354

public int lastIndexOf(Object elem) {
  return lastIndexOf(elem, elementCount - 1);
}
```

Figure 4: Example of inconsistent synchronization.

without synchronization. The bug is that because of the lack of synchronization, the element count may not be accurate when it is passed to the `lastIndexOf(Object, int)` method (which is synchronized). This could result in an `ArrayIndexOutOfBoundsException`.

In the development of the FindBugs tool, we have run it regularly on several applications and libraries, tracking new releases of those packages as they are made available. We have noticed that it is fairly common for programmers not to understand the synchronization requirements of classes when they are performing maintenance (to fix bugs or add new features), and we have seen a number of synchronization bugs introduced this way. Detectors such as the inconsistent synchronization detector can serve as a useful safeguard against the introduction of bugs during code evolution.

4.8 Static Field Modifiable By Untrusted Code (MS)

This problem describes situations where untrusted code is allowed to modify static fields, thereby modifying the behavior of the library for all users. There are several possible ways this mutation is allowed:

- A static non-final field has public or protected access.

- A static final field has public or protected access and references a mutable structure such as an array or `Hashtable`.

- A method returns a reference to a static mutable structure such as an array or `Hashtable`.

For example, Sun's implementation of `java.awt.Cursor` has a protected static non-final field `predefined` that caches references to 14 predefined cursor types. Untrusted code can extend `Cursor` and freely modify the contents of this array or even make it point to a different array of cursors. This would allow, for example, any applet to change the cursors in a way that would affect all applets displayed by the browser, possibly confusing or misleading the user. Another example is the `javax.sql.rowset.spi.SyncProvider` class in Sun's implementation of the core J2SE libraries, which contains 11 public static nonfinal int fields with names that are all uppercase (which is usually used to denote a static final field).

4.9 Null Pointer Dereference (NP), Redundant Comparison to Null (RCN)

Calling a method or accessing an instance field through a null reference results in a `NullPointerException` at runtime. This detector looks for instructions where a null value might be dereferenced.

Our implementation of the detector for this pattern uses a straightforward dataflow analysis. The analysis is strictly

```
// Eclipse 3.0,
// org.eclipse.jdt.internal.ui.compare,
// JavaStructureDiffViewer.java, line 131

Control c= getControl();
if (c == null && c.isDisposed())
    return;
```

Figure 5: Example of null pointer dereference.

```
// Sun JDK 1.5 build 59,
// java.awt, MenuBar.java, line 168

if (m.parent != this) {
    add(m);
}
helpMenu = m;
if (m != null) {
    ...
```

Figure 6: A redundant null comparison.

intraprocedural; it does not try to determine whether parameters to a method could be null, or whether return values of called methods could be null. The detector takes `if` comparisons into account to make the analysis more precise. For example, in the code

```
if (foo == null) {
    ...
}
```

the detector knows that `foo` is null inside the body of the `if` statement.

Two types of warnings are produced. Null pointer dereferences that would be guaranteed to occur given full statement coverage of the method are assigned high priority. Those guaranteed to occur given full branch coverage of the method are assigned medium priority[2].

An example of a null pointer dereference found by the detector is shown in Figure 5. We were surprised to find that a bug this obvious could find its way into a mature, well-tested application. However, this was only one of a significant number of similar bugs we found in Eclipse.

In addition to finding possible null pointer dereferences, this detector also identifies reference comparisons in which the outcome is fixed because either both compared values are null, or one value is null and the other non-null. Although this will not directly result in misbehavior at runtime, it very often indicates confusion on the part of the programmer, and may indicate another error indirectly. (This phenomenon is described in greater detail in [46].) Figure 6 shows some code from the `java.awt.MenuBar` class from Sun JDK 1.5 build 59. Here the code manifests the implicit belief that m is not null, because the `parent` field is accessed, and later that it might be null because it is explicitly checked. Because the beliefs contradict, one must be incorrect. In this paper, we only report RCN warnings in which a reference is dereferenced before is it checked for null.

4.10 Non-Short-Circuit Boolean Operator (NS)

Like the equivalent operators in C and C++, the Java `&&` and `||` operators have short-circuit evaluation. For this

---

[2]Because full branch coverage is often infeasible, some of the medium priority warnings do not indicate real bugs.

```
// Eclipse 3.0,
// org.eclipse.ui.internal.cheatsheets.views,
// CheatSheetPage.java, line 83
if(cheatSheet != null &
   cheatSheet.getTitle() != null)
    return cheatSheet.getTitle();
```

Figure 7: A non-short-circuit boolean operator bug.

reason, they are often used to test a reference value against null, and call a method if the reference is found not to be null.

Surprisingly, the non-short-circuiting `&` and `|` operators are also defined for boolean values. Programmers may unintentionally use one of these operators where they intended to use a short-circuiting boolean operator. Because both boolean expressions are evaluated unconditionally, a null pointer exception may result. An example is shown in Figure 7; this example would also be caught by the NP pattern, but other cases might result in problems not caught by our existing bug detectors, such as an out of bounds array reference.

4.11 Open Stream (OS)

When a program opens an input or output stream, it is good practice to ensure that the stream is closed when it becomes unreachable. Although finalizers ensure that Java I/O streams are automatically closed when they are garbage collected, there is no guarantee that this will happen in a timely manner. There are two reasons why streams should be closed as early as possible. First, operating system file descriptors are a limited resource, and running out of them may cause the program to misbehave. Another reason is that if a buffered output stream is not closed, the data stored in the stream's buffer may never be written to the file (because Java finalizers are not guaranteed to be run when the program exits).

The Open Stream detector looks for input and output stream objects which are created (opened) in a method and are not closed on all paths out of the method. The implementation uses dataflow analysis to determine all of the instructions reached by the definitions (creation points) of streams created within methods, and to track the state (nonexistent, created, open, closed) of those streams. If a stream in the open state reaches the exit block of the control flow graph for a method, we emit a warning.

To reduce false positives, we ignore certain kinds of streams. Streams which escape (are passed to a method or assigned to a field) are ignored. Streams passed into a method or loaded from a field are ignored, since we assume that other code is responsible for closing those streams. Streams that are known not to correspond to any real file resource, such as byte array streams, are ignored. Finally, any stream transitively constructed from an ignored stream is ignored (since it is common in Java to "wrap" one stream object with another).

An example of a bug found by this detector is shown in Figure 8. The `FileReader` object is never closed by the method.

4.12 Read Return Should Be Checked (RR)

The `java.io.InputStream` class has two `read()` methods which read multiple bytes into a buffer. Because the number

```
// DrJava stable-20040326
// edu.rice.cs.drjava.ui
// JavadocFrame.java, line 103

private static File _parsePackagesFile(
  File packages, File destDir) {
  try {
    FileReader fr =
      new FileReader(packages);
    BufferedReader br =
      new BufferedReader(fr);
    ...
    // fr/br are never closed
```

Figure 8: Example of open stream.

```
// GNU Classpath 0.08
// java.util
// SimpleTimeZone.java, line 798

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

Figure 9: An example of read return ignored.

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/stale elements in 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 the availability of data in the input stream) is seen, we inhibit the emission of any warnings for the next 70 instructions[3]. 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 9. 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

---

[3]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.

object will not be deserialized correctly, and the stream will be out of sync (preventing other objects from being deserialized).

4.13 Return Value Should Be Checked (RV)

The standard Java libraries have a number of immutable classes. 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.

4.14 Non-serializable Serializable class (SE)

This pattern looks for classes that implement the `Serializable` interface but which cannot be serialized. We check for two reasons why a class might not be serializable:

- It contains a non-transient instance field of a type that does not implement `Serializable`, or

- The superclass of the class is not serializable and doesn't have an accessible no-argument constructor.

We perform two additional refinements on this. If a class does not implement `Serializable` directly (but inherits it from a superclass or an interface) and doesn't declare a `serialVersionUID` field, the priority is reduced (to a level below that reported in this paper). We also lower the priority (below that reported in this paper) if a non-serializable instance field has an abstract or interface type (since all concrete instances the fields references may be serializable).

4.15 Uninitialized Read In Constructor (UR)

When a new object is constructed, each field is set to the default value for its type. In general, it is not useful to read a field of an object before a value is written to it. Therefore, we check object constructors to determine whether any field is read before it is written. Often, a field read before it is initialized results from the programmer confusing the field with a similarly-named parameter.

An example of a bug found by this detector is shown in Figure 10. In this case, the `monitorName` field is read an passed to another method before it has been initialized to any value.

```
// JBoss 4.0.0RC1
// org.jboss.monitor,
// SnapshotRecordingMonitor.java,
// line 44

public SnapshotRecordingMonitor()
{
  log = Logger.getLogger(monitorName);
  history = new ArrayList(100);
}
```

Figure 10: Example of uninitialized read in constructor.

```
// JBoss 4.0.0RC1
// org.jboss.deployment.scanner
// AbstractDeploymentScanner.java, line 185

// If we are not enabled, then wait
if (!enabled) {
  try {
    synchronized (lock) {
      lock.wait();
...
```

Figure 11: An example of an unconditional wait.

4.16   Unconditional Wait (UW)

Coordinating threads using `wait()` and `notify()` is a frequent source of errors in multithreaded programs. This pattern looks for code where a monitor wait is performed unconditionally upon entry to a synchronized block. Typically, this indicates that the condition associated with the wait was checked without a lock held, which means that a notification performed by another thread could be missed.

The detector for this bug pattern uses a linear scan over the analyzed method's bytecode. It looks for calls to `wait()` which are preceded immediately by a `monitorenter` instruction and are not the target of any branch instruction.

An example of a bug found by this detector is shown in Figure 11. The `enabled` field may be modified by multiple threads. Because the check of `enabled` is not protected by synchronization, there is no guarantee that `enabled` is still true when the call to `wait()` is made. This kind of code can introduce bugs that are very hard to reproduce because they are timing-dependent.

4.17   Wait Not In Loop (Wa)

The most robust way to implement a condition wait is to use a loop that repeatedly checks the condition within a synchronized block, calling `wait()` when the condition is not true. Programmers sometimes believe that the loop is not necessary: for example, there might only be a single condition associated with the monitor, so any notification would have to correspond to the condition becoming true. However, even if the monitor is associated with a single condition, there is a window between the time that the waiting thread is woken and when it reacquires the lock, during which another thread could cause the condition to become false again. The specification of `wait()` also allows spurious wakeups.

For these reasons, instances of code which do not call `wait()` in a loop are likely to be errors. While it is possible to correctly use `wait()` without a loop, such uses are

rare and worth examining, particularly in code written by developers without substantial training and experience writing multithreaded code.

The implementation of the detector for this bug pattern uses a linear scan over the bytecode of analyzed methods. It records the first occurrence of a call to `wait()`, and the first occurrence of a branch target (which is presumed to be a loop head instruction). If a call to `wait()` precedes the earliest branch target, then the detector emits a warning.

5.   EVALUATION

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 high and medium priority warnings produced by FindBugs version 0.8.4[4] 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

- Eclipse, version 3.0

- DrJava, version stable-20040326

- JBoss, version 4.0.0RC1

- jEdit, version 4.2pre15

---

[4]We used the `-workHard` FindBugs command line option, which enables analysis to eliminate some infeasible exception paths in analyzed methods.

code	classpath-0.08 warnings	serious	mostly harmless	false pos	rt.jar 1.5.0 build 59 warnings	serious	mostly harmless	false pos
DC	1	100%	0%	0%	88	100%	0%	0%
EC	0	—	—	—	8	100%	0%	0%
IS2	46	58%	23%	17%	116	44%	47%	7%
NP	7	85%	0%	14%	37	100%	0%	0%
NS	0	—	—	—	12	25%	66%	8%
OS	6	50%	0%	50%	13	15%	0%	84%
RCN	5	80%	0%	20%	35	57%	0%	42%
RR	9	100%	0%	0%	12	91%	0%	8%
RV	5	100%	0%	0%	7	71%	0%	28%
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%

code	eclipse-3.0 warnings	serious	mostly harmless	false pos	drjava-stable-20040326 warnings	serious	mostly harmless	false pos
DC	88	100%	0%	0%	0	—	—	—
EC	19	57%	0%	42%	0	—	—	—
IS2	63	61%	22%	15%	2	0%	0%	100%
NP	70	78%	7%	14%	0	—	—	—
NS	14	78%	21%	0%	0	—	—	—
OS	26	46%	0%	53%	4	100%	0%	0%
RCN	69	40%	11%	47%	0	—	—	—
RR	39	38%	0%	61%	0	—	—	—
RV	8	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%

code	jboss-4.0.0RC1 warnings	serious	mostly harmless	false pos	jedit-4.2pre15 warnings	serious	mostly harmless	false pos
DC	3	100%	0%	0%	0	—	—	—
EC	4	100%	0%	0%	0	—	—	—
IS2	34	23%	32%	44%	3	33%	33%	33%
NP	24	83%	4%	12%	1	0%	0%	100%
NS	0	—	—	—	1	0%	0%	100%
OS	10	60%	0%	40%	4	75%	0%	25%
RCN	14	42%	0%	57%	4	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%

Figure 12: Evaluation of false positive rates for selected bug pattern detectors. (Figure 2 lists the bug pattern codes used in the left-hand column.)

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

Figure 13: Bug counts for selected other detectors.

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

Figure 14: Application sizes and total number of warnings generated by FindBugs and PMD.

GNU Classpath [27] is an open source implementation of the core Java runtime libraries. rt.jar is Sun's implementation of the APIs for J2SE [31]. Eclipse [18] and DrJava [16] are popular open source Java integrated development environments. JBoss [33] is a popular Java application server. jEdit [34] 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 took no more than 65 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 45 minutes to analyze. The maximum amount of memory required to perform the analyses was approximately 500 MB. We have not attempted to tune FindBugs for performance or memory consumption.

5.1 Empirical Evaluation

Figure 12 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 instance 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 develop-

ment 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 ever find every bug. From our evaluation of Find-Bugs on real applications, we conclude that simple static tools like bug pattern detectors find an important class of bugs that would otherwise go undetected.

5.2 Other Detectors

In Figure 13 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.

5.3 Comparison with PMD

In Figure 14, we list the total number of thousands of lines of source code for each benchmark application[5], the total number of high and medium priority warnings generated by FindBugs version 0.8.4, and the number of warnings generated by PMD version 1.9 [38][6]. In general, FindBugs produces a much lower number of warnings than 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 in 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.

6. ANECDOTAL EXPERIENCE

In this section, we report on some of our experience in using the FindBugs tool in practice. In addition, we report

---

[5]Note that the figure for rt.jar is low because not all of its source code is available in the standard public distribution.
[6]For PMD, we used the suggested rulesets: basic, unused-code, imports, and favorites.

some information about the experience of two outside groups which have adopted our tool. These reports are not rigorous, but simply anecdotal reports of our experience.

6.1 Why Bugs Occur

We have been actively working on FindBugs for a year and a half, and have looked at a huge number of bugs. Based on this experience, we can offer the following observations on why bugs occur. These are by no means exhaustive, but may provide some food for thought.

Everyone makes dumb mistakes. This is perhaps the most fundamental theme of our work so far. Detectors for the most blatant and dumb mistakes imaginable routinely turn up real bugs in real software. One way to explain this phenomenon is that a huge number of bugs are just one step removed from a syntax error. For example, many of the null pointer bugs we've seen fall into this category: the programmer intended to use the `&&` operator, but mistakenly used the `||` operator. We even found a bug in code written by Joshua Bloch, author of *Effective Java* [8]: the bug, a class with an `equals()` method but no `hashCode()` method, was a violation of one of the proscriptions laid out in his book. The lesson here is that even the best programmers are not perfect 100% of the time. Bug checkers take static checking further than the compiler, and are able to catch errors that the compiler ignores.

Java offers many opportunities for latent bugs. The hashcode/equals and covariant equals bug patterns are examples of *latent* bugs: they don't affect the correctness of a program until a particular kind of runtime behavior occurs. For example, a class with a covariant equals method will work correctly until someone puts it into a map or set. Then, rather than failing in an obvious manner (such as throwing an exception), the program will quietly perform the wrong computation. Similarly, there are a number of patterns and requirements (such as those for a serializable class) that are not checked by the compiler but simply result in runtime errors when violated. Bug checkers help increase the visibility of some of these latent errors.

Programming with threads is harder than people think. We did a study of concurrency errors in Java programs [30], and found that misuse of concurrency (such as deliberate use of data races to communicate between threads) is widespread. The problem here is that programmers are not as scared of using threads as they should be. Java tends to hide many of the potential negative consequences of concurrency glitches. For example, a data race cannot cause a program to violate type safety, or corrupt memory. However, due to the aggressive reordering of memory accesses by modern processors and JVMs, programming with data races is a very bad idea. Concurrency bugs are especially problematic because they can be extremely difficult to reproduce. Bug checkers can help prevent concurrency errors before they make it into deployed code.

6.2 Our Experience

Our null pointer exception detector was partially inspired by a bug in Eclipse 2.1.0 that was fixed in 2.1.1 (bug number 35769). Our detector accurately identifies this error as well as a number of similar errors in the current releases of Eclipse.

Early, non-public builds of Sun's JDK 1.4.2 included a substantial rewrite of the `StringBuffer` class. Our tool found a serious data race in the `append(boolean)` method, and our feedback to Sun led them to fix this error in build 12.

In build 32 of Sun's JDK 1.5.0, a potential deadlock was introduced into the `ORBImpl` class, which is part of Sun's CORBA implementation. In this case, the `get_next_response()` method calls `wait()` while holding a lock that is also needed by the `notifyORB()` method, which is responsible for notifying the waiting thread. We discovered the bug using a detector which looks for calls to `wait()` made with more than one lock held. Even though it is possible to write correct code that calls `wait()` while holding multiple locks, this idiom is error prone and warrants careful inspection when used. When we reported the issue to Sun, it was confirmed to be a bug and a fix was made in build 44.

We applied our tool to the Java implementation of the International Children's Digital Library [17], and found a number of serious synchronization and thread problems. In talking to the developers, some of the errors we found closely paralleled a bug in an earlier version of the software that they had spent substantial time trying to diagnose and fix in the previous months. When we applied our tool to an earlier version of their software, we immediately found the error they had spent substantial time on.

We have used our tool in a senior level undergraduate course that covers many advanced Java topics, including threads and distributed programming. In recent semesters, we have encouraged students to use FindBugs as part of the development process for their projects. We have also applied the tool to project submissions from previous semesters. Our tool found many errors, including problems that one would never expect to find in production code, such as

- Executing the `run()` method of a thread object rather than starting the thread, and

- Performing `wait()` or `notify()` on an object while not holding a lock on that object.

We did not expect that the detectors we wrote particularly for undergraduate course work would find many problems in production code. However, we did find a case in Eclipse where a thread object was created but never run.

One of the nice features of the Java programming language is that it is impossible to "forget" to release a lock. However, JCP JSR-166 introduces a new set of more powerful concurrency abstractions to the Java programming language. These new concurrency mechanisms include locks for which unbalanced acquire and release calls are allowed. We worked with the JSR-166 expert group to develop a detector that generate a warning if an lock acquisition was not obviously matched with a release on all program paths (including paths arising from exceptions). When run on the code developed as part of JSR-166, the detector generated only 9 warnings on 78 uses of acquire that corresponded to places where non-standard locking was being performed. We didn't find any errors, but the care that went into the crafting of that library made it unlikely that there would be any to find.

6.3 User Experience

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

6.3.1 A Geographic Information System Application

One of the users is 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.

6.3.2 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 15 shows a graph of the number of warnings produced by FindBugs, CheckStyle [10], and PMD [38] 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 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.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have argued that a significant class of easily detectable bugs exists in production software. We have also outlined an approach, based on bug patterns, for turning specific bugs into detectors that can find similar bugs. Because we were able to identify a large number of bugs in real applications and libraries, we believe that wider adoption of static bug-finding tools could significantly improve software quality.

We believe that better integrating bug-finding tools into the development process is an important direction for future research. Some of the key challenges are:

1. Raising the awareness of developers of the usefulness of bug-finding tools

2. Incorporating bug-findings tools more seamlessly into the development process: for example, by providing them as part of Integrated Development Environments

3. Making it easier for developers to define their own (application-specific) bug patterns

4. Better ranking and prioritization of generated warnings

5. Identification and suppression of false warnings

6. Reducing the cost of bug-finding analysis, through incremental analysis techniques, and background or distributed processing

Ultimately, we envision application of bug-finding tools as a ubiquitous part of the software development process, much like unit testing.

8. RELATED WORK

Automatic techniques to find bugs in software have been extensively studied in previous research. One of the best known early tools for finding bugs in software is Lint [35], which used heuristics to find a variety of common errors in C programs. Many of the checks performed by Lint, such as use of uninitialized variables, are now integrated into compilers. More modern Lint-like bug checkers include LCLint [23, 22] and Jlint [3]. The FindBugs tool inherits much of its philosophy from the tools in the Lint family.

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. Although PREfix is unsound, it produces a relatively low percentage of false warnings.

Dawson Engler and his students and collaborators have done very successful work in developing effective analysis techniques to find bugs, including meta-level techniques for constructing bug detectors. The MC system [28] uses a
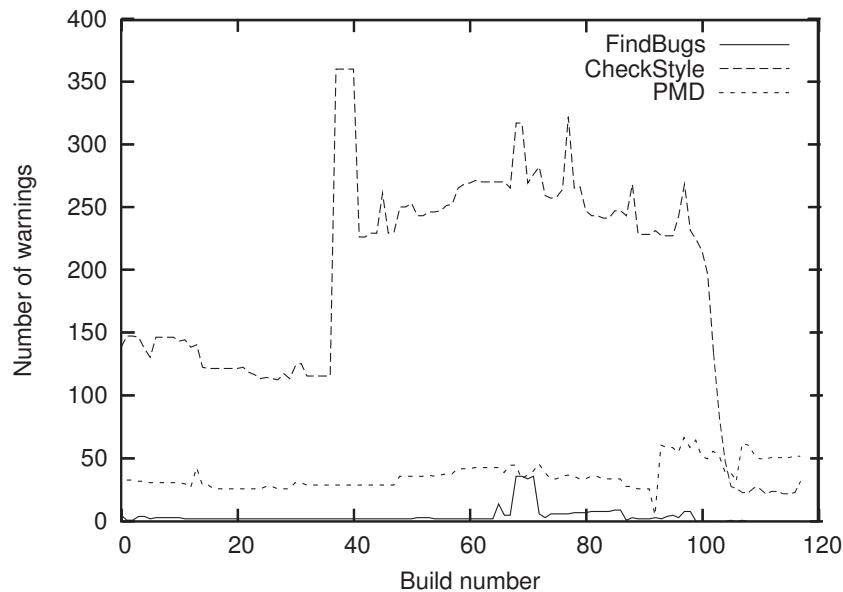
Figure 15: Number of warnings produced by FindBugs, CheckStyle, and PMD on a large financial application.

custom language, called Metal, to construct state-machine based detectors to find bugs in C programs. Metal uses syntactic patterns to model the effects of program statements on the state of "interesting" values on paths through a program. For example, pointer values can be checked to detect statements that dereference a value which could be null. Although the analysis performed by MC is unsound, it is very effective, finding hundreds of bugs in widely used open source and commercial software. MC has been used mainly to find bugs in operating system kernels [20, 12, 4], although the general techniques it employs are applicable to any system. Recent work by Back and Engler [5] has extended the analysis techniques used in MC to Java programs.

Checkers for bugs in multithreaded programs is an important subcategory of bug-finding tools. Warlock [44] examines C programs for accesses to variables not protected by a consistent set of mutual exclusion locks. RacerX [19] uses many of the same ideas as the MC system to find potential race conditions and deadlocks in C programs. It employs several novel heuristics for reducing false warnings.

An important challenge in effective use of unsound analysis techniques is mitigating the effect of false warnings. In [36], Kremenek and Engler propose a statistical technique for ranking warnings produced by unsound static analysis tools. Their work is based on the hypothesis that analysis decisions which result in relatively few program locations flagged as errors are more likely to be correct than those which mark many program locations as errors. By using their ranking technique, the authors were able to usefully apply error checkers with high false warning rates, by ranking the warnings likely to be genuine above those likely to be false.

In contrast to unsound analyses, sound analyses are capable of proving the absence of certain kinds of bugs. Although

sound analyses may produce false warnings, programs for which no warnings are produced are guaranteed not to contain any instances of the kind of bug checked by the analysis. The CQual system [25] uses type qualifiers to associate correctness properties with program types. The authors have used CQual to find format string vulnerabilities [43] and locking bugs [26]. SLAM [6] verifies that C programs are free of violations of temporal correctness properties. It abstracts programs into *boolean programs*, the reachable states of which are explored. If error states are not reachable from the boolean program, then the original program is also free of violations. Reaching an error state may indicate an infeasible path; when this happens, the boolean program is refined an re-analyzed. The authors used SLAM both to validate and find bugs in Windows device drivers. The ESP system [15] also analyzes C programs to verify temporal correctness properties. It uses a novel analysis technique to avoid losing information at control joins, while avoiding (in most cases) tracking an exponential number of program states.

Several general static analysis techniques allow specification of code patterns which may be used to find bugs in programs. ASTLOG [13] is a language for specifying patterns to match the abstract syntax trees of C and C++ programs. 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. A paper by Liu et. al. [37] describes an efficient technique for finding occurrences of regular expression patterns in arbitrary graphs, matching query parameters with concrete graph features. Many program properties, including occurrences of bug patterns, can be concisely stated using this technique.

Several dynamic analyses to find bugs have been proposed in recent years. Eraser [41] dynamically computes the set

of locks held during accesses to shared data. Accesses to the same location with inconsistent lock sets are potential bugs. In [11], Choi et. al. build upon the basic Eraser lock set technique by proposing novel techniques to improve the efficiency of dynamic race detection. Static analysis is used to avoid instrumentation of accesses that cannot be involved in races, and several instrumentation and runtime optimizations are used to reduce the cost of instrumentation without sacrificing accuracy. Hangal and Lam [29] use instrumentation to detect violations of likely runtime program invariants, and were able to find the root causes of several difficult bugs in large Java programs. Dynamic analysis to detect likely program invariants was pioneered by Ernst et. al. in [21].

One important issue in bug-finding research is the need to evaluate the effectiveness of tools in finding bugs. This problem is difficult because the population of "real" bugs in a program is generally not known, especially for large systems. One approach is to simply compare the output of a number of different bug finding tools; the union of the genuine bugs found by all tools compared can be considered a lower bound on the number of real bugs in the program. Rutar et. al. [40] compare a number of tools for finding bugs in Java programs.

Acknowledgments

9. REFERENCES

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

[2] Apache Ant, http://ant.apache.org/, 2004.

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

[4] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy, Oakland, California*, May 2002.

[5] G. Back and D. Engler. MJ: A system for constructing bug-finding analyses for Java. http://www.stanford.edu/~gback/gback-icse2004.pdf, 2003.

[6] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, Jan. 2002.

[7] The Byte Code Engineering Library, http://jakarta.apache.org/bcel/, 2004.

[8] J. Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2002.

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

[10] CheckStyle, http://checkstyle.sourceforge.net, 2004.

[11] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, July 2002.

[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[13] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *USENIX Conference on Domain Specific Languages*, pages 229–241, Santa Barbara, 1997.

[14] M. C. Daconta, E. Monk, J. P. Keller, and K. Bohnenberger. *Java Pitfalls*. John Wiley & Sons, Inc., 2000.

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

[16] DrJava, http://www.drjava.org/, 2004.

[17] A. Druin, B. Bederson, A. Weeks, A. Farber, J. Grosjean, M. Guha, J. Hourcade, J. Lee, S. Liao, K. Reuter, A. Rose, Y. Takayama, L., and L. Zhang. The international children's digital library: Description and analysis of first use. Technical Report HCIL-2003-02, Human-Computer Interaction Lab, Univ. of Maryland, January 2003.

[18] Eclipse, http://www.eclipse.org/, 2004.

[19] D. Engler and K. 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.

[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*, Oct. 2000.

[21] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering*, 27(2):1–25, Feb. 2001.

[22] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, Pennsylvania, May 1996.

[23] D. Evans, J. Guttag, J. Horning, and Y. M. 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.

[24] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002.

[25] J. S. Foster, M. Fähndrich, and A. 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.

[26] J. S. Foster, T. Terauchi, and A. 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.

[27] GNU Classpath, http://www.gnu.org/software/classpath/, 2004.

[28] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 69–82, Berlin, Germany, June 2002.

[29] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.

[30] D. Hovemeyer and W. Pugh. Finding concurrency bugs in Java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, Newfoundland, Canada, July 2004.

[31] Java(tm) 2 Platform, Standard Edition, http://java.sun.com/j2se/, 2004.

[32] Collected java practices. http://www.javapractices.com.

[33] JBoss, http://www.jboss.org/, 2004.

[34] jEdit, http://www.jedit.org/, 2004.

[35] S. Johnson. Lint, a C program checker. In *UNIX Programmer's Supplementary Documents Volume 1 (PS1)*, April 1986.

[36] T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA*, pages 295–315, June 2003.

[37] Y. A. Liu, T. Rothamel, F. Yu, S. Stoller, and N. Hu. Parametric regular path queries. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Washington, D.C., USA, June 2004.

[38] PMD, http://pmd.sourceforge.net, 2004.

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

[40] N. Rutar, C. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering*, Saint-Malo, France, November 2004.

[41] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[42] D. Schmidt and T. 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.

[43] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., Aug. 2001.

[44] N. Sterling. WARLOCK — a static data race analysis tool. In *Proceedings of the USENIX Annual Technical Conference*, pages 97–106, Winter 1993.

[45] B. Tate. *Bitter Java*. Manning Publications, 2002.

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