On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation

Li Sui[1(✉)], Jens Dietrich[2], Michael Emery[1], Shawn Rasheed[1], and Amjed Tahir[1]

[1] Massey University Institute of Fundamental Sciences, 4410 Palmerston North, New Zealand
{L.Sui,S.Rasheed,A.Tahir}@massey.ac.nz
[2] Victoria University of Wellington School of Engineering and Computer Science, 6012 Wellington, New Zealand
jens.dietrich@ecs.vuw.ac.nz
http://ifs.massey.ac.nz/, https://www.victoria.ac.nz/ecs

Abstract. Static program analysis is widely used to detect bugs and vulnerabilities early in the life cycle of software. It models possible program executions without executing a program, and therefore has to deal with both false positives (precision) and false negatives (soundness). A particular challenge for sound static analysis is the presence of dynamic language features, which are prevalent in modern programming languages, and widely used in practice.

We catalogue these features for Java and present a micro-benchmark that can be used to study the recall of static analysis tools. In many cases, we provide examples of real-world usage of the respective feature. We then study the call graphs constructed with *soot*, *wala* and *doop* using the benchmark. We find that while none of the tools can construct a sound call graph for all benchmark programs, they all offer some support for dynamic language features.

We also discuss the notion of possible program execution that serves as the ground truth used to define both precision and soundness. It turns out that this notion is less straight-forward than expected as there are corner cases where the (language, JVM and standard library) specifications do not unambiguously define possible executions.

Keywords: Static analysis · Call graph construction · Soundness
Benchmark · Java · Dynamic proxies · Reflection
Dynamic class loading · Invokedynamic · sun.misc.Unsafe · JNI

This work was supported by the Science for Technological Innovation (SfTI) National Science Challenge (NSC) of New Zealand (PROP-52515-NSCSEED-MAU). The work of the second author was supported by a faculty gift by Oracle Inc.

S. Ryu (Ed.): APLAS 2018, LNCS 11275, pp. 69–88, 2018.
https://doi.org/10.1007/978-3-030-02768-1_4

1 Introduction

Static analysis is a popular technique to detect bugs and vulnerabilities early in the life cycle of a program when it is still relatively inexpensive to fix those issues. It is based on the idea to extract a model from the program without executing it, and then to reason about this model in order to detect flaws in the program. Superficially, this approach should be sound in the sense that all possible program behaviour can be modelled as the entire program is available for analysis [13]. This is fundamentally different from dynamic analysis techniques that are inherently unsound as they depend on drivers to execute the program under analysis, and for real-world programs, these drivers will not cover all possible execution paths. Unfortunately, it turns out that most static analyses are not sound either, caused by the use of dynamic language features that are available in all mainstream modern programming languages, and prevalent in programs. Those features are notoriously difficult to model.

For many years, research in static analysis has focused on precision [31] - the avoidance of false positives caused by the over-abstraction of the analysis model, and scalability. Only more recently has soundness attracted more attention, in particular, the publication of the soundiness manifesto has brought this issue to the fore [26].

While it remains a major research objective to make static analysis sound (or, to use a quantitative term, to increase its recall), there is value in capturing the state of the art in order to explore and catalogue where existing analysers fall short. This is the aim of this paper. Our contributions are: (1) a micro-benchmark consisting of Java programs using dynamic language features along with a call graph oracle representing possible invocation chains, and (2) an evaluation of the call graphs constructed with *soot*, *wala* and *doop* using the benchmark.

2 Background

2.1 Soundness, Precision and Recall

We follow the soundness manifesto and define the soundness of a static analysis with respect to possible program executions: "analyses are often expected to be sound in that their result models *all possible executions* of the program under analysis" [26]. Similarly, precision can be defined with respect to possible executions as well – a precise analysis models *only possible executions*.

Possible program executions are the *ground truth* against which both soundness and precision are defined. This can also be phrased as the absence of false negatives (FNs) and false positives (FPs), respectively, adapting concepts widely used in machine learning. In this setting, soundness corresponds to *recall*. Recall has a slightly different meaning as it is measurable, whereas soundness is a quality that a system either does or does not possess.

2.2 Call Graphs

In our study, we focus on a particular type of program behaviour: method invocations, modelled by (static) call graphs [18,32]. The aspect of possible executions to be modelled here are method invocations, i.e. that the invocation of one *source* method triggers the invocation of another *target* method. Another way to phrase this in terms of the Java stack is that the *target* method is above the *source* method on the stack at some stage during program execution. We use the phrases *trigger* and *above* to indicate that there may or may not be intermediate methods between the source and the target method. For instance, in a JVM implemented in Java, the stack may contain intermediate methods between the source and the target method to facilitate dispatch.

Static call graph construction has been used for many years and is widely used to detect bugs and vulnerabilities [32,38,40]. In statically constructed call graphs (from here on, called call graphs for short), methods are represented by vertices, and invocations are represented by edges. Sometimes vertices and edges have additional labels, for instance to indicate the invocation instructions being used. This is not relevant for the work presented here and therefore omitted. A source method invoking a target method is represented by an edge from the (vertex representing the) source method to the (vertex representing the) target method. We are again allowing indirect invocations via intermediate methods, this can be easily achieved by computing the transitive closure of the call graph.

2.3 Java Programs

The scope of this study is Java, but it is not necessarily obvious what this means. One question is which version we study. This study uses Java 8, the version widely used at the time of writing this. Due to Java's long history of ensuring backward compatibility, we are confident that this benchmark will remain useful for future versions of Java.

Another question is whether by Java we mean programs written in the Java language, or compiled into JVM byte code. We use the later, for two reasons: (1) most static analysis tools for Java use byte code as input (2) by using byte code, we automatically widen the scope of our study by allowing programs written in other languages that can be compiled into Java byte code.

By explicitly allowing byte code generated by a compiler other than the (standard) Java compiler, we have to deal with byte code the standard compiler cannot produce. We include some programs in the benchmark that explicitly take advantage of this. We note that even if we restricted our study to byte code that can be produced by the Java compiler we would still have a similar problem, as byte code manipulation frameworks are now widely used and techniques like Aspect-Oriented Programming [21] are considered to be an integral part of the Java technology stack.

2.4 Possible Program Executions

The notion of possible program execution is used as ground truth to assess the soundness and the precision of call graph construction tools. This also requires a clarification. Firstly, we do not consider execution paths that are triggered by JVM or system (platform) errors. Secondly, none of the benchmark programs use random inputs, all programs are deterministic. Their behaviour should therefore be completely defined by their byte code.

It turns out that there are scenarios where the resolution of a reflective method call is not completely specified by the specification[1], and possible program executions depend on the actual JVM. This will be discussed in more detail in Sect. 5.1.

2.5 Dynamic Language Features

Our aim is to construct a benchmark for *dynamic language features* for Java. This term is widely used informally, but some discussion is required what this actually means, in order to define the scope of this study. In general, we are interested in all features that allow the user to customise some aspects of the execution semantics of a program, in particular (1) class and object life cycle (2) field access and (3) method dispatch. There are two categories of features we consider: (1) features built into the language itself, and exposed by official APIs. In a wider sense, those are reflective features, given the ability of a system "to reason about itself" [36]. Java reflection, class loading, dynamic proxies and `invokedynamic` fit into this category. We also consider (2) certain features where programmers can access extra-linguistic mechanisms. The use of native methods,`sun.misc.Unsafe` and serialisation are in this category. Java is not the only language with such features, for instance, Smalltalk also has a reflection API, the ability to customise dispatch with `doesNotUnderstand`, binary object serialisation using the Binary Object Streaming Service (BOSS), and the (unsafe-like) `become` method [14].

This definition also excludes certain features, in particular the study of exceptions and static initializers (<*clinit*>).

3 Related Work

3.1 Benchmarks and Corpora for Empirical Studies

Several benchmarks and datasets have been designed to assist empirical studies in programming languages and software engineering research. One of the most widely used benchmarks is *DaCapo* [6] - a set of open source, real-world Java programs with non-trivial memory loads. *DaCapo* is executable as it provides a customizable harness to execute the respective programs. The key purpose of this benchmark is to be used to compare results of empirical studies, e.g. to compare the performance of different JVMs. The *Qualitas Corpus* [39] provides

[1] Meaning here a combination of the JVM Specification [2] and the documentation of the classes of the standard library.

a larger set of curated Java programs intended to be used for empirical studies on code artefacts. *XCorpus* [10] extends the Qualitas Corpus by adding a (partially synthetic) driver with a high coverage.

SPECjvm2008 [3] is a multi-threaded Java benchmark focusing on core Java functionality, mainly the performance of the JRE. It contains several executable synthetic data sets as well as real-world programs.

Very recently, Reif et al. [30] have published a Java test suite designed to test static analysers for their support for dynamic language features, and evaluated *wala* and *soot* against it. While this is very similar to the approach presented here, there are some significant differences: (1) the authors of [30] assume that the tests (benchmark programs) "provide the ground truth". In this study, we question this assumption, and propose an alternative notion that also take characteristics of the JVM and platform used to execute the tests into account. (2) The study presented here also investigates *doop*, which we consider important as it offers several features for advanced reflection handling. (3) While the construction of both test suites/benchmarks was motivated by the same intention, they are different. Merging and consolidating them is an interesting area for future research.

3.2 Approaches to Handle Dynamic Language Features in Pure Static Analysis

Reflection: reflection [14,36] is widely used in real-world Java programs, but is challenging for static analysis to handle [22,24]. Livshits et al. [27] introduced the first static reflection analysis for Java, which uses points-to analysis to approximate the targets of reflective call sites as part of call graph construction. Landman et al. [22] investigated in detail the challenges faced by static analysers to model reflection in Java, and reported 24 different techniques that have been cited in the literature and existing tool. Li et al. [24] proposed *elf*, a static reflection analysis with the aim to improve the effectiveness of Java pointer analysis tools. This analysis uses a self-inferencing mechanism for reflection resolution. *Elf* was evaluated against *doop*, and as a result it was found that *elf* was able to resolve more reflective call targets than *doop*. Smaragdakis et al. [35] further refined the approach from [27] and [24] in terms of both recall and performance. *Wala* [12] has some built-in support for reflective features like `Class.forName`, `Class.newInstance`, and `Method.invoke`.

invokedynamic: Several authors have proposed support for `invokedynamic`. For example, Bodden [7] provided a *soot* extension that supports reading, representing and writing `invokedynamic` byte codes. The *opal* static analyser also provides support for `invokedynamic` through replacing `invokedynamic` instructions using Java `LambdaMetaFactory` with a standard `invokestatic` instruction [1]. *Wala* provides support for `invokedynamic` generated for Java 8 lambdas[2].

[2] https://goo.gl/1LxbSd and https://goo.gl/qYeVTd, both accessed 10 June 2018.

Dynamic Proxies: only recently, at the time of writing this, Fourtounis et al. [15] have proposed support for dynamic proxies in *doop*. This analysis shows that there is a need for the mutually recursive handling of dynamic proxies and other object flows via regular operations (heap loads and stores) and reflective actions. Also, in order to be effective, static modelling of proxies needs full treatment of other program semantics such as flow of string constants.

3.3 Hybrid Analysis

Several studies have focused on improving the recall of static analysis by adding information obtained from a dynamic (pre-)analysis. Bodden et al. proposed *tamiflex* [8]. *Tamiflex* runs a dynamic analyses by on instrumented code. The tool logs all reflective calls and feeds this information into a static analysis, such as *soot*. Grech et al. [17] proposed *heapdl*, a tool similar to *tamiflex* that also uses heap snapshots to further improve recall (compared to *tamiflex*). *Mirror* by Liu et al. [25] is a hybrid analysis specifically developed to resolve reflective call sites while minimising false positives.

Andreasen et al. [4] used a hybrid approach that combines soundness testing, blended analysis, and delta debugging for systematically guiding improvements of soundness and precision of TAJS - a static analyser for JavaScript. Soundness testing is the process of comparing the analysis results obtained from a pure static analysis with the concrete states that are observed by a dynamic analysis, in order to observe unsoundness.

Sui et al. [37] extracted reflective call graph edges from stack traces obtained from GitHub issue trackers and Stack Overflow Q&A forums to supplement statically built call graphs. Using this method, they found several edges *doop* (with reflection analysis enabled) was not able to compute. Dietrich et al. [11] generalised this idea and discuss how to generate *soundness oracles* that can be used to examine the unsoundness of a static analysis.

3.4 Call Graph Construction

Many algorithms have been proposed to statically compute call graphs. A comparative study of some of those algorithms was presented by Tip and Palsberg [40]. Class Hierarchy Analysis (CHA) [18] is a classic call graph algorithm that takes class hierarchy information into account. It assumes that the type of a receiver object (at run time) is possibly any subtype of the declared type of the receiver object at the call site. CHA is imprecise, but fast. Rapid Type Analysis (RTA) extends CHA by taking class instantiation information into consideration, by restricting the possible runtime types to classes that are instantiated in the reachable part of the program [5]. Variable Type Analysis (VTA) models the assignments between different variables by generating subset constraints, and then propagates points-to sets of the specific runtime types of each variable along these constraints [38]. k-CFA analyses [34] add various levels of call site sensitivity to the analysis.

Murphy et al. [29] presented one of the earlier empirical studies in this space, which focused on comparing the results of applying 9 static analysis tools (including tools like GNU cflow) for extracting call graphs from 3 C sample programs. As a results, the extracted call graphs were found to vary in size, which makes them potentially unreliable for developers to use. While this was found for C call graph extractors, it is still likely that the same problem will apply to extractors in other languages. Lhoták [23] proposed tooling and an interchange format to represent and compare call graphs produced by different tools. We use the respective format in our work.

4 The Benchmark

4.1 Benchmark Structure

The benchmark is organised as a Maven[3] project using the standard project layout. The actual programs are organised in name spaces (packages) reflecting their category. Programs are minimalistic, and their behaviour is in most cases easy to understand for an experienced programmer by "just looking at the program". All programs have a `source()` method and one or more other methods, usually named `target(..)`.

Each program has an integrated oracle of expected program behaviour, encoded using standard Java annotations. Methods annotated with `@Source` are call graph sources: we consider the program behaviour triggered by the execution of those methods from an outside client. Methods annotated with `@Target` are methods that may or may not be invoked directly or indirectly from a call site in the method annotated with `@Source`. The expectation whether a target method is to be invoked or not is encoded in the `@Target` annotation's `expectation` attribute that can be one of three values: `Expected.YES` – the method is expected to be invoked , `Expected.NO` – the method is not expected to be invoked, or `Expected.MAYBE` – exactly one of the methods with this annotation is expected to be invoked, but which one may depend on the JVM to be used. For each program, *either* exactly one method is annotated with `@Target(expectation=Expected.YES)`, *or* some methods are annotated with `@Target(expectation=Expected.MAYBE`.

The benchmark contains a `Vanilla` program that defines the base case: a single source method that has a call site where the target method is invoked using a plain `invokevirtual` instruction. The annotated example is shown in Listing 1.1, this also illustrates the use of the oracle annotations.

[3] https://maven.apache.org/, accessed 30 August 2018.

```
1  public class Vanilla {
2      public boolean TARGET = false;
3      public boolean TARGET2 = false;
4      @Source public void source(){
5              target();
6      }
7      @Target(expectation = YES) public void target(){
8              this.TARGET = true;
9      }
10     @Target(expectation = NO) public void target(int o) {
11             this.TARGET2 = true;
12     }
13 }
```

Listing 1.1. Vanilla program source code (simplified)

The main purpose of the annotations is to facilitate the set up of experiments with static analysers. Since the annotations have a retention policy that makes them visible at runtime, the oracle to test static analysers can be easily inferred from the benchmark program. In particular, the annotations can be used to test for both FNs (soundness issues) and FPs (precision issues).

In Listing 1.1, the target method changes the state of the object by setting the `TARGET` flag. The purpose of this feature is to make invocations easily observable, and to confirm actual program behaviour by means of executing the respective programs by running a simple client implemented as a junit test. Listing 1.2 shows the respective test for `Vanilla` – we expect that after an invocation of `source()` by the test driver, `target()` will have been called after `source()` has returned , and we check this with an assertion check on the `TARGET` field. We also tests for methods that should not be called, by checking that the value of the respective field remains `false`.

```
1  public class VanillaTest {
2      private Vanilla vanilla;
3      @Before public void setUp() throws Exception {
4          vanilla = new Vanilla();
5          vanilla.source();
6      }
7      @Test public void testTargetMethodBeenCalled() {
8          Assert.assertTrue(vanilla.TARGET);
9      }
10     @Test public void testTarget2MethodHasNotBeenCalled() {
11         Assert.assertFalse(vanilla.TARGET2);
12     }
13 }
```

Listing 1.2. Vanilla test case (simplified)

4.2 Dynamic Language Features and Vulnerabilities

One objective for benchmark construction was to select features that are of interest to static program analysis, as there are known vulnerabilities that exploit those features. Since the discussed features allow bypassing Java's security model, which relies on information-hiding, memory and type safety, Java security vulnerabilities involving their use have been reported that have implications ranging from attacks on confidentiality, integrity and the availability of applications. Categorised under the Common Weakness Enumeration (CWE) classification, untrusted deserialisation, unsafe reflection, type confusion, untrusted pointer dereferences and buffer overflow vulnerabilities are the most notable.

CVE-2015-7450 is a well-known serialisation vulnerability in the Apache Commons Collections library. It lets an attacker execute arbitrary commands on a system that uses unsafe Java deserialisation. Use of reflection is common in vulnerabilities as discussed by Holzinger et al. [19] where the authors discover that 28 out of 87 exploits studied utilised reflection vulnerabilities. An example is CVE-2013-0431, affecting the Java JMX API, which allows loading of arbitrary classes and invoking their methods. CVE-2009-3869, CVE-2010-3552, CVE-2013-08091 are buffer overflow vulnerabilities involving the use of native methods. As for vulnerabilities that use the `Unsafe` API, CVE-2012-0507 is a vulnerability in `AtomicReferenceArray` which uses `Unsafe` to store a reference in an array directly that can violate type safety and permit escaping the sandbox. CVE-2016-4000 and CVE-2015-3253 reported for Jython and Groovy are due to serialisable invocation handlers for proxy instances. While we are not aware of vulnerabilities that exploit invokedynamic directly, there are several CVEs that exploit the method handle API used in the invokedynamic bootstrapping process, including CVE-2012-5088, CVE-2013-2436 and CVE-2013-0422.

The following subsections contain a high-level discussion of the various categories of programs in the benchmark. A detailed discussion of each program is not possible within the page limit, the reader is referred to the benchmark repository for more details.

4.3 Reflection

Java's reflection protocol is widely used and it is the foundation for many frameworks. With reflection, classes can be dynamically instantiated, fields can be accessed and manipulated, and methods can be invoked. How easily reflection can be modelled by a static analysis highly depends on the *usage context*. In particular, a reflective call site for *Method.invoke* can be easily handled if the parameter at the method access site (i.e., the call site of `Class.getMethod` or related methods) are known, for instance, if method name and parameter types can be inferred. Existing static analysis support is based on this wider idea.

However, this is not always possible. The data needed to accurately identify an invoked method might be supplied by other methods (therefore, the static analysis must be inter-procedural to capture this), only partially available (e.g., if only the method name can safely be inferred, a static analysis may decide to

over-approximate the call graph and create edges for all possible methods with this name), provided through external resources (a popular pattern in enterprise frameworks like spring, service loaders, or JEE web applications), or some custom procedural code. All of those usage patterns do occur in practice [22,24], and while exotic uses of reflection might be rare, they are also the most interesting ones as they might be used in the kind of vulnerabilities static analysis is interested to find.

The benchmark examples reflect this range of usage patterns from trivial to sophisticated. Many programs overload the target method, this is used to test whether a static analysis tool achieves sound reflection handling at the price of precision.

4.4 Reflection with Ambiguous Resolution

As discussed in Sect. 2, we also consider scenarios where a program is (at least partially) not generated by `javac`. Since at byte code level methods are identified by a combination of name and descriptor, the JVM supports return type overloading, and the compiler uses this, for instance, in order to support covariant return types [16, Sect. 8.4.5] by generating bridge methods. This raises the question how the methods in `java.lang.Class` used to locate methods resolve ambiguity as they use only name and parameter types, but not the return type, as parameters. According to the respective class documentation, "If more than one method with the same parameter types is declared in a class, and one of these methods has a return type that is more specific than any of the others, that method is returned; otherwise one of the methods is chosen arbitrarily"[4]. In case of return type overloading used in bridge methods, this rule still yields an unambiguous result, but one can easily engineer byte code where the arbitrary choice clause applies. The benchmark contains a respective example, `dpbbench.ambiguous.ReturnTypeOverloading`. There are two target methods, one returning `java.util.Set` and one returning `java.util.List`. Since neither return type is a subtype of the other type, the JVM is free to choose either. In this case we use the `@Target (expectation=MAYBE)` annotation to define the oracle. We acknowledge that the practical relevance of this might be low at the moment, but we included this scenario as it highlights that the concept of possible program behaviour used as ground truth to assess the soundness of static analysis is not as clear as it is widely believed. Here, possible program executions can be defined either with respect to all or some JVMs.

It turns out that Oracle JRE 1.8.0_144/OpenJDK JRE 1.8.0_40 on the one hand and IBM JRE 1.8.0_171 on the other hand actually do select different methods here. We have also observed that IBM JRE 1.8.0_171 chooses the incorrect method in the related `dpbbench.reflection.invocation.ReturnTypeOverloading` scenario (note the different package name). In this scenario, the overloaded target methods return `java.util.Collection` and `java.util.List`, respectively, and the IBM JVM dispatches to the method

[4] https://goo.gl/JG9qD2, accessed 24 May 2018.

returning `java.util.Collection` in violation of the rule stipulated in the API specification. We reported this as a bug, and it was accepted and fixed report[5].

A similar situation occurs when the selection of the target method depends on the order of annotations returned via the reflective API. This scenario does occur in practice, for instance, the use of this pattern in the popular *log4j* library is discussed in [37]. The reflection API does not impose constraints on the order of annotations returned by `java.lang.reflect.Method.getDeclaredAnnotations()`, therefore, programs have different possible executions for different JVMs.

```
1  public class Invocation {
2      public boolean TARGET = false;
3      public boolean TARGET2 = false;
4      @Retention(RUNTIME) @Target(METHOD) @interface Method{}
5      @Source public void source() throws Exception {
6          for (Method method:Invocation.class.getDeclaredMethods()){
7                  if (method.isAnnotationPresent(Method.class)){
8                      method.invoke(this, null);
9                      return;
10     }   }   } }
11     @Method @Target (expectation=MAYBE) public void target(){
12         this.TARGET =true;
13     }
14     @Method @Target (expectation=MAYBE) public void target2(){
15         this.TARGET2 =true;
16     }
17 }
```

Listing 1.3. Example where the selection of the target method depends on the JVM being used (simplified)

When executing those two examples and recording the actual call graphs, we observe that the call graphs differ depending on the JVM being used. For instance, in the program in Listing 1.3, the target method selected at the call site in `source()` is `target()` for both Oracle JRE 1.8.0_144 and OpenJDK JRE 1.8.0_40 , and `target2()` for IBM JRE 1.8.0_171.

4.5 Dynamic Classloading

Java distinguishes between classes and class loaders. This can be used to dynamically load, or even generate classes at runtime. This is widely used in practice, in particular for frameworks that compile embedded scripting or domain-specific languages "on the fly", such as Xalan[6].

There is a single example in the benchmark that uses a custom classloader to load and instantiate a class. The constructors of the respective class are the expected target methods.

[5] https://github.com/eclipse/openj9/pull/2240, accessed 16 August 2018.
[6] https://xalan.apache.org, accessed 4 June 2018.

4.6 Dynamic Proxies

Dynamic proxies were introduced in Java 1.3, they are similar to protocols like Smalltalk's `doesNotUnderstand`, they capture calls to unimplemented methods via an invocation handler. A major application is to facilitate distributed object frameworks like CORBA and RMI, but dynamic proxies are also used in mock testing frameworks. For example, in the *XCorpus* dataset of 75 real-world programs, 13 use dynamic proxies [10] (implement `InvocationHandler` and have call sites for `Proxy.newProxyInstance`). Landman et al. observed that "all [state-of-the-art static analysis] tools assume .. absence of Proxy classes" [22].

The benchmark contains a single program in the `dynamicProxy` category. In this program, the source method invokes an interface method `foo()` through an invocation handler. In the invocation handler, `target(String)` is invoked. The `target` method is overloaded in order to test the precision of the analysis.

4.7 Invokedynamic

The `invokedynamic` instruction was introduced in Java 7. It gives the user more control over the method dispatch process by using a user-defined bootstrap method that computes the call target. While the original motivation behind `invokedynamic` was to provide support for dynamic languages like Ruby, its main (and in the OpenJDK 8, only) application is to provide support for lambdas. In OpenJDK 9, `invokedynamic` is also used for string concatenation [33].

For known usage contexts, support for `invokedynamic` is possible. If `invokedynamic` is used with the `LambdaMetafactory`, then a tool can rewrite this byte code, for instance, by using an alternative byte code sequence that compiles lambdas using anonymous inner classes. The Opal byte code rectifier [1] is based on this wider idea, and can be used as a standalone pre-processor for static analysis. The rewritten byte code can then be analysed "as usual".

The benchmark contains three examples defined by Java sources with different uses of lambdas. The fourth examples is engineered from byte code and is an adapted version of the dynamo compiler example from [20]. Here, `invokedynamic` is used for a special compilation of component boundary methods in order to improve binary compatibility. The intention of including this example is to distinguish between `invokedynamic` for particular usage patterns, and general support for `invokedynamic`.

4.8 Serialisation

Java serialisation is a feature that is used in order to export object graphs to streams, and vice versa. This is a highly controversial feature, in particular after a large number of serialisation-related vulnerabilities were reported in recent years [9,19].

The benchmark contains a single program in this category that relates to the fact that (de-)serialisation offers an extra-linguistic mechanism to construct objects, avoiding constructors. The scenario constructs an object from a stream,

and then invokes a method on this object. The client class is not aware of the actual type of the receiver object, as the code contains no allocation site.

4.9 JNI

The Java Native Interface (JNI) is a framework that enables Java to call and be called by native applications. There are two programs using JNI in the benchmark. The first scenario uses a custom `Runnable` to be started by `Thread.start`. In the Java 8 (OpenJDK 8), `Runnable.run` is invoked by `Thread.start` via an intermediate native method `Thread.start0()`. This is another scenario that can be handled by static analysis tools that can deal with common usage patterns, rather than with the general feature. The second program is a custom example that uses a grafted method implemented in C.

4.10 sun.misc.Unsafe

The class `sun.misc.Unsafe` (unsafe for short) offers several low level APIs that can bypass constraints built into standard APIs. Originally intended to facilitate the implementation of platform APIs, and to provide an alternative for JNI, this feature is now widely used outside the Java platform libraries [28]. The benchmark contains four programs in this category, (1) using unsafe to load a class (`defineClass`), (2) to throw an exception (`throwException`), (3) to allocate an instance (`allocateInstance`) and (4) to swap references (`putObject`, `objectFieldOffset`). leads to an error that was r

5 Experiments

5.1 Methodology

We conducted an array of experiments with the benchmark. In particular, we were interested to see whether the benchmark examples were suitable to differentiate the capabilities of mainstream static analysis frameworks. We selected three frameworks based on (1) their wide use in the community, evidenced by citation counts of core papers, indicating that the respective frameworks are widely used, and therefore issues in those frameworks will have a wider impact on the research community, (2) the respective frameworks claim to have some support for dynamic language features, in particular reflection, (3) the respective projects are active, indicating that the features of those frameworks will continue to have an impact.

Based on those criteria, we evaluated *soot-3.1.0*, *doop*[7] and *wala-1.4.3*. For each tool, we considered a basic configuration, and an advanced configuration to switch on support for advanced language features. All three tools have options

[7] As doop does not release versions, we used a version built from commit 4a94ae3bab4edcdba068b35a6c0b8774192e59eb.

to switch those features on. This reflects the fact that advanced analysis is not free, but usually comes at the price of precision and scalability.

Using these analysers, we built call graphs using a mid-precision, context-insensitive variable type analysis. Given the simplicity of our examples, where each method has at most one call site, we did not expect that context sensitivity would have made a difference. To the contrary, a context-sensitive analysis computes a smaller call graph, and would therefore have reduced the recall of the tool further. On the other hand, a less precise method like CHA could have led to a misleading higher recall caused by the accidental coverage of target methods as FPs.

For *wala*, we used the 0-CFA call graph builder. By default, we set `com.ibm.wala.ipa.callgraph.AnalysisOptions.ReflectionOptions` to `NONE`, in the advanced configuration used, it was set to `FULL`.

For *soot*, we used spark (`"cg.spark=enabled,cg.spark=vta"`). For the advanced configuration, we also used the `"safe-forname"` and the `"safe-newinstance"` options. There is another option to support the resolution of reflective call sites, `types-for-invoke`. Enabling this option leads to an error that was reported, but at the time of writing this issue has not yet been resolved[8].

For *doop*, we used the following options: `context-insensitive`, `ignore-mainmethod`, `only-application-classes-fact-gen`. For the advanced configuration, we also enabled `reflection reflection-classic reflection-high-soundness-mode reflection-substring-analysis reflection-invent-unknownobjects reflection-refined-objects` and `reflection-specula tive-use-based-analysis`.

We did not consider any hybrid pre-analysis, such as *tamiflex* [8], this was outside the scope of this study. This will be discussed in more detail in Sect. 5.3.

The experiments were set up as follows: for each benchmark program, we used a lightweight byte code analysis to extract the oracle from the `@Target` annotations. Then we computed the call graph with the respective static analyser using the method annotated as `@Source` as entry point, and stored the result in probe format [23]. Finally, using the call graph, we computed the FPs and FNs of the static call graph with respect to the oracle, using the annotations as the ground truth. For each combination of benchmark program and static analyser, we computed a *result state* depending on the annotations found in the methods reachable from the `@Source`-annotated method in the computed call graph as defined in Table 1. For instance, the state ACC (for accurate) means that in the computed call graph, all methods annotated with `@Target(expectation=YES)` and none of the methods annotated with `@Target(expectation=NO)` are reachable from the method annotated with `@Source`. The FP and FN indicate the presence of false positive (imprecision) and false negatives (unsoundness), respectively, the FN+FP state indicates that the results of the static analysis are both unsound and imprecise. Reachable means that there is a path. This is slightly more gen-

[8] https://groups.google.com/forum/m/#!topic/soot-list/xQwsU7DlmqM, accessed 5 June 2018.

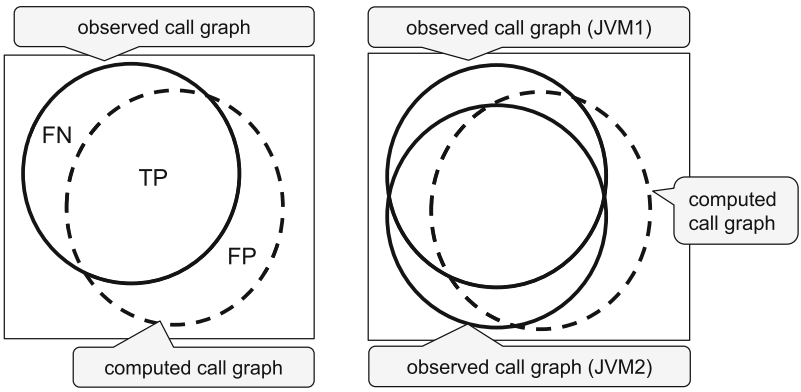
eral than looking for an edge and takes the fact into account that a particular JVM might use intermediate methods to implement a certain dynamic invocation pattern.

Table 1. Result state definitions for programs with consistent behaviour across different JVMs

Result	Methods reachable from source by annotation	
State	`@Target(expectation=YES)`	`@Target(expectation=NO)`
ACC	All	None
FP	All	Some
FN	None	None
FN+FP	None	Some

Figure 1(a) illustrates this classification. As discussed in Sect. 4.4, there are programs that use the `@Target(expectation=MAYBE)` annotation, indicating that actual program behaviour is not defined by the specification, and depends on the JVM being used. This is illustrated in Fig. 1(b).

For the programs that use the `@Target(expectation=MAYBE)` annotation, we had to modify this definition according to the semantics of the annotation: during execution, exactly one of these methods will be invoked, but it is up to the particular JVM to decide which one. We define result states as shown in Table 2. Note that the `@Target(expectation=YES)` and the `@Target(expectation=MAYBE)` annotations are never used for the same program, and there is at most one method annotated with `@Target(expectation=YES)` in a program.



(a) Consistent behaviour across JVMs (b) Inconsistent behaviour across JVMs

Fig. 1. Observed vs computed call graph

This definition is very lenient - we assess the results of a static analyser as sound (ACC or FP) if it does compute a path that links the source with *any* possible target. This means that soundness is defined with respect to the behaviour observed with only *some*, *but not all*, JVMs.

Table 2. Result state definition for programs with behaviour that depends on the JVM

Result	Methods reachable from source by annotation	
State	`@Target(expectation=MAYBE)`	`@Target(expectation=NO)`
ACC	Some	None
FP	Some	Some
FN	None	None
FN+FP	None	Some

5.2 Reproducing Results

The benchmark and the scripts used to obtain the results can be found in the following public repository: https://bitbucket.org/Li_Sui/benchmark/. Further instructions can be found in the repository `README.md` file.

5.3 Results and Discussion

Results are summarised in Table 3. As expected, none of the static analysers tested handled all features soundly. For *wala* and *doop*, there are significant differences between the plain and the advanced modes. In the advanced mode, both handle simple usage patterns of reflection well, but in some cases have to resort to over-approximation to do so. *Wala* also has support for certain usage patterns of other features: it models `invokedynamic` instructions generated by the compiler for lambdas correctly, and also models the intermediate native call in `Thread.start`. This may be a reflection of the maturity and stronger industrial focus of the tool. *Wala* also models the dynamic proxy when in advanced mode. We note however that we did not test *doop* with the new proxy-handling features that were just added very recently [15].

While *soot* does not score well, even when using the advanced mode, we note that *soot* has better integration with *tamiflex* and therefore uses a fundamentally different approach to soundly model dynamic language features. We did not include this in this study. How well a dynamic (pre-) analysis works depends a lot on the quality (coverage) of the driver, and for the micro-benchmark we have used we can construct a perfect driver. Using soot with *tamiflex* with such a driver would have yielded excellent results in terms of accuracy, but those results would not have been very meaningful.

None of the frameworks handles any of the `Unsafe` scenarios well. There is one particular program where all analysers compute the wrong call graph edge:

the target method is called on a field that is initialised as `new Target()`, but between the allocation and the invocation of `target()` the field value is swapped for an instance of another type using `Unsafe.putObject`. While this scenario appears far-fetched, we note that `Unsafe` is widely used in libraries [28], and has been exploited (see Sect. 4.2).

Table 3. Static call graph construction evaluation results, reporting the number of programs with the respective result state, format: (number obtained with basic configuration)/(number obtained with advanced configuration)

Category	Analyser	ACC	FN	FP	FN+FP
Vanilla	*soot*	1/1	0/0	0/0	0/0
	wala	1/1	0/0	0/0	0/0
	doop	1/1	0/0	0/0	0/0
Reflection	*soot*	0/1	12/11	0/0	0/0
	wala	0/4	12/3	0/5	0/0
	doop	0/0	12/8	0/4	0/0
Dynamic class loading	*soot*	0/0	1/1	0/0	0/0
	wala	0/0	1/1	0/0	0/0
	doop	0/0	1/1	0/0	0/0
Dynamic proxy	*soot*	0/0	1/1	0/0	0/0
	wala	0/1	1/0	0/0	0/0
	doop	0/0	1/1	0/0	0/0
Invokedynamic	*soot*	0/0	4/4	0/0	0/0
	wala	3/3	1/1	0/0	0/0
	doop	0/0	4/4	0/0	0/0
JNI	*soot*	1/1	1/1	0/0	0/0
	wala	1/1	1/1	0/0	0/0
	doop	0/0	2/2	0/0	0/0
Serialisation	*soot*	1/1	0/0	0/0	0/0
	wala	1/1	0/0	0/0	0/0
	doop	0/0	1/1	0/0	0/0
Unsafe	*soot*	0/0	2/2	1/1	1/1
	wala	0/0	2/2	1/1	1/1
	doop	0/0	2/2	1/1	1/1
Reflection-ambiguous	*soot*	0/0	2/2	0/0	0/0
	wala	0/0	2/0	0/2	0/0
	doop	0/0	2/1	0/1	0/0

6 Conclusion

In this paper, we have presented a micro-benchmark that describes the usage of dynamic language features in Java, and an experiment to assess how popular static analysis tools support those features. It is not surprising that in many cases the constructed call graphs miss edges, or only achieve soundness by compromising on precision.

The results indicate that it is important to distinguish between the actual features, and a usage context for those features. For instance, there is a significant difference between supporting `invokedynamic` as a general feature, and `invokedynamic` as it is used by the Java 8 compiler for lambdas. The benchmark design and the results of the experiments highlights this difference.

We do not expect that static analysis tools will support all of those features and provide a sound and precise call graph in the near future. Instead, many tools will continue to focus on particular usage patterns such as "support for reflection used in the Spring framework", which have the biggest impact on actual programs, and therefore should be prioritised. However, as discussed using examples throughout the paper, more exotic usage patterns do occur, and can be exploited, so they should not be ignored. The benchmark can provide some guidance for tool builders here.

An interesting insight coming out of this study is that notions like *actual programs behaviour* and *possible program executions* are not as clearly defined as widely thought. This is particularly surprising in the context of Java (even in programs that do not use randomness, concurrency or native methods), given the strong focus of the Java platform on writing code once, and run it anywhere with consistent program behaviour. This has implications for the very definitions of soundness and precision. We have suggested a pragmatic solution, but we feel that a wider discussion of these issues is needed.

Acknowledgement. We thank Paddy Krishnan, Francois Gauthier and Michael Eichberg for their comments.

References

1. Invokedynamic rectifier/project serializer. http://www.opal-project.de/DeveloperTools.html
2. The Java language specification. https://docs.oracle.com/javase/specs
3. SPECjvm2008 benchmark. www.spec.org/jvm2008
4. Andreasen, E.S., Møller, A., Nielsen, B.B.: Systematic approaches for increasing soundness and precision of static analyzers. In: Proceedings of SOAP 2017. ACM (2017)
5. Bacon, D.F., Sweeney, P.F.: Fast static analysis of c++ virtual function calls. In: Proceedings of the OOPSLA 1996. ACM (1996)
6. Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the OOPSLA 2006. ACM (2006)
7. Bodden, E.: Invokedynamic support in soot. In: Proceedings of the SOAP 2012. ACM (2012)

8. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the ICSE 2011. ACM (2011)
9. Dietrich, J., Jezek, K., Rasheed, S., Tahir, A., Potanin, A.: Evil pickles: DoS attacks based on object-graph engineering. In: Proceedings of the ECOOP 2017. LZI (2017)
10. Dietrich, J., Schole, H., Sui, L., Tempero, E.: XCorpus-an executable corpus of Java programs. JOT **16**(4), 1:1–24 (2017)
11. Dietrich, J., Sui, L., Rasheed, S., Tahir, A.: On the construction of soundness oracles. In: Proceedings of the SOAP 2017. ACM (2017)
12. Dolby, J., Fink, S.J., Sridharan, M.: T.J. Watson Libraries for Analysis (2015). http://wala.sourceforge.net
13. Ernst, M.D.: Static and dynamic analysis: synergy and duality. In: Proceedings of the WODA 2003 (2003)
14. Foote, B., Johnson, R.E.: Reflective facilities in Smalltalk-80. In: Proceedings of the OOPSLA 1989. ACM (1989)
15. Fourtounis, G., Kastrinis, G., Smaragdakis, Y.: Static analysis of Java dynamic proxies. In: Proceedings of the ISSTA 2018. ACM (2018)
16. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification. Java Series, Java SE 8 edn. Addison-Wesley Professional, Boston (2014)
17. Grech, N., Fourtounis, G., Francalanza, A., Smaragdakis, Y.: Heaps don't lie: countering unsoundness with heap snapshots. In: Proceedings of the OOPSLA 2017. ACM (2017)
18. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: Proceedings of the OOPSLA 1997. ACM (1997)
19. Holzinger, P., Triller, S., Bartel, A., Bodden, E.: An in-depth study of more than ten years of Java exploitation. In: Proceedings of the CCS 2016. ACM (2016)
20. Jezek, K., Dietrich, J.: Magic with dynamo-flexible cross-component linking for Java with invokedynamic. In: Proceedings of the ECOOP 2016. LZI (2016)
21. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45337-7_18
22. Landman, D., Serebrenik, A., Vinju, J.J.: Challenges for static analysis of Java reflection-literature review and empirical study. In: Proceedings of the ICSE 2017. IEEE (2017)
23. Lhoták, O.: Comparing call graphs. In: Proceedings of the PASTE 2007. ACM (2007)
24. Li, Y., Tan, T., Sui, Y., Xue, J.: Self-inferencing reflection resolution for Java. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 27–53. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_2
25. Liu, J., Li, Y., Tan, T., Xue, J.: Reflection analysis for Java: uncovering more reflective targets precisely. In: Proceedings of the ISSRE 2017. IEEE (2017)
26. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: a manifesto. CACM **58**(2), 44–46 (2015)
27. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for Java. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 139–160. Springer, Heidelberg (2005). https://doi.org/10.1007/11575467_11
28. Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M., Nystrom, N.: Use at your own risk: the Java unsafe API in the wild. In: Proceedings of the OOPSLA 2015. ACM (2015)

29. Murphy, G.C., Notkin, D., Griswold, W.G., Lan, E.S.: An empirical study of static call graph extractors. ACM TOSEM **7**(2), 158–191 (1998)
30. Reif, M., Kübler, F., Eichberg, M., Mezini, M.: Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In: Proceedings of the SOAP 2018. ACM (2018)
31. Rountev, A., Kagan, S., Gibas, M.: Evaluating the imprecision of static analysis. In: Proceedings of the PASTE 2004. ACM (2004)
32. Ryder, B.G.: Constructing the call graph of a program. IEEE TSE **3**, 216–226 (1979)
33. Shipilev, A.: JEP 280: indify string concatenation. http://openjdk.java.net/jeps/280
34. Shivers, O.: Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie Mellon University (1991)
35. Smaragdakis, Y., Balatsouras, G., Kastrinis, G., Bravenboer, M.: More sound static handling of Java reflection. In: Feng, X., Park, S. (eds.) APLAS 2015. LNCS, vol. 9458, pp. 485–503. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26529-2_26
36. Smith, B.C.: Reflection and semantics in LISP. In: Proceedings of the POPL 1984. ACM (1984)
37. Sui, L., Dietrich, J., Tahir, A.: On the use of mined stack traces to improve the soundness of statically constructed call graphs. In: Proceedings of the APSEC 2017. IEEE (2017)
38. Sundaresan, V., et al.: Practical virtual method call resolution for Java. In: Proceedings of the OOPSLA 2000. ACM (2000)
39. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: Qualitas corpus: a curated collection of Java code for empirical studies. In: Proceedings of the APSEC 2010 (2010)
40. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: Proceedings of the OOPSLA 2000. ACM (2000)