On the Construction of Soundness Oracles

Jens Dietrich *  Li Sui  Shawn Rasheed  Amjed Tahir
School of Engineering and Advanced Technology, Massey University, Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz, leesui0207@gmail.com, s.rasheed@massey.ac.nz, A.Tahir@massey.ac.nz

Abstract

One of the inherent advantages of static analysis is that it can create and reason about models of an entire program. However, mainstream languages such as Java use numerous dynamic language features designed to boost programmer productivity, but these features are notoriously difficult to capture by static analysis, leading to unsoundness in practice. While existing research has focused on providing sound handling for selected language features (mostly reflection), based on anecdotal evidence and case studies, there is little empirical work to investigate the extent to which particular features cause unsoundness of static analysis in practice. In this paper, we (1) discuss language features that may cause unsoundness and (2) discuss a methodology that can be used to check the (un)soundness of a particular static analysis, call-graph construction, based on soundness oracles. These oracles can also be used for hybrid analyses.

CCS Concepts → Software and its engineering → Automated static analysis: Dynamic analysis

Keywords  Static analysis, Soundness, Dynamic analysis

1. Introduction

Static analysis is used to build models of software and then to reason about these models in order to detect design flaws, bugs and vulnerabilities. An inherent advantage of static analysis is that it can be, at least in principle, sound: it can cover the entire program. In order to be useful, these models use abstractions, and this has an impact on their precision: they over-approximate actual program behaviour. Dynamic analysis on the other hand uses a driver (harness) to execute (exercise) the actual program and build models by observing this execution. It therefore only represents actual program behaviour, but misses behaviour not triggered by the driver. It is therefore precise, but inherently unsound.

Unfortunately, in practice, static analysis is not sound either. Languages like Java are full of dynamic features invented to boost programmer productivity and facilitate the design of frameworks. These features are notoriously difficult to capture in static analysis, and as a result, static analysis tools are unsound. What is more, attempts to improve soundness often cause a loss of precision, as features have to be modelled using over-approximation. The authors of the Soundness Manifesto put it like this: “we are not aware of a single realistic whole-program analysis tool (for example, tools widely used for bug detection, refactoring assistance, programming automation, and so forth) that does not purposefully make unsound choices” [18].

Figure 1 illustrates this situation.

While there is a significant body of work on improving the soundness of static analysis, mostly by improving the handling of reflection, there is little work on how to systematically study the soundness of static analysis tools and methods for real world programs. Such a study requires representative corpora of programs, and soundness oracles: actual program behaviour obtained by means other than static analysis. Such oracles can then be used to detect, catalogue and measure the gaps in static analysis.

In this paper, we discuss and compare several methods that can be used to construct such soundness oracles. Our work only considers one particular kind of program analysis: call graph construction. However, this is a foundational analysis widely used for bug and vulnerability detection, and intertwined with points-to / alias analysis (call graph construction on-the-fly [23]).

We also present a list of language features that are potential sources of unsoundness. While many of these features

* This project was supported by a gift from Oracle Labs Australia to the first author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
have been discussed in previous research, there are some more exotic features that are not widely known. For those cases, we provide some examples illustrating the problems a static analysis would encounter.

2. Related Work

There are two seminal papers that set the scene for our work: Ernst [9] explored the “synergy and duality” of static and dynamic analysis, and argued to combine them in order to “blend the strength of both approaches”. Our approach adapts this idea by generating some (but not all) soundness oracles by means of dynamic analysis. More recently, Livshits et al. [18] published the “In Defense of Soundiness” manifesto where they emphasize the need to make unsound choices. They then went on to issue “a call to the community to identify clearly the nature and extent of unsoundness in static analyses”. This work is in direct response to this call.

Research on improving the soundness of static analysis has focused on reflection handling. A good, up-to-date overview of work in this space is given by Landman et al [17]. We focus the discussion on certain core milestone papers. Livshits et al [19] have investigated support for reflection in the context of points-to analysis. They do this by associating objects with reflective call sites, and then track strings that are used as class names. Type information from cast statements is used to improve precision, taking advantage of the fact that often cast statements are used at reflective instantiation sites. Smaragdakis et al [24] have improved this approach further by adding substring and string flow analysis. This work uses the doop framework [7].

Bodden et al. [6] proposed Tamiflex – a tool to improve the soundness of static analysis with soot [27]. Tamiflex uses a hybrid analysis with additional information inferred from exercising the program under analysis. Therefore, the quality of the analysis strongly relies on the driver (harness) provided by the user. Bodden has also proposed support for invokeDynamic in soot [5], although this only supports the parsing and representation of invokeDynamic, not the actual reasoning.

Sridharan et al. proposed the “framework for frameworks” [23] – an approach to deal with dynamic language features in frameworks. The idea is to improve the analysis by taking configuration files into account, given that many frameworks store data used in reflection (like class names) in files (deployment descriptors, component metadata, etc). Some programming tools implement a similar strategy, for instance, some refactoring browsers embedded in Integrated Development Environments (IDEs) will apply refactorings to framework files like web.xml.

To summarise, these papers illustrate three different approaches to deal with the unsoundness problem, as follows: (1) make static analysis more expressive and try to model and reason about dynamic language features [19] [24], (2) use a hybrid analysis [6] or (3) use contextual information on how dynamic features are actually being used in programs [25].

3. Dynamic Language Features in Java
3.1 Overview

We provide a brief overview of language features that are difficult to handle in static analysis tools. The list contains some of the usual suspects, such as reflection, plus some more exotic features not widely known. These features are not necessarily orthogonal. For instance, embedded scripting may use other dynamic features such as reflection, invoke-dynamic or dynamic class loading depending on whether scripts are interpreted or compiled. But treating it separately is warranted in our opinion as the semantics of scripts could be exploited by static analysis tools.

3.2 Reflection

Java has a rich reflection API that is widely used in practice. There are several levels of reflection APIs in Java, from low level access to methods and fields, to libraries offering a higher level of abstraction like java.beans.Introspector. Reflection is often used for design patterns like service locator or dependency injection (Spring, OSGi DS, etc) [10], an example of such a mechanism with built-in platform support is java.util.ServiceLoader that locates “services” libraries advertised in component (library) meta data.

3.3 Class Loading

Java programs use dedicated class loaders to initialise classes, which can be customised. Since the identity of classes is based on the fully qualified class name and the class loader used, class loaders provide a solution for DLL hell style problems: the co-existence of multiple inconsistent versions of the same class. This make the use of class loaders popular in module frameworks like OSGi where “plugin” components have private class paths. The problem for call graph construction is that (1) not all classes present at runtime are visible to static analysis, (2) not all of the possible actual types of objects are known, which is necessary for precise devirtualisation.

Interestingly, class unloading is also possible and now widely used in enterprise servers, usually by using the OSGi platform supporting the unloading of bundles. The main application is to facilitate hot upgrades. Static analysis could also exploit this by reasoning about program behaviour that is unreachable as classes have been unloaded by then. However, this would only improve precision, not soundness.

3.4 Proxies

Proxies are a language feature to “fake” interfaces by delegating calls to implemented methods to invocation handlers. They are the Java equivalent of Smalltalk’s famous doesNotUnderstand protocol. The main original use case
is the representation of remote objects (client stubs) as an alternative to the static generation of code from (Java or CORBA IDL) interfaces. But proxies have found multiple other applications, such as mock testing (Mockito) or to implement aspect-oriented programming (AOP) (Spring). The problems for static analysis caused by proxies are to correctly model the invocation handlers.

### 3.5 Serialization

Object serialization is used to serialize object graphs, the main use cases are persistence, remoting and deep copying. There are two serialization mechanisms built into Java, binary and XML-based, plus numerous third party solutions. While XML based serialization internally uses reflection, binary serialization is directly supported by the JVM. When a program uses serialization, certain call sites become known only at runtime, and this behaviour is difficult to capture in call graph construction. Binary serialization can use special class loaders, for instance to load remote classes when used in RMI.

### 3.6 Instrumentation

Instrumentation is the addition of bytecode. There are several use cases, including (1) gathering data, for instance, to be used by instrumenting profilers, and (2) adding behaviour, for instance, in order to address cross-cutting concerns in AOP. While early generation instrumentation was static, applied by a pre- or post-compiler to either source- or bytecode respectively, modern instrumentation techniques can be applied late when classes are loaded. This causes problems for static analysis tools that do not have access to the instrumented code.

### 3.7 Native Methods and Incoming Calls

Many Java programs contain native methods, and this is problematic for static analysis tools that cannot trace these calls. But the opposite is also true: native (or rather, external) code can invoke Java methods, using mechanisms such as RMI, CORBA or Protocol Buffers. An example for a complex, bi-directional communication with the platform is java.nio.file.WatchService. The documentation stipulates that “The implementation that observes events from the file system is intended to map directly on to the native file event notification facility where available, or to use a primitive mechanism, such as polling, when a native facility is not available.” This could include native method calls (when polling is used), or mechanisms where system functions call Java methods (when native file event notification is used).

### 3.8 Unsafe

Java contains the class `sun.misc.Unsafe` that provides an API for several low-level functions. There are ongoing attempts to restrict access to this class in Java 9, however, it turns out that it is still widely used in real-world programs. As an example for how Unsafe can cause problems for static analysis, consider the code in figure 2.

```java
import sun.misc.Unsafe;
import java.lang.reflect.Field;

class Main {
    public static void main(String[] args) throws Exception {
        Field f = Unsafe.class.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        Unsafe unsafe = (Unsafe) f.get(null);
        Foo o = (Foo)unsafe.allocateInstance(Foo.class);
        System.out.println(o);
    }
}
```

First, the Unsafe singleton instance is accessed using reflection in order to avoid a SecurityException that restricts access to the instance. Then an instance of Foo is created using allocateInstance. Typically, static analysis tools recognise allocation sites by looking for a bytecode instruction new, dup, invokespecial <init>. However, in this case an instance is created with invoke-virtual sun/misc/Unsafe.allocateInstance. Missing allocation sites may result in missing call graph edges if call graph construction on-the-fly is used.

### 3.9 Invokedynamic

The invokedynamic instruction was introduced in Java 7 and is used by the Java compiler in Java 8 to compile lambdas. Invokedynamic gives the user more control over the dispatch mechanism by resolving call targets at runtime via a user-defined bootstrap method. In Java 8, this is only used for the compilation of lambdas, and bootstrapping uses a fixed set of methods in java.lang.invoke.LambdaMetafactory. However, a non-standard Java compiler like dynamo or a compiler for a different JVM language may produce code that uses a different bootstrapping mechanisms, and therefore static analyses should not rely on the presence of the LambdaMetafactory.

### 3.10 Scripting and Runtime Compilation

In recent years, domain-specific languages (DSL) have become increasingly popular and are now widely used. Examples include embedded full-fledged Turing-complete languages like the Rhino ECMA Script engine embedded in Java, expression languages like JSP EL, OGNL and MVEL, templating languages like velocity and StringTemplate, and query and transformation languages like sql, xpath and xslt.
When expressions in a language are evaluated, (Java) methods of the host program may be invoked. To illustrate this, consider the following example in figure 3.

This shows how the MVEL expression and template engine compiles and evaluates an expression. According to the semantics of MVEL, `bean.value` will get interpreted as a method call to `getValue()`, and therefore, a sound call graph analysis tool needs to compute a path from `Main.main()` to `Bean.getValue()`.

To use DSLs at runtime, interpretation or compilation of DSL code is required. This can range from interpretation using Java reflection to full-fledged bytecode generation combined with classloading. For instance, the xalan-2.7.2 compiler `org.apache.xalan.xslt.compiler.XSLTC` generates bytecode using the `bcel` library.

Depending on whether scripts are interpreted or compiled, method calls may or may not be reflective. But even if a compiler that produces bytecode was used, a static analysis would still have problems as it might not be able to access the code. It is for instance possible to compile completely in memory, without ever storing a class definition on a disk.

There are standard APIs to facilitate the integration of DSLs (JSR223 [2]) and runtime compilation (JSR199 [1]). Examples of real-world libraries using these APIs are `jmeter-4.0` (JSR223) and `drools-7.0.0` (JSR199).

4. **Soundness Oracle Generation**

4.1 **Introduction**

Assuming that all static analysis tools encounter the same problems, such as “how to deal with reflection”, soundness oracles should be generated by means other than static analysis. One could argue that there is some benefit in generating oracles with a static analysis tool A and use it to assess static analysis tool B, but we believe that there is limited benefit to this. When discussing methods to generate soundness oracles, we are interested in three aspects:

1. **Size**: larger oracles are generally better, as they are more likely to cover the gaps.
2. **Quality**: oracles should contain interesting behaviour: program behaviour leading to bugs or vulnerabilities, or behaviour resulting from the use of dynamic language features and likely to be missed by static analysis.
3. **Effort**: it should be easy to generate oracles, preferably, the process should be automated.

In the context of call graph construction, such a soundness oracle contains call graph edges.

4.2 **On-board Tests and Executables**

The obvious way to generate soundness oracles is by means of dynamic analysis. Programs are exercised starting with existing program entry points (main(String[])) for standalone executables, `doGet(HttpServletRequest, HttpServletResponse)` for web applications, etc). This can be facilitated by crafting dedicated “drivers”, this is the approach taken by DaCapo [4]. The executing program is then observed using injected code (instrumentation) or by acquiring thread dumps (sampling), and the oracle is built from the results of this observation.

The main problem with this approach is that it is difficult to find or create drivers that exercise a large percentage of relevant program behaviour. This can be approximated by measuring branch coverage: low coverage indicates that large parts of the program are never exercised. For instance, the average coverage obtained when using the DaCapo driver is rather low (16.10%) [3].

A potentially better approach to exercise programs is the use of unit tests. The use of frameworks like `junit` is common practice in many Java programs, and many software engineers are aware of coverage metrics and aim for high coverage. Unfortunately, we found that the coverage obtained by running built-in tests on real world programs is still relatively low [3], impacting the size of the oracle. On the other hand, the oracles generated with built-in tests should contain interesting and relevant program behaviour as programmers are more likely to write tests for functionality that uses complex features, in particular when this has caused problems in the past and test cases were created as part of documenting and fixing bugs in the spirit of test-driven development [3].

The effort needed to generate such an oracle is moderate. Test case execution is usually easy to automate, in particular as many projects use canonical project layouts and naming conventions (such as the Maven project layout). To execute main methods and similar program entry points usually requires meaningful runtime parameters, and finding those can be challenging.
4.3 Generated Tests

A viable alternative to existing tests is test case generation. There are several tools available for Java and other platforms. We have experimented with Evosuite [12], a test generation framework that uses an evolutionary algorithm to generate test suites. There are a number of similar tools such as Randoop [22] and Pex [26] (for .NET).

Test generation tools aim for high coverage. In a study on the XCorpus [8], a subset of 70 Java programs from the Qualitas Corpus v 20130901, we observed an average branch coverage of 55.86%. This compares favourably with the coverage of only 34.42% obtained with built-in tests for the 28 / 70 programs that had tests. The coverage of oracle problem is in particular, many users report problems along with stack traces. Since these stack traces have a canonical format, they can be gathered and parsed. Many of the languages features that cause unsoundness use particular exceptions or errors to report problems, such as:

1. InstantiationException – reflective instantiation via Class.newInstance()
2. InvocationTargetException – reflective method calls via Method.invoke(...)
3. UncheckedThrowingException – reflective method calls via InvocationHandler.invoke(...) used for proxies
4. javax.script.ScriptException – may include a (reflective) call to a method when a script is evaluated

When users encounter and report these exceptions, the stack traces often reveal call graph edges associated with the use of dynamic language features. For instance, the following stack trace was reported on StackOverflow (https://goo.gl/bfFRh0):

```
Processing main file...
Unable to start FOP: java.lang.reflect.
InvocationTargetException
  at ... at java.lang.reflect.Method.invoke(Unknown Source)
  at org.apache.fop.cli.Main.
  startFOPWithDynamicClasspath(Main.java:133)
  at org.apache.fop.cli.Main.main(Main.java:207)
  Caused by: javax.lang.RuntimeExec
  at java.lang.Runtime.exec(Unknown Source)
```

This reveals a reflective method call of org...Main.-
  startFOP using a call site in org...Main.startFOP-
  WithDynamicClasspath(). Th extraction of stacktraces from online sources can be automated to some extent. A challenge is the extraction of correct versioning information.

4.4 Stacktrace Mining

Software repositories like GitHub and Bitbucket and forums like StackOverflow are rich sources of information. In particular, many users report problems along with stack traces. The respective reflective call site is in org.apache.commons.collections.functors.InvokerTransformer#transform.

Many known vulnerabilities follow the same pattern, including CVE-2016-2510 (BeanShell) and CVE-2015-3253 (Groovy). The extraction of call graph edges from these vulnerabilities is expensive as it has to be performed manually, however, the results are usually of high value as they can be used to test static analysis tools that detect vulnerabilities. In addition to the official CVE registry, there are numerous other security-related discussion forums and websites that are valuable sources of “exotic behaviour”, such as Chris Frohoff’s yssorerial repository [13], which contains payloads for serialization-related vulnerabilities, including the call chains for CVE-2015-7450. There is also an example of how embedded compilation and class loading can be used to construct call graph edges and chains, respectively. In the Spring1 call chain [13], the embedded XSLT compiler (org.apache.xalan.xslt.TemplatesImpl) is used to generate and load a class defined from data read from an incoming stream. The initialisation of this class


https://github.com/ysoserial [accessed 17 March 17]
then triggers the static block (<clinit>) to execute, which then invokes Runtime.exec().

4.6 Summary
Table 1 provides an overview of the methods discussed.

<table>
<thead>
<tr>
<th>method</th>
<th>quality</th>
<th>size</th>
<th>effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>on-board executables</td>
<td>high</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>generated tests</td>
<td>medium</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>stacktrace mining</td>
<td>high</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>cve analysis</td>
<td>high</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>

Table 1. Methods to generate soundness oracles

5. Conclusion
In this paper, we have discussed Java language features that cause static analysis tools to make unsound choices. We then discussed four different approaches to build soundness oracles by means of dynamic analysis and mining data sources. The next step is to create those oracles, and to assess the output of static analysis tools against them.

References