MJ - A System for Constructing Bug-Finding Analyses for Java

Godmar Back Dawson Engler
Stanford University
Computer Systems Laboratory
{gback|engler}@stanford.edu

Abstract

Many software defects result from the violation of programming rules: rules that describe how to use a programming language and its libraries and rules that describe the dos and don'ts within a given application, library or system. MJ *is a language and an engine that can succinctly express many of these rules for programs written in Java.* MJ *programs are checkers that are compiled into compiler extensions. A static analysis engine applies the extensions to user code and flags rule violations. We have implemented and tested several extensions in* MJ *for both general and application-specific rules. Our checkers have found dozens of bugs in some widely-deployed and mature software systems.*

1. Introduction

Software has too many bugs. Recently, there has been a significant amount of work devising automatic tools to detect such bugs. These range from annotation-based approaches [9, 11, 17], to stand-alone tools tailored to checking specific error types [4, 19], to tools that allow users to extend them to check new properties [2, 5, 7, 8].

Annotation-based systems have the strength of providing a form of checkable documentation, but tend to require a heavy investment by the user before yielding good results. In contrast, tool-based approaches tend to have a much lower incremental cost per-checked line of code, and hence higher bug counts (often thousands for large systems). Extensible tools have the further advantage that they can be easily tailored to do unusual analyses (such as using statistical analysis to infer which properties to check) or to directly target identified expensive bugs to ensure they will not be reintroduced.

The bulk of this prior work has focused on unsafe languages (i.e., C and C++). Although some bug-finding tools for Java have been developed, most are either annotation-based and require a large amount of labor, or only check a fixed set of properties (such as race detection), or are limited to enforcing stylistic rules (e.g., that switch statements have at most five case arms).

This paper describes MJ, an extensible bug-finding system for checking Java. It provides users with a language and runtime system that allows them to write extensions that are dynamically linked into a sophisticated compiler framework and automatically applied to bytecode. Many extensions are less than a hundred lines of code, are easily applied to new systems without alterations, and have found dozens of bugs in widely-used systems.

MJ is based on the metacompilation approach [8], which we originally developed in the context of C. This paper extends this approach to Java, demonstrating that it works well in the context of a type-safe language. The cornerstone of the approach is that many abstract rules have a straightforward mapping to source code. Thus, given knowledge of a rule, a compiler extension can automatically find violations of it. For example, the rule "calls to `v.lock()` must be matched with a call to `v.unlock()`" can be checked by following every path after a `v.lock()` call making sure it hits an `v.unlock()` call. In general, checking rules has two parts: (1) mapping the actions that are relevant to a rule to source code and (2) expressing the rule constraints on these actions. MJ supports the first by providing a rich, high-level pattern language that allows users to easily match source constructs. It supports checking in two ways. First, it provides a simple state machine-based, dataflow-oriented language which allows users to concisely express constraints. While the language provides a framework, it does not limit the analyses users can implement—users can interweave arbitrary code to extend and customize it. Second, MJ plugs these checks into a compiler framework that takes care of both low-level details and provides sophisticated analyses. For example, this framework automatically reconstructs variables from bytecode-level register assignments, uses reaching definitions and other dataflow analyses to track these variables, and performs flow-sensitive analysis.

The main contributions of this paper are:

1. A system and language that make it easy to write a broad set of checkers in a unified framework.
2. Showing that common sources of bugs can be expressed succinctly within this framework: most of our checkers are less than a hundred lines of code.
3. Showing that the approach is effective by using it to find dozens of bugs in more than six different systems.

The rest of this paper is structured as follows: Section 2 explains the syntax and semantics of MJ using three small examples. Section 3 describes the MJ language; Section 4 describes the system in detail. Section 5 reports on our experiences with uses of specific extensions. Section 6 discusses related work, and Section 7 concludes.

2. Overview

This section gives two motivating examples for MJ. In Java, objects of the built-in string class are immutable. A common source of bugs is to assume that string operations such as `replace()`, `concat()`, or `trim()` affect the string objects to which they are applied, when in fact a new string object is returned. As an example, consider the code in Figure 1, which MJ found in Oceanstore [16], a large software system for distributed global persistent data.

In this code snippet, the programmer tries to create a unique message id by combining (concatenating) different elements of a message and computing a SHA-1 hash on the result. In reality, only the message `src` element is used to compute the SHA-1 hash, making the message id not nearly as unique (and secure) as the programmer intended. This example demonstrates both that potentially serious logical bugs can result from obvious violations of simple API rules and that they can be readily detected automatically.

Figure 2 shows a simple MJ program that can identify this and similar bugs. It uses the pattern `java.lang.String.anymethod(...)` to match code locations where any method of the String class is invoked with any number of parameters. Upon finding a match, the analysis engine will bind the returned value to a state variable `str`, which will be tracked to all code locations that

OceanStore ostore/tapestry/channel/ChannelRouteMsg.java:

```
public SecureHash generate_msg_id() {
    String  id = src.toString();
    id.concat( dest.toString() );
    id.concat( Integer.toString(channel_id) );
    id.concat( Integer.toString(sequence) );
    id.concat( Integer.toString(frg_index) );
    msg_id = new SHA1Hash(id);
    return msg_id;
}
```

Figure 1. Calls to `id.concat` do not affect `id`.

```
sm stringchecker {
    state decl { java.lang.String } str;
    { public HashMap tracking = new HashMap(); }
    init { tracking = new HashMap(); }

start:
    { str = java.lang.String.anymethod(...) }
    ==> str.tracked, {
        tracking.put(str.getDefinition(), ...);
    };

str.tracked:
    { str } ==> {     // matches any use
        tracking.remove(str.getDefinition());
    };
    final { bugs.addAll(tracking.values()); }
}
```

Figure 2. String checker: identifies locations where (immutable) strings are discarded

the so-bound definition reaches. In addition, it will record this match in a hashtable, which is accomplished by calling out to the user-specified Java code embedded in the { ... } clause.

Subsequently, the analysis engine will examine all locations reached by that definition to see whether they use the returned value. This check is described using the simple {`str`} pattern, which denotes any use other than simply assigning the returned value to one (or more) local variables. It is guarded by the state `str.tracked`, which binds `str` to only those values tracked from a previous call to a string method. If a use is found along any path, the definition is removed from the hashtable. After all paths have been explored, definitions that are left in the hashtable are flagged as potential errors.

A second example illustrates MJ's ability to perform simple flow-sensitive analyses. In Java, a common idiom is to check if an object reference is `null`, and then to derefer-

```
sm nullchecker {
  state decl anyobject o;

  start:
     { o != null } ==> true = o.isnotnull,
                       false = o.isnull ;
  o.isnull:
     { anyclass.anyfield = o } ==>
          { /* ignore this innocuous use */ }
      // any use
    | { o } ==> { err("use of null object"); } ;
  o.isnotnull: { o.size() } ==>
      { /* possible size() pattern */ }
    ;
}
```

Figure 3. Null checker: flags when variables are accessed that could be null

stringchecker checks that results of string manipulations aren't discarded, warns if non-interned strings are compared, warns if unneeded copies are made

droppedexceptions warns if exceptions aren't thrown

apples&oranges warns if Object.equals() is invoked on incompatible objects

nullchecker warns if a pointer known to be or possibly null is dereferenced.

lockunlock checks that calls to lock/unlock methods are paired on all paths.

signedbyte warns of potentially wrong sign-extension.

superfinalizer warns if finalize() method doesn't call super.finalize() on all paths

staticlock warns when a public static field is used as a lock.

neverused flags redundant operations

nullargs infers statistically if arguments must be checked for `null`

Table 1. Examples of the types of properties that can be checked by short MJ programs

ence the reference only on those paths where it is not `null`. Not following this idiom is often a bug that can crash the current thread or application. Figure 3 shows an MJ program that can detect such violations. This checker matches conditional branches that depend on the outcome of a comparison with `null`. Different variable states are propagated along the true (`isnull`) and false (`isnotnull`) edges in the control flow graph, carrying with them the flow-related information about the outcome of the test. If a subsequent use of this object reference is found along a path that is reachable from the `isnull` edge, the checker flags this as a possible bug. Figure 4 shows three typical bugs this checker found in mature and widely-used software systems.

MJ extensions are not verifiers and can miss errors. Conversely, they may report false positives. MJ language features can be used to identify and suppress certain false positives. For instance, we found that the nullchecker produces a number of false positives for Java collection types if the programmer treated a `null` reference similar to a collection with zero elements. In these cases, the null check was followed by a call to a method such as `size()` on the non-null path. The result of `size()` was then used to guard against the potentially dangerous access. Adding a MJ pattern {`o.size()`} guarded by the "o.isnotnull" state allows us to recognize this particular false positive easily.

JDK 1.4 javax/swing/LayoutComparator.java:

```
if (a == null) {
    // 'a' is not part of a Window hierarchy.
        Can't cope.
    throw new ClassCastException(a.toString());
}
```

Rhino 1.5 RC5 org/mozilla/javascript/IRFactory.java:

```
String s = id.getString();
if (s != null && s.equals("__proto__")
    || s.equals("__parent__")) { ... }
```

Jigsaw 2.2.2 org/w3c/jigsaw/filters/UseProxyFilter.java:

```
Reply r = request.makeReply(HTTP.USE_PROXY);
if (r != null) {
    /* ... */
    r.setStream(g);
}
r.setLocation(proxy_url);
return r;
```

Figure 4. Typical null checker bugs

Table 1 shows a list of checkers we implemented to demonstrate the flexibility and applicability of MJ. The checkers shown are generally small (the median number of lines is 68), the largest checker (neverused) is 391 lines, the majority of which is code that relates to statistical ranking and HTML presentation. Table 2 shows the codebases over which we ran our checkers. On a 1.13GHz PC with 2 GB of RAM using Sun's JDK 1.4 Hotspot VM, it took between a few seconds and 2 minutes to run these checkers, depending on the codebase.

OceanStore A distributed global persistent storage system, CVS Jan 2003, 166K

JDK 1.4 javax javax.* hierarchy in JDK 1.4.1, including the Swing GUI, 132K

Ptolemy A system for simulation, version II 2.0.1, 87K

Jigsaw W3C's webserver, version 2.2.2, 68K

OpenMap OpenMap 4.6beta Sep 2003 by BBN Technologies [3], 124K

Mozilla Rhino JavaScript engine, version 1.5 RC5, 31K

Netbeans A integrated IDE for Java, CVS May 2003, 217K

Table 2. Code bases to which we applied our checkers. The line numbers are the estimated effective number of lines as computed from the line number information in the bytecode

3. The MJ Language

The MJ system consists of a description language for analyses, a compiler that turns these analyses into compiler passes, and runtime support for processing and presenting analysis results. Its basic structure is shown in Figure 5. To analyze code using the MJ system, a user merely has to load its bytecode into the JOEQ environment, which requires the same actions needed to run it, such as placing jar files and setting up classpaths.

An MJ checker is compiled to Java code by the MJ compiler. The compiled Java code, along with the MJ runtime, is dynamically loaded by the JOEQ analysis framework. The MJ runtime provides support for accessing JOEQ internals. A library ranks and presents analysis results; its output can be either plain text for quick analyses, XML for processing in a bug database, or HTML. The produced HTML links to the original source code, marking the flagged lines in color for easy inspection.

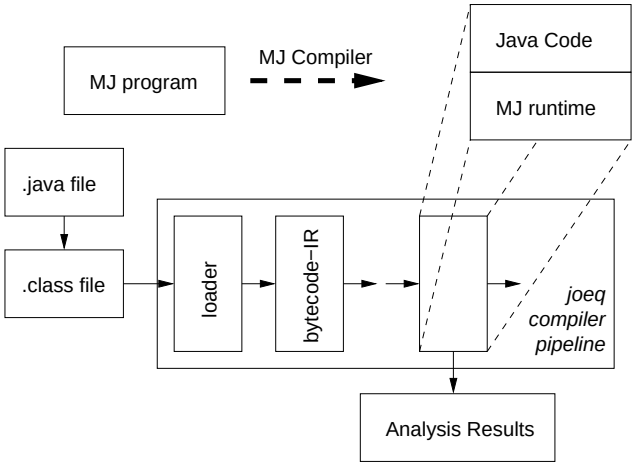

Figure 5. Overview of the MJ system

3.1. Pattern Matching

The MJ language is based on the *metal* checking language we developed for checking C code [13]. MJ programs use a state machine abstraction that is combined with traditional dataflow analysis [18]. A programmer can specify patterns that match Java source code constructs, such as method invocations, field accesses etc. Patterns are associated with actions, which are performed if a location matching the pattern is encountered in bytecode. An action may propagate a new global state to source locations that are reachable from the location matched.

Patterns may contain references to typed state variables declared in the MJ program. If a pattern containing such a variable is matched, the variable is bound to the operand in whose place it occurs. If the analysis associates a state with a variable, the resulting state variable is propagated to all locations that this operand reaches.

Patterns are guarded by states, which are either global states or those of state variables. For instance, in Figure 2, the pattern `{ str }` is guarded by state `str.tracked`. If a pattern is guarded by a state, a match is only possible at those locations that are reached by the guarding state. If a state guards multiple patterns, which must be separated by a "`|`" character, the analysis engine matches patterns in the order in which they are listed. A special `start` state is assumed to reach all locations. A special `stop` state may be used to stop the propagation of a state variable. For instance,

```
o.isnull:
  { static java.lang.System.exit(int) } ==> o.stop
```

prevents the propagation of variable state `o.isnull` when a call to `exit()` is seen, preventing the nullchecker from flagging a false positive if the programmer used an idiom such as "`if (o == null) System.exit(1);`" followed by an access to `o`.

Table 3 shows some examples of legal patterns. These include arithmetic operations, comparisons of scalars and references, array reads and writes, object allocations, type tests (instanceof), type casts, method invocations (including calls to initializers, static methods and the super.* calling convention), field accesses, and others. For field accesses and method invocations, a programmer may use regular expressions to specify the field and method name; for instance, `"^set.*"(...)` will match all `set` methods with any number of arguments. In addition, we support several kinds of wildcards: `anymethod`, `anyfield`, `anyclass`, `anyobject`, and `anyvalue`. Special patterns are used for entering and leaving synchronized blocks and for paths that leave a current method.

Pattern matching relies on compile-time type information. We follow the intuition that a programmer would have about the types of the objects involved. When matching a method invocation or field access, we match sites where the object's compile-time type is compatible with the type specified. For object allocations, we require that the type be the same. A special keyword `anynew` matches allocation sites that are merely compatible with the type given. A special modifier `exact` restricts call sites and field accesses to the exact compile-time type.

If an analysis cannot be expressed in the pattern and state model that MJ supports directly, then a programmer may use callouts written in Java. A special `${...}` pattern allows a user to specify arbitrary Java code that is invoked when the analysis engine needs to determine if an action should be triggered at a given location.

Examples of declarations

```
state decl anyobject o;
state decl anyvalue r;
state decl { int } v;
state decl { java.io.File } f;
```

Examples of patterns

`java.lang.String.anymethod(...)` matches all method calls on a `String` object with any number of arguments

`static java.lang.System.exit(int)` matches calls to `System.exit()` with one integer argument

`super.finalizer()` matches all calls to `super.finalizer()`

`f = anyclass.anymethod(...)` matches any method returning a `java.io.File` object

`r = anyobject.".*[Ff]lag.*"` matches all reads from fields that have `flag` or `Flag` in their field name

`v = byte[int]` matches reads from a byte array

`return v` matches return statements that return `v`

`v | int` matches any bitwise or of `v` with an integer operand

`new Type()` matches any allocation of `Type` objects

`(Type)o` matches any cast of `o` to `Type`

`o == null` matches any test of `o` against `null`

`lock o` matches the beginning of synchronized(o) block

Table 3. Examples of declarations and patterns MJ supports

4. MJ's Runtime Environment

MJ programs are translated into specialized dataflow problems. State propagation is modeled as a dataflow problem whose transfer function creates states for code locations as needed, and perform a combined reaching definitions analysis to track variable states to the locations they reach. In this section, we explain how our generated code implements these algorithms. We give some necessary background on the JOEQ compiler infrastructure first.

4.1. The JOEQ Infrastructure

JOEQ [20] is a component framework designed to facilitate research in virtual machine and compilation technologies. We use its frontend to load Java bytecode, which allows us to add our analyses as passes to its compiler. Because JOEQ was designed with extensibility in mind it makes extensive use of the visitor design pattern [12]. JOEQ is itself implemented in Java, so adding a pass is as simple as writing a specific visitor class for JOEQ's intermediate representation (IR).

A MJ program is compiled into a set of Java classes that implement a specialized intraprocedural forward dataflow analysis. A static skeleton class provides the frame for the analysis, the generated program subclasses from this skeleton. The skeleton provides the basic framework for traversing the control flow graph and computing dataflow facts, compiler-generated subclass implements the specific transition function given by the MJ program. User-provided code is enclosed in methods of that subclass and invoked as needed. The translated MJ program uses the underlying JOEQ infrastructure.

Using JOEQ benefits us in several ways: first, it performs the mundane task of Java class loading, including the resolution of link-time references. More importantly, it allows us to make seamless use of other analysis components, such as class hierarchy analysis, that have been implemented within or for the JOEQ framework.

4.2. Dataflow Formulation

MJ's dataflow analysis computes dataflow facts for each quad instruction in JOEQ's IR. Quads are a form of four-address-code; operands are registers, constants, or types. Quad IR uses a similar instruction set as Java bytecode: for instance, a `getfield` (read from instance field) bytecode instruction is translated into a `GETFIELD` quad. The main difference between bytecode and quad IR is that the bytecode-to-quad translation replaces Java's stack-based execution model with a more convenient to use register-based model. Every local variable, incoming parameter, and stack location is assigned a register.

MJ patterns match individual quad instructions in JOEQ's IR. Because the Java source-to-bytecode translation preserves most of a program's semantics, we can easily identify most source patterns in the quad representation. We use the line number information and local variable tables (if available) to map quads and registers back to source code lines and local variables.

We implement the dataflow analysis using a traditional worklist algorithm. Each quad has an associated state cache, which includes global states and those bound to a state variable. Global states are represented as integers. A variable state is a tuple *(v.i, def, set)* where v represents the state variable in the MJ program, $v.i$ the integer representing the state v is in, and *def* represents the source lo-

cation where the variable was bound. A definition is a pair $(quad, reg)$, meaning that this variable was bound to register reg at quad $quad$. $regset$ represents the set of registers that the so-defined register can reach.

The set of states held in a quad's state cache represents the *in* set for that quad in the dataflow analysis. Our generated code computes the transfer function for a quad as follows. For each global state in its *in* set, we examine if any of the patterns guarded by that state match this quad. If the quad does not match the pattern, the global state is propagated to all successor quads. If there is a match, the action associated with the match dictates what function is to be executed. If the action describes a new global state other than the `stop` state as its target, a new global state is created and propagated to all or some successors.

If the action targets a state variable, a new variable state is created that is bound to the registers that contained the operands that matched the position of the state variable in the MJ program. For instance, consider the `o != null` pattern shown in Figure 3 in conjunction with the following quad with quad id $\#2$. This quad is a conditional branch that jumps to label `BB7` if register `R2` is not equal to `null`:

```
#2   IFCMP_A R2 Object, AConst: null, NE, BB7
```

In the null checker, the transfer function for this quad will propagate variable state $(o.isnotnull, (\#2, R2), R2)$ to the quad at label `BB7`, and it will propagate variable state $(o.isnull, (\#2, R2), R2)$ to the fallthrough quad of the branch.

Variable states that are in the *in* set of a quad require additional handling: first, when deciding whether a quad matches the pattern guarded by such a state, we need to also check whether any of the registers in the variable state's register $regset$ may reach the register that is used in place of the bound variable v. Second, even if there is no match, we need to compute reaching definitions in parallel by adding copied registers to a state's $regset$ and by removing a quad's killset from the state's $regset$. If the $regset$ becomes empty (that is, if the definition to which the variable was bound is dead), the state is killed.

The confluence operator for our dataflow analysis is the union of the $regset$s computed individually for each *(v.i, def)* pair. In other words, we merge register sets at join points if they represent the same variable state and if they were bound at the same location in the quad code.

4.3. User-Defined States

For cases where the provided state model is too restrictive, we provide a way for users to extend and customize how MJ creates and propagates states. For instance, a user may wish to track additional, dependent definitions along with a state variable. By adding a `use` clause to a state variable declaration, the user tells the runtime system to use instances of a user-provided class to represent states instead of the default, system-provided implementation. For instance, `state decl { int } typeflag use DependentStateVariable;` tells the MJ runtime to instantiate objects of the user class `DependentStateVariable` when tracking instances of `typeflag`. The user-provided class is responsible for implementing the dataflow analysis methods that implement the transfer function across a quad and the confluence function that is used when merging successor states.

4.4. Limitations

Instead of basing our pattern matching on the quad IR, we could have modified a Java source frontend to match source code directly. Aside from the obvious advantage that working with compiled code enables analyses in the absence of source code, using quads also simplifies the matching of patterns that may occur in different variations. For instance, a pattern such as `a < b` will also match if the source code contained a negated pattern such as `b >= a`, because their representations at the IR level are identical.

A disadvantage of quad matching is that a few source constructs generate multiple quads. For instance, an object allocation using the `new` operator is translated into a `NEW` quad instruction, followed by an `INVOKESPECIAL` instruction to call the respective constructor. A MJ programmer may decide to match on either instruction, depending on the analysis.

A second example of more complex matching are `instanceof` type inclusion tests. These are not directly translated into a branch instruction: instead, the result of the test is stored in a register, which is subsequently tested in a branch. In this case, a state variable needs to keep track of two registers: the register that holds the reference to the object to which the type test was applied, and the register that holds the boolean result of the test. We use a user-defined state variable as described in Section 4.3 to implement this.

5. Applications

This section presents a suite of checkers written using MJ. These checkers give a feel for the variety of checks that can be expressed in the MJ framework.

5.1. Checking Language-specific Properties

MJ extensions do not have be involved to be useful. Some language-specific rules can be expressed very succinctly. For instance, in Java, while all objects can be compared using `Object.equals()`, typically the comparison make only sense if the compile-time type of the objects are

Bug caused by using `equals()` on incompatible types

```
boolean filterComment()
{
  User user = User.getUser();
  Document d = getContainingDocument();
  return !user.equals(d.getAuthor());
}
```

Short checker that found this bug:

```
sm applesandoranges {
  decl anyobject objl, objr;

start:
  { objl.equals(objr) } ==> {
    // warn if !objl.type.isCompatible(objr.type);
  }
}
```

Figure 6. Bug caused by using `equal()` on incompatible types, resulting in loss of confidentiality in a commercial system.

related; otherwise, the implementation of equals() may either crash or, worse, silently return `false`, leaving a possible error untrapped.

Figure 6 shows a security hole caused by violating this rule and a checker that found it. The shown `filterComment()` method is part of a code base of a leading enterprise software system. The method's purpose is to determine whether a user requesting a document is authorized to see comments that should be visible only to the author. In this case, the consequence of this bug was a breach of confidentiality, exposing these comments to all viewers.

5.2. Checking Interface Properties

Many APIs have the property that a call to `A` must be followed by a call to `B`. For instance, calls to `lock()` must be paired with calls to `unlock()`. Java provides the try/finally construct to reduce the likelihood that such properties are

Jigsaw org/w3c/jigsaw/proxy/ProxyFrame.java:

```
try {
    DirectoryResource dir =
        (DirectoryResource)root.lock();
    ...
} catch (InvalidResourceException ex) {
    root.unlock();
}
```

Figure 7. Mismatched pairing: the programmer placed the call to `unlock()` in the `catch` instead of a `finally` block.

```
sm lockunlock {
  state decl anyobject v;

start:
  { v."lock"(...) } ==> v.locked;

v.locked:
  { v."unlock"(...) } ==> v.stop
  | $end_of_path$ ==> {
        /* flag error */ };
}
```

Figure 8. Lock checker: flags where intraprocedural lock/unlock pairing is violated

violated: by placing the call to B into the finally clause, the programmer ensures that B is called no matter which path is taken to leave the try block.

Nevertheless, the compiler cannot enforce proper use of try/finally. Figure 7 shows a bug in Jigsaw 2.2.2 where the programmer mistakenly placed the required call to unlock in a catch instead of a finally clause.

Figure 8 shows a simple MJ program that detected this bug: this checker matches invocations of `v.lock()` and tracks the locked instance `v`. If a call to `unlock()` is encountered that is invoked on the same object, we stop tracking the object. If a tracked object reaches *end_of_path*, i.e., an edge in the control flow graph that leaves the current function, a potential error is signaled.

This error occurred despite a comment by the implementer of the `ResourceReference` interface that explicitly recommends the use of the try/finally pattern when using this interface. The length of that comment is about the same as the length of checker; we conclude that analysis tools such as MJ could be used as enforceable documentation. It is also possible to develop checkers in conjunction with the use of automatic interface extraction tools, such as [21].

5.3. Redundant Operations

Redundancies, such as redundant assignments, have been shown to be correlated with actual bugs in C code [22]. To test whether this holds true for Java code as well, we implemented an MJ extension, `neverused`, that identifies such redundancies.

Figure 9 shows an example of a bug this checker found in the OceanStore system: a function that needs to update a value in a hashtable did not, presumably because the programmer mistakenly assumed that an assignment to a reference that was obtained from the hashtable via `get()` would affect the value stored in the hashtable. Our checker flags this example because the result of the call to `msg.getMsgs()` is never used.

OceanStore ostore/apps/visdemo/Visdemo.java:

```
public void registerReqMsg(SonarReqMessage msg) {
    SecureHash treeId = ...  /* ... */
    QSVector network_msg =
        (QSVector)_network_msgs.get(treeId);

    if(msg.getType() == SonarReqMessage.NETWORK &&
        network_msg != null)
        network_msg = msg.getMsgs();
    /* network_msg is dead here, needed was:
      _network_msgs.put(treeId, network_msg) */
}
```

Figure 9. A bug from a redundancy: a return result is never used

Our checker looks for instance and static field reads, object allocations, and instance and static method invocations whose results are never used. We ran this checker over our codebases; we found that reads from static fields that are unused are often the result of defensive programming: a programmer might initialize a local variable to some safe value. We found numerous instances of unused reads of instance fields; in some cases one field is read but a different field is used.

To determine whether a method call or object allocation whose return value is discarded is a bug or not, we attempted to use statistical ranking [15]. We compared the number of sites n where a given method or constructor was invoked to the number of sites e where the return value of the invocation was used. The underlying assumption is that if a method's result is used almost all of the time, then the instances in which it is discarded are likely errors. We found that using statistical ranking in this way worked well in that it pushed redundant calls to the top. We were surprised by the sheer number of redundant calls that can be found in mature Java code. However, statistical ranking did not push the most serious bugs to the top; most top-ranked redundant calls were simply inefficiencies: a programmer would ignore and then repeat the call. Still, we believe that a diligent programmer would want to inspect all such instances and clean them up.

To our surprise, we found that a particular class of bugs showed up at the top of the ranking for unused objects: namely exceptions. Unused exceptions typically occur if a programmer forgets a preceding `throw` keyword. For instance, in Ptolemy, objects of type `ptolemy.kernel.util.IllegalActionException` are used 838 out of 839 times, ranking the one occurrence where a programmer erroneously omitted the throw at the top. After having the obvious-in-hindsight realization that exceptions are practically never constructed for side-effect, we wrote a smaller checker that identifies this very case. The MJ pattern {`anynew java.lang.Throwable()`} allowed for easy identification of allocation sites of all throwables. This example demonstrates a strength of MJ: it makes it easy to create checkers tailored to specific bug patterns that emerge while analyzing actual code.

5.4. Inferring Application-specific Properties

MJ also allows the construction of analyses that infer programming rules from programmer behavior. For instance, different implementations of a virtual method of a class or interface are expected to abide by the same contract. Most implementations are error-free and will exhibit similar behavior. Implementations that deviate from the dominant behavior are likely bugs.

As an example of such an analysis, we analyzed how methods treated object references they received as arguments. We modified our nullchecker to track incoming parameters and recorded if a parameter was checked and dereferenced (C), or dereferenced unchecked (U), or not dereferenced at all. We only included virtual and interface methods with more than one implementation and grouped methods by their shared supermethod. We expect that if a parameter is almost always checked before being dereferenced, then dereferencing it without checking is a bug. Conversely, if it is almost always accessed without a check, the check might be redundant.

For instance, in OpenMap we identified 5 methods for which $C \neq 0$ and $U \neq 0$. We used statistical analysis to sort the results based on the ratio of $C/(C+U)$, adjusted for total number. The three top-ranked entries had $C_1 = 23, U_1 = 1$, $C_2 = 23, U_2 = 1$, $C_3 = 17, U_3 = 2$. The developers of OpenMap confirmed that all 4 unchecked accesses were indeed bugs. For the other packages we analyzed, the analysis failed to find obvious bugs that we could confirm without deeper knowledge of the system.

5.5. Generalizing Identified Bugs

Easy-to-use tools such as MJ can change the way programmers write test cases for bugs they encounter and fix.

```
boolean checkIPRange(Request r, User u) {
  byte abyte0[] = ... get IP address
  /* ... */
  long l = 0L;
  for(int i = 0; i < 4; i++)
    l = l << 8 | abyte0[i];
  /* ... */
}
```

Figure 10. Bug caused by Java's automatic sign extension

```
sm badbyteor {
    state decl { int } x;
    state decl { long } lx;

start:
    { x = byte [ int ] } ==> x.castfrombyte;

x.castfrombyte:
    { lx = (long)x } ==> lx.castfrombyte
    | { x | int } ==> {
        /* possible error */ }, x.stop;

lx.castfrombyte:
    { lx | long } ==> {
        /* possible error */ }, lx.stop;
}
```

Figure 11. Checker that flags if a value read from a byte array is ORed with an int or long

An anecdote illustrates this conviction. A colleague working for an ecommerce company asked the first author for help with a problem in a third-party library used to implement IP-address based access control. Figure 10 shows a decompiled picture of the relevant code portion. Their code worked fine on the company's internal (10.0.0.x) network, but failed for a customer who used IP addresses in the range 160.x.x.x, whose representation in a byte array contains negative values (160 = -96). The bug was caused because a programmer disregarded Java's rules for sign expansion: all scalars, including bytes, in Java are signed, and promotion to `int` or `long` extends the sign, which flooded the higher-order bits of the IP address with 1s.

Figure 11 shows a MJ program that captures the essence of this bug. In the start state, it matches all reads from byte arrays. The integer value read from the array is tracked in a state variable. The program flags if that value is used in a bitwise OR. (If the programmer means to treat the value as an unsigned value, it would likely be used in a bitwise AND with constant 0xff.) To account for the situation shown in the example, we also match the case where the value is cast to a 64-bit long, track the resulting long value and flag if that long value is used in a bitwise OR comparison. Having written this checker, we tested it against our other code bases. Indeed, in Jigsaw class `org.w3c.www.mux.MuxReader`, we found this statement

```
a[i] = (buffer[bufptr] |
        (buffer[bufptr+1] << 8)) & 0xffff;
```

in method `msgShortArrayToIntArray()`.

This example shows that MJ programs can in some cases take the place of unit tests: a programmer who finds and fixes an unexpected bug can generalize the underlying root cause of the bug in a checker.

5.6. Summary of Results

Figure 12 summarizes the number of bugs that were found by four of our checkers that check language-specific properties. We only counted those bugs that were either confirmed by the developers or that we could verify without knowledge of the system in question. For the A&O and the nullchecker, we also list the number of false positives (FPs), and potential bugs or redundant checks (labeled "Bugs?"). A major cause of false positives for the nullchecker are false paths, which we expect to rectify by including path-sensitivity to our analysis. The table does not include the bugs we found with more specialized analyses, and it does not include the many anomalies found by the neverused checker, which included several confirmed bugs. It also does not include the bugs found in the commercial closed-source codebase.

6. Related Work

A variety of bug detection and checking tools have been developed for unsafe languages such as C and C++. These include annotation-based approaches [9, 11], tools that cover a specific set of error types [4, 19], such as buffer overrun errors, and to tools that allow users to extend them to check new properties [2, 5, 7, 8]. In the remainder of this section, we focus on the work that has been done in the Java context.

The most closely related project to ours is the FindBugs [14] system. FindBugs is a framework for writing bug detectors for Java code. It provides an API to plugins that check for particular bug patterns, and like MJ provides support for presenting bugs to the user. Unlike MJ, FindBugs does not provide language support for writing bug detectors. All bug detector plugins have to be written in Java and must interact with FindBugs's API. Another difference is that FindBugs is based on a bytecode-engineering library [6] and does not have the support of an underlying compiler infrastructure like JOEQ.

Jlint and Jlint2 [1] are other examples of tools that support a fixed set of analyses. Jlint2 has been used successfully to find concurrency-related and other bugs in Java code using global static analysis.

ESC/Java and Houdini [10, 17] are annotation-based tools that use a theorem prover to check invariants in Java code. While ESC/Java is able to prove much stronger properties, requiring annotations limits its scalability.

7. Conclusion

Static analysis is a promising method for finding and eliminating bugs at compile time in Java. Key for making

Package	LoC	stringchecker	dropped-exceptions	apples&oranges		nullchecker			Total Bugs
				Bugs	FP	Bugs	Bugs?	FP	
OceanStore	166K	5	0	2	0	5	5	13	12
JDK 1.4 javax	132K	1	0	0	2	9	2	22	10
Ptolemy	87K	4	1	0	0	1	1	1	6
Jigsaw	68K	4	0	0	1	1	13	13	5
OpenMap	124K	4	19	0	0	8	4	0	31
Mozilla Rhino	31K	0	0	0	0	1	4	3	1
Netbeans	217K	4	2	0	3	3	5	18	9
Total	825K	22	22	2	6	28	34	70	74

Figure 12. Bugs and false positives from four of our generic checkers.

static analysis work is the availability of flexible tools that allow developers to devise specific analyses for the specific sources of problems that exist within their application or system. MJ allows the construction of such analyses; we have demonstrated its flexibility by applying it to a variety of bug-finding analyses.

References

[1] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proc. of the 13th Australian Software Engineering Conference (ASWEC'01)*, pages 68–75, 2001.

[2] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.

[3] BBN Technologies. OpenMap™. A JavaBeans™-based toolkit for geographical applications, http://openmap.bbn.-com.

[4] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[5] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. of the 9th ACM Conference on Computer and Communications Security*, pages 235 – 244. ACM Press, 2002.

[6] M. Dahm et al. Byte code engineering libray (BCEL). http://jakarta.apache.org/bcel/.

[7] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, Sept. 2000.

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

[10] C. Flanagan, K. Rustan, and M. Leino. Houdini, an annotation assistant for ESC/Java. In *Symposium of Formal Methods Europe*, pages 500–517, Mar. 2001.

[11] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[13] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[14] D. Hovemeyer and B. Pugh. FindBugs - a bug pattern detector for Java. http://www.cs.umd.edu/˜pugh/java/bugs/.

[15] T. Kremenek and D. Engler. Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proc. of the 10th Annual International Static Analysis Symposium (SAS 2003)*, San Diego, CA, June 2003.

[16] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, Mar. 2003.

[17] K. Rustan, M. Leino, G. Nelson, and J. Saxe. ESC/Java user's manual. Technical note 2000-002, Compaq Systems Research Center, Oct. 2001.

[18] J. D. Ullman. A survey of data flow analysis techniques. In *Second USA-Japan Computer Conference*, pages 335–342, 1975.

[19] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Networking and Distributed System Security Symposium 2000*, San Diego, CA, Feb. 2000.

[20] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In *Proc. of the SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators (IVME03)*, San Diego, CA, June 2003.

[21] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. of the ACM International Symposium of Software Testing and Analysis*, pages 218–228, Rome, Italy, July 2002.

[22] Y. Xie and D. Engler. Using redundancies to find errors. In *Proc. of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 2002)*, Charleston, SC, Nov. 2002.