On the Recall of Static Call Graph Construction in Practice

Li Sui
l.sui@massey.ac.nz
Massey University
Palmerston North, New Zealand

Jens Dietrich
jens.dietrich@vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

Amjed Tahir
a.tahir@massey.ac.nz
Massey University
Palmerston North, New Zealand

George Fourtounis
gfour@di.uoa.gr
University of Athens
Athens, Greece

ABSTRACT

Static analyses have problems modelling dynamic language features soundly while retaining acceptable precision. The problem is well-understood in theory, but there is little evidence on how this impacts the analysis of real-world programs. We have studied this issue for call graph construction on a set of 31 real-world Java programs using an oracle of actual program behaviour recorded from executions of built-in and synthesised test cases with high coverage, have measured the recall that is being achieved by various static analysis algorithms and configurations, and investigated which language features lead to static analysis false negatives.

We report that (1) the median recall is 0.884 suggesting that standard static analyses have significant gaps with respect to the proportion of the program modelled (2) built-in tests are significantly better to expose dynamic program behaviour than synthesised tests (3) adding precision to the static analysis has little impact on recall indicating that those are separate concerns (4) state-of-the-art support for dynamic language features can significantly improve recall (the median observed is 0.935), but it comes with a hefty performance penalty, and (5) the main sources of unsoundness are not reflective method invocations, but objects allocated or accessed via native methods, and invocations initiated by the JVM, without matching call sites in the program under analysis.

These results provide some novel insights into the interaction between static and dynamic program analyses that can be used to assess the utility of static analysis results and to guide the development of future static and hybrid analyses.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Software defect analysis**; *Dynamic analysis.*

KEYWORDS

static program analysis, testing, soundness, java, call graph construction, test case generation

ACM Reference Format:
Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380441

1 INTRODUCTION

Static program analysis is widely used to detect faults (such as bugs and vulnerabilities) in software early in its life cycle, when corrective action is still relatively inexpensive. Static analysis constructs and reasons about a model of program behaviour. It is highly desirable that this process (1) does not miss faults, i.e., it produces *no false negatives* and is therefore *sound* and (2) that it does not produce false alerts, i.e., it produces *no false positives* and is therefore *precise*. Unfortunately, Rice's theorem states that such a perfect analysis is generally not possible [53], so in practice, analyses aim for reasonable trade-offs between soundness and precision, and in addition to this, performance.

Unsoundness is usually attributed to the ubiquitous presence of dynamic programming language features in modern programming languages, used to implement generic components that can adapt to and be reused in different contexts. For instance, in call graph construction, the static analysis computes a directed graph with functions (methods, procedures) being represented by vertices, and invocations being represented by edges. With such a model, questions relevant for program security such as "can a function performing some I/O operation be reached from a program entry point?" or questions related to maintenance such as "what is the impact of renaming or otherwise refactoring a method?" can be answered. However, if reflection is used where a reference to a target method is computed at runtime and the language contains a feature to invoke function references (such as Smalltalk's `perform`, JavaScript's `eval` or Java's `invoke`), then it becomes difficult for a static analysis to accurately model those invocations. The emphasis is on *accuracy*, as false negatives can always be avoided by sacrificing precision. For instance, a static analysis could just resolve a dynamic call site to all possible methods, therefore avoiding false negatives by accepting (a large number of) false positives.

Problems with soundness have attracted more attention recently after the publication of the Soundiness Manifesto [42]. Most of the research on the topic is on improving static analysis to model a particular dynamic feature such as reflection [41, 56], dynamic proxies [21], or `invokedynamic` [11, 22], or by providing a dynamic pre-analysis to capture additional program behaviour that is then added into the static analysis [12, 26, 60].

In this paper, we measure the impact dynamic language features have on the soundness of statically constructed call graphs. In particular, we set out to answer the question: "How unsound is static call graph construction in practice", using call graph construction as a case study as this is a foundational analysis underpinning many higher-level static analyses. Since the term *sound* is usually considered as binary (i.e., something is either unsound or sound), we use the term *recall* instead: we measure the recall of static call graph construction as the percentage of known methods invoked present in the statically constructed call graph. Unfortunately, recall cannot be measured exactly as this would require prior knowledge of all possible program behaviours. However, we can approximate *possible program behaviour* with *known program behaviour*. We refer to this as the *oracle*, and measure and report recall with respect to an oracle of such known behaviour. More specifically, we investigate the following research questions:

RQ1 What is the the recall of static call graph construction with respect to an oracle of actual program behaviour ?
RQ2 What is the impact of context sensitivity on recall ?
RQ3 How effective is state-of-the-art reflection support ?
RQ4 Which particular language features cause static analysis false negatives ?

We study this problem for Java programs, as Java is one of the most popular programming languages, and it also is the focus of much of the existing work in static analysis including recent work to model its dynamic language features. The static analysis framework used is *doop* [13] as it provides implementations of many standard call graph construction algorithms; and offers state-of-the-art support for several dynamic language features [21, 22, 56].

The results presented here can guide static analysis users to better understand what (not) to expect from static analysis, and how to interpret and use its results. The results are also useful for static analysis tool builders to guide them where to best focus efforts to improve the recall for their analysis. Furthermore, it provides some insights into the potential of hybrid analyses.

2 RELATED WORK

2.1 Call Graph Construction

There is a large body of research into call graph construction, algorithms mainly differ in achieving different trade-offs between precision and runtime. Tip and Palsberg [63] conducted an earlier study comparing the main approaches. A main problem in call graph construction for Java and related languages is how they deal with virtual method invocations that are only resolved at runtime. Class Hierarchy Analysis (CHA) [30] is a lower-precision fast algorithm that uses class hierarchy information to resolve virtual methods. Rapid Type Analysis (RTA) [9] also takes allocation sites into account. 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 [61]. k-CFA analyses [55] add various levels of call site sensitivity to the analysis in order to further improve precision.

2.2 Handling of Dynamic Language Features

Reflection is a dynamic language feature in Java and similar languages, first introduced in LISP and Smalltalk [20, 58]. It's impact on static analysis has been comprehensively studied. Livshits et al. [43] used points-to analysis to approximate the targets of reflective call sites as part of call graph construction. Li et al. [38] proposed *elf* in order to improve the effectiveness of Java pointer analysis tools. Smaragdakis et al. [56] further refined this approach in order to improve both recall and performance. *Wala* [19] has built-in support for certain reflective features such as `Class::forName`, `Class::newInstance`, and `Method::invoke`.

The `invokedynamic` instruction is another dynamic language feature aimed at providing developers with more control over method dispatch, and mainly used to compile lambdas. Bodden [11] proposed a *soot* extension that supports reading and rewriting `invokedynamic` byte codes. The *opal* static analyser also provides support for `invokedynamic` through replacing `invokedynamic` instructions which use the Java `LambdaMetaFactory` with a standard `invokestatic` instruction [1]. *Wala* provides support for `invokedynamic` generated for Java 8 lambdas[1]. Like the different approaches to handle reflection, support for `invokedynamic` often does not address the language feature as such, but only particular usage patterns. In particular, the above-mentioned approaches assume that a certain bootstrap method is used. This works well as long as this is how the respective feature is used in real-world Java programs (e.g., making assumptions about the byte code emitted by the current Java compiler), but fails if byte code produced by non-Java or non standard Java compilers is analysed [59]. This is a relevant problem as the JVM has become a polyglot platform. Support for `invokedynamic` has been very recently added to *doop* [22], however, this was not yet available when the experiments presented here were conducted.

Fourtounis et al. [21] have recently proposed the first analysis for *dynamic proxies*, based on the *doop* framework. This is one of the features we have used to evaluate the level of recall that can be achieved by a static analysis with support for dynamic language features.

2.3 Hybrid Analyses

Hybrid analyses aim at combining static and dynamic techniques in order to offset their respective weaknesses in terms of soundness and precision [18]. In particular, a dynamic pre-analysis can be used to record executions, and then the information recorded can be fed into a static analysis, for instance, by modifying the original code (e.g., "unreflecting"), or by creating a spec that can be used directly to augment a static analysis model (e.g., by generating additional facts for datalog-based static analysers). *Tamiflex* by Bodden et al. [12] is such an approach where a Java program is instrumented, and method invocations are recorded. The original code is then enriched with "unreflected" code, this approach has

[1] https://goo.gl/1LxbSd and https://goo.gl/qYeVTd, both accessed 14 Jan 19

the advantage of being tool-agnostic. Grech et al. [26] proposed *heapdl*. This tool is conceptually similar to *tamiflex* but also uses heap snapshots to further improve recall. *Mirror* by Liu et al. [41] is another hybrid analysis aimed at resolving reflective call sites. Sui et al. [60] proposed a rather different approach to extract additional edges from stack traces reported in online error reports. While the number of invocations found with this method is small, some of them were indeed missed by static analysis tools, and the authors argue that those are "high-value" edges as they were involved in bugs or vulnerabilities.

In general, all of these methods can boost recall but are not suitable to make an analysis sound as they only capture a fraction of actual program behaviour. Dietrich et al. [17] generalised this idea and discussed several general approaches of obtaining *soundness oracles* that can be used to assess and improve the recall of static analysis.

2.4 Empirical Studies

Murphy et al. [46] compared the results of 9 static analysis tools applied to three C sample programs. They found that the extracted call graphs varied in size, which makes them potentially unreliable for developers to use. Lhoták [35] proposed an infrastructure to facilitate the assessment and comparison of static analysis tools, this includes a call graph interchange format.

Landman et al. [33] studied the challenges faced by static analysers to model reflection in Java. They found that the parts of the reflection API that prove problematic for static analysers are widely used. They used a lightweight static analyses based on detecting patterns in the abstract syntax tree of programs for their analysis. They opted for a sound under-approximation in their pattern detection, similar to our oracle construction. The main difference to our approach is that they are not able to quantify the impact reflection has on the actual static analysis models, as they are not computing an oracle of actual program behaviour. Our study also aims at including other source of unsoundness, not just reflection.

Mastrangelo et al. [45] looked into how one of the more exotic dynamic language features we also study, the low-level `sun.misc.Unsafe` API, is used in practice.

More recently, two conceptually similar micro-benchmarks were proposed to describe the impact of dynamic language features on call graph construction by Reif et al. [52] and Sui et al. [59]. The synthetic benchmark programs in both studies are minimalistic by design, and facilitate the construction of an oracle of actual program behaviour by hand. The studies then analyse how common static analysers perform in analysing the respective programs. An interesting feature of the Sui study is that it identifies that the notion of actual program behaviour needs to take the JVM used into account, as the resolution of reflective call sites is not completely determined by the the rules in the JVM specification, and different mainstream JVMs interpret them differently. We discuss the impact that this has on our methodology briefly in Section 3.5.5. Pontes et al. [49] have recently systematically studied cases of under-determined specifications and non-conformances in the reflection API implemented by different JVMs.

Judge [51] builds upon [52], and also contains a case-study experiment on *xalan* in order to assess the recall of the static call graphs constructed by several static analysis tools. This is close to what we set out to achieve in this paper, however, we work with a larger data set in order to discover generalisable patterns, and use a different highly automated method to construct the oracle and categorise false negatives.

Karim and Lhoták studied the construction of call graphs for the application part of programs [3]. They used a methodology similar to ours to assess the soundness of the statically constructed call graphs against recorded program executions. They used a smaller data set comprising programs from dacapo 2006 [10] and SPEC JVM 98 [2]. We systematically study soundness for whole program analysis, study the impact of reflection analysis that was not available when the Karim and Lhoták study was conducted, using a much richer model of program behaviour, and classify sources of unsoundness.

3 METHODOLOGY

In this section, we discuss the methodology used in our study. An overview of the process is given in Figure 1. This study is based on the comparison of two models – a static model computed by means of a static analysis (the static call graph – SCG), and a dynamic model (the context call tree - CCT), constructed by observing an executing program.

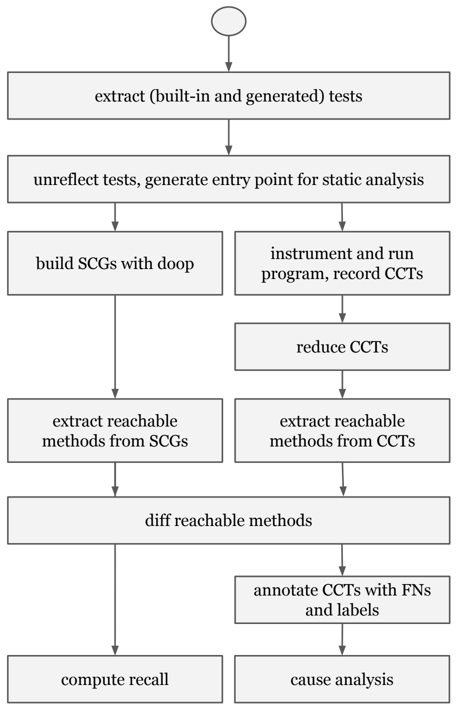
extract (built-in and generated) tests
unreflect tests, generate entry point for static analysis
build SCGs with doop
instrument and run program, record CCTs
reduce CCTs
extract reachable methods from SCGs
extract reachable methods from CCTs
diff reachable methods
annotate CCTs with FNs and labels
compute recall
cause analysis

Figure 1: Study setup overview (SCG - static call graph, CCT - context call tree, FN - false negative)

3.1 Oracle Construction

Our study hinges on the notion of a *soundness oracle* (oracle for short) – a known ground truth of actual program behaviour to which the statically computed call graph can be compared to. As the focus of our study is on call graphs, a soundness oracle consists of vertices in a call graph representing methods. Methods are identified and modelled using a combination of the name of the class defining the method, the method name and a descriptor as defined in the Java language specification [40, Sect. 4.3.3] to model overloading. Call graph edges consist of pairs of (source and sink) methods.

In order to build oracles of actual program behaviour to compare static analysis models with, programs need to be exercised in a way that exposes as much possible program behaviour as possible. Then an oracle can be generated by monitoring the executing program, and recording method invocations by means of instrumentation. Unfortunately, many (Java) programs do not have `main` methods as they are intended to be used as libraries, or be executed in containers such as application servers. Even if a program had a `main` method, it would often not be clear how to supply arguments to obtain meaningful (e.g., high-coverage) program runs. On the other hand, unit tests are widely used, and provide suitable entry points to trigger program executions. There are two approaches to source tests: either to use tests *built-into* the program to be analysed, or to generate (synthesise) tests by a tool. Test case generation [8, 15, 48] is a viable option as it can achieve high coverage in particular if it is feedback-directed, and its main limitation (to create meaningful assertions) is irrelevant if we just want to observe program behaviour, without checking the program for correctness.

We argue that built-in test cases still have a special quality to them as they represent *intended* program behaviour. Generated tests still represent *potential* program behaviour that is valuable to consider (e.g., in vulnerability detection with fuzzing), but there is another quality in that intended behaviour better reflects the behaviour end users will encounter when the software is deployed, and we were therefore interested to study this separately. However, by also being able to use generated tests in addition to built-in tests, we obtain a richer model of program behaviour, reflected in increased coverage. The programs in the data set we used have both built-in and generated tests, see Section 3.2 for details. Figure 2 depicts the branch coverage obtained for the programs investigated using built-in, generated and combined tests. In general, better coverage is achieved with generated tests. But this figure also indicates that when combining generated and built-in tests, coverage significantly increases, indicating a certain level of orthogonality between those tests. This is somehow surprising – both seem to exercise different parts of the programs under test.

We make the assumption here that high (branch) coverage is correlated with more methods being invoked while the program executes (i.e., higher method coverage). This assumption is based on the observation that (1) with higher coverage more invocation instructions become reachable and (2) as more allocation sites (object creation sites) become reachable, more methods become reachable when virtual invocations are resolved.

3.2 Dataset Acquisition

We have used the *xcorpus* data set [16] for our experiments for the following reasons: (1) it is based on the widely used qualitas corpus [62] that consists of a large curated (to be representative) set of real-world Java programs (2) it has a built-in driver with high coverage for each program, obtained by using the *evosuite* test case generation [23] (3) programs in the *xcorpus* use several dynamic language features, the respective analysis can be found in [16], and finally (4) the *xcorpus* has been used recently in related work by different authors [21, 51, 59]. Of the 75 programs in the *xcorpus* , we study the following 31 programs that have at least one built-in *junit* test: *castor-1.3.1, checkstyle-5.1, commons-collections-3.2.1, drools-7.0.0.Beta6, findbugs-1.3.9, fitjava-1.1, guava-21.0, htmlunit-2.8, informa-0.7.0-alpha2, javacc-5.0, jena-2.6.3, jFin_DateMath-R1.0.1, jfreechart-1.0.13, jgrapht-0.8.1, ApacheJMeter_core-3.1, jrat-0.6, jrefactory-2.9.19, log4j-1.2.16, lucene-4.3.0, mockito-core-2.7.17, nekohtml-1.9.14, openjms-0.7.7-beta-1, oscache-2.4.1, pmd-4.2.5, quartz-1.8.3, tomcat-7.0.2, trove-2.1.0, velocity-1.6.4, wct-1.5.2, weka-3-7-9, mockito-core-2.7.17.*

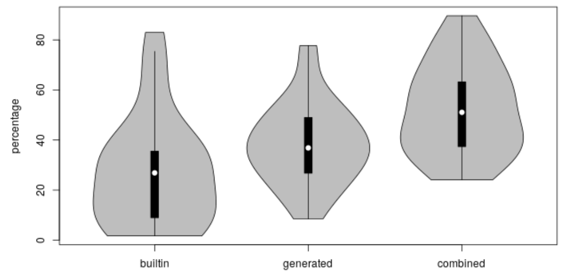

Figure 2: Branch coverage obtained by executing built-in, generated and combined tests in %, for the data set of 31 programs described in Section 3.2

3.3 Static Entry Point Generation

Usually, *junit* test cases are executed by employing a *junit runner*. *Junit* will then detect tests and fixtures based on the presence of annotations (*junit4*) or naming patterns (*junit3*), and invoke the respective methods.

The fact that *junit* uses reflection to invoke tests poses a significant problem for static analyses – this is the very problem we are investigating. We therefore created a light-weight pre-analysis and code generator in order to *unreflect* the *junit* test cases and create static drivers. Listings 1 and 2 illustrate the process. The unreflected code for each test case runs in its own exception handler to ensure that test cases resulting in exceptions or catchable errors did not prevent the execution of subsequent tests. This would fail if tests resulted in uncatchable throwables, such as out of memory errors, preventing following unreflected tests to execute. We monitored for those cases by logging, and found that those cases are very rare.

```
import org.junit.*;
public class Test42 {
  private T tested = null;
  @BeforeClass public void beforeClass() {};
  @Before public void setUp() {tested = new T();}
  @Test public void test() {tested.foo();}
  @After public void tearDown() {tested = null;}
  @AfterClass public void afterClass() {};
}
```

Listing 1: Original JUnit test case

```
public class Driver_Test42 {
  public static void main(String[] args) throws Exception {
    Test42 test = new Test42();
    test.beforeClass();
    test.setup();
    test.test();
    test.tearDown();
    test.afterClass();
  }
}
```

Listing 2: Unreflected JUnit test case (simplified)

JUnit has some features that are difficult to capture by unreflection, such as custom rules and runner. Our unreflection technique supports the following *junit* features (package names omitted), supporting both *junit3* and *junit4* conventions.

(1) test methods annotated with `@Test`
(2) non-static fixtures annotated with `@Before` or `@After`
(3) static fixtures annotated with `@BeforeClass` or `@AfterClass`
(4) tests annotated with `@RunWith(Parameterized.class)`
(5) test methods in subclasses of `junit.framework.TestCase` complying to *junit3* test method conventions
(6) *junit3* fixtures, i.e. `setUp()` and `tearDown()` implemented in subclasses of `junit.framework.TestCase`

We removed *junit* and *evosuite* framework classes from the analysis scope as their presence would have added additional vertices and edges to the oracle (not just the methods defined in the respective frameworks, but also (standard) library function invoked through those), and therefore would have biased the recall results. This required the mechanised removal of some call sites from tests, in particular invocations of junit `assert*` methods. The coverage data shown in Figure 2 was obtained by invoking the unreflected tests with the generated driver, measured with *jacoco* [2].

In addition to the drivers we set up for each test class, we also generated a global driver class with a single `main` method calling all generated test `main` methods, to be used as a single static analysis entry point.

3.4 Static Call Graph (SCG) Construction

The static model is the static call graph (SCG), a directed graph (V, E) consisting of a set of vertices V and a set of edges $E \subseteq V \times V$. Vertices represent methods, while edges represent invocation relationships. For our study, we used the *doop* framework with different configurations to construct the call graph. *doop* implements a wide range of algorithms including support for context sensitivity, and several dynamic language features, this support is comparable to or exceeds similar features available in alternative frameworks such as *soot* [32] and *wala* [19] as demonstrated in recent benchmark-based comparative studies [52, 59]. The *doop* version used was *4.14.4*. We used the following options for the respective analyses:

- base analysis: `-a context-insensitive`
- context-sensitive analysis: `-a 1-call-site-sensitive`
- reflection analysis: `-a context-insensitive -reflection --reflection-classic --reflection-dynamic-proxies --reflection-method-handles --simulate-native-returns`

In all cases, the `-main` option was used with the generated entry point as argument.

[2]https://www.eclemma.org/jacoco/

3.4.1 Bytecode Acquisition.

In order to run the static analysis we needed the byte code of the program and the library the program depends on. This required us to first resolve the symbol references to dependencies in the *xcorpus* programs. For each program, we used the `ivy resolve` task to fetch dependencies from Maven and in some cases (project-) local repository, and made local copies of these libraries available for the static analysis.

3.4.2 Library Dependencies.

A crucial decision to be made when setting up the static analysis is the handling of libraries. We run the static analyses in two modes:

(1) *superjar mode*: To make all library classes part of the program to be analysed, this was done by building a single "super" jar containing all library classes.
(2) *library mode*: Library code is handled differently by only representing the parts of the library used by the program. This is supported by *doop* , but introduces some additional unsoundness. The main reason is that *doop* relies on the facts *soot* [32] generates from (library) code, and if library code is only accessed through reflection or similar means, those facts set will be incomplete. Even if *doop* is used with reflection support, the analyses may still fail to generate some call graph edges.

Handling libraries and the main program differently is a widely used technique in static program analysis [4, 5, 50]. By investigating both settings, we are in a position to measure the impact this has on the analysis recall.

3.5 Context Call Tree (CCT) Construction

The dynamic model used in order to provide the oracle is the calling context tree (CCT) [6]. A vertex in the CCT is an invocation, i.e. a pair consisting of a method and a unique generated id. The root of this tree is an entry point (such as "`main`"). In order to construct the CCTs, stacks were monitored by means of instrumentation with ASM [14]. On method entry and exit, stack push and pop instructions were logged, this information was then used to build the CCTs once the method executions had ended. For each entry point and thread, a separate CCT was created.

Using CCTs enabled us to precisely model method invocations. In particular, we were interested in whether certain methods flagged as false negatives would remain reachable from a program entry point once reflective methods were removed in order to measure the impact these reflective methods have on recall. This kind of *cause analysis* would not have been possible had we used a coarse dynamic call graph (DCG) representation that allows spurious paths.

3.5.1 Reducing CCTs.

The downside of using CCTs however is that they quickly grow very large. Even fairly simple programs quickly create CCTs of several GBs in size. We investigated several techniques to reduce the size of the CCTs, and implemented a simple loop reduction. When methods are invoked in loops, a new branch is created for each iteration. Often, those branches are isomorphic and therefore redundant: for each branch, the same methods are invoked in the same order [3]. We removed redundant branches caused by

[3]More precisely, we can define two branches with roots $v_1 = (method_1, id_1)$ and $v_2 = (method_2, id_2)$ as isomorphic as follows: if the vertices don't have successors,

loops [4]. In most cases, this reduced the size of CCTs dramatically by order of a magnitude. Removing redundant branches does not affect the cause analysis as the reachability of methods is not affected.

We also encountered branches spawned by our instrumentation, with invocations of `sun.instrument.InstrumentationImpl::transform` as root. Those branches were removed to ensure that the experimental setup did not bias the results.

3.5.2 Threads. Threads were modelled by building a separate CCT for each thread. The code inserted via instrumentation generates thread identifiers using a combination of thread id and identity hash code of the current thread. Those identifiers and the thread names were attached as meta data to the CCTs, this information was then used later in the cause analysis (Section 3.7) in order to identify system threads by name.

3.5.3 Exceptions. The semantics of exception handling [40, sect 2.10] was modelled by instrumenting exception handlers to log the removal of methods from the stack when an exception is propagated. We did not track unhandled exceptions. While this can be done by instrumenting `uncaughtException` methods in classes implementing the `Thread.UncaughtExceptionHandler`, this was not necessary as the JVM specification states that "If no suitable exception handler is found before the top of the method invocation chain is reached, the execution of the thread in which the exception was thrown is terminated" [40, sect 2.10]. I.e., there is no further manipulation of the respective stack that would lead to the recording of additional invocations.

3.5.4 Native Methods. Native methods cannot be directly instrumented. We therefore run a simple pre-analysis to identify call sites that resolve to native methods, and instrument the respective call sites (e.g. `invoke*` instructions). In rare cases where a virtual method can be resolved to both native and non-native methods, this approach could fail to record some edges, resulting in a smaller oracle.

3.5.5 Limitations. There is an interesting corner case where the resolution of a reflective call site depends on the JVM used due to the non-deterministic nature of the order of method annotations queried via the reflection API [59]. Apparently, this feature is used by at least one program in the data set we are using, *log4j*. We found that the order of annotations is also changed by instrumentation of the respective methods. We consider this a *Heisenbug* and think that it will have no impact on our results as we do not expect that any static analysis can correctly model the (procedural) logic of resolving a reflective call site to the *first* method m annotated with annotation a in class c. We expect a static analysis will either find none of the targets (unsound), or all of them (imprecise), so which particular one is selected when the oracle is built is irrelevant.

they are isomorphic iff $method_1 = method_2$. Otherwise, they are isomorphic iff $method_1 = method_2$ and the ordered lists of children are element-wise isomorphic.

[4]We used the following simple algorithm: we traversed the tree to compute structural hashes from the invoked methods and the hashes of the successors for all vertices, and then looked for siblings with identical hashes. For those candidate roots of isomorphic branches, we did a recursive structural comparison to avoid hash collision.

3.6 Measuring Reachability and Recall

In order to answer RQs 1-3 we had to measure the recall of the static analysis with respect to an oracle. For both the SCG and a set of CCTs (corresponding to different threads encountered during program execution) we can easily extract sets of reachable methods with a single traversal: for the SCG, this is just the set vertices, and for the CCTs, this is the set of methods that occur in any invocation in any of the CCTs. For a given program, let M_{SCG} and M_{CCT} be those sets. Then we can define *recall* as follows:

$$recall := \frac{|M_{CCT} \cap M_{SCG}|}{|M_{CCT}|}$$

A sound analysis has a recall value of 1, whereas the presence of false negatives, i.e. methods that are observed when the program executes, but not computed as reachable by the static analysis, lowers the recall value. It is important here to remember that the recall measured here is relative to the oracle that is itself unsound as it does not reflect all *possible* program behaviour. We believe that this is the only possible method to approach the problem as it is practically impossible to construct a driver that triggers all possible program behaviours except for trivial micro-benchmarks [5].

Note that our approach uses reachable methods and is therefore vertex-based, not edge-based. We opted for this approach for the following reasons:

(1) We have encountered method invocations by the JVM, i.e., without visible call sites (see also discussion in Section 3.7). A vertex-based approach allowed us to capture those methods as sources of unsoundness.
(2) The recall measured with a vertex-based approach can be higher than the recall that would have been obtained with an edge-based approach. This makes our measurements conservative. The overall message of our paper is that the recall observed in practice is relatively low, and this observation would therefore remain valid even if we switched to an edge-based approach.
(3) A vertex-based approach to study call graphs is widely used, examples include [36], [3] and [51].
(4) There are several analysis clients relying on a vertex-based call graph reachability analysis, including dead code elimination [31] and static regression test selection [54].

3.7 Cause Analysis and CCT Tagging

In order to answer RQ4 we had to analyse which particular language features were used to spawn branches within a CCT that contain invocations of methods which were not reachable in the statically constructed call graph(s). For some features, in particular method invocations through reflection, this is straightforward: remove vertices corresponding to invocations of `Method::invoke` from the CCT, and count the vertices marked as false negatives (with respect to a static analysis) that are only reachable via (i.e., dominated by) those vertices. This approach is illustrated in Figure 3. It provides a measure of the impact the presence of `Method::invoke` has on the recall of the analysis. We refer to dynamic features that can be detected through the presence of certain methods in the CCT as

[5]This is the approach taken in [52, 59], but this methods is not suitable to quantify the impact of the issue on *real-world* input.

dynamic invocations (DI). The basic idea is to *tag* dynamic invocations with a label corresponding to the language feature (such as `#Method.invoke`), and then measure the (percentage of) invocations corresponding to false negatives in the CCT dominated [34] by the tagged vertices.

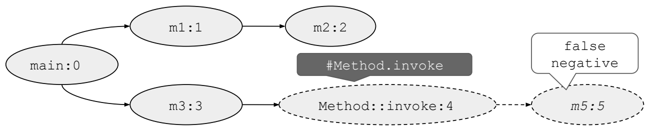

Figure 3: CCT cause analysis for dynamic invocations, the presence of `Method::invoke` can explain why `m5` is not reachable in the SCG. Invocations are represented using the following syntax <class-name>::<method-name>:<invocation-id>

Another common dynamic invocation pattern occurs when lambdas are compiled and the `invokedynamic` instruction is used. We tracked those invocations by taking advantage of naming patterns used by the OpenJDK compiler [25].

While we initially expected that dynamic invocations would explain most analysis false negatives, this was not the case. It turns out that *dynamic allocations (DALL)* also have a significant impact. An example is the use of `Class::newInstance`. This dynamically creates an object `obj` of some type `T`, and the static analysis has to track method invocations `v.foo()` with `v` pointing to `obj`. If the object was not correctly tracked, then devirtualisation would not be modelled correctly, and the analysis result may contain false negatives.

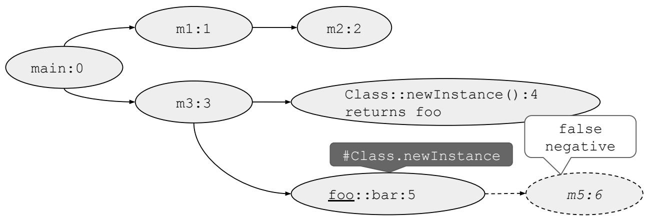

Figure 4: CCT cause analysis for dynamic allocations, the use of `Class::newInstance` can explain why `m5` is not reachable in the SCG.

```
void m3(String clsName) throws Exception {
    T foo = Class.forName(clsName).newInstance();
    foo.bar();
}
```

Listing 3: Invocation of a method with a dynamically allocated object

Tracking those allocations required additional instrumentation to enrich the CCT with additional information in form of vertex labels. Figure 4 illustrates our approach, using the code snippet from Listing 3. Objects returned by dynamic allocation sites were tracked, and invocations where `this` at the call site pointed to such an object were tagged, with `#Class.newinstance` in this case.

This tagging can be considered as a form of lightweight dynamic taint analysis [47], where objects are considered tainted when they are dynamically allocated. Note that we tracked the last dynamically allocated object we encountered. In particular, this matters when considering that *Class::newInstance* uses *Constructor::newInstance*. If an object has been created by *Class::newInstance*, it was therefore already marked as being created by *Constructor::newInstance*, however, this annotation was then overridden. Therefore, when we tagged an invocation with `#Constructor.newinstance`, this means that the application had created the object by invoking `Constructor::newInstance` directly, not indirectly via the intermediate `Class::newInstance`.

A situation similar to dynamic allocation arises when an object is accessed via reflection or similar means, such as a reflective heap access via `Field::get`. We refer to this pattern as *dynamic access (DACC)*, and model it like dynamic allocation by tracking objects returned by the invocations of these methods. We also tracked objects returned by native methods. They are also included in the *DALL* category, however, we did not track whether the objects returned were actually newly allocated objects, or already known objects. So there could be some cases of dynamic access in this category.

Table 1: Dynamic invocation, allocation and access patterns used for tagging, (..) means any set of parameter, _ is short for "java.lang"

invocation pattern	tag	category
`_.reflect.Method::invoke`	`#method.invoke`	DI
`*::lambda$*`	`#lambda`	DI
`_.reflect.InvocationHandler::invoke`	`#dynproxy.invoke`	DI
`_.invoke.MethodHandle::invoke*`	`#handler.invoke`	DI
`_.Class::newInstance`	`#class.newinstance`	DALL
`_.reflect.Constructor::newInstance`	`#constructor.newinstance`	DALL
`java.io.ObjectInputStream::readObject`	`#deserialize`	DALL
`sun.misc.Unsafe::getObject`	`#unsafe.getobject`	DALL
objects returned by native methods	`#nativeallocation`	DALL
`_.reflect.Field::get`	`#field.get`	DACC
invocations without call sites in program	`#nocallsite`	SYS
roots of system threads	`#systemthread`	SYS

The next pattern we have encountered are false negatives caused by invocations without matching call sites in the program. There are methods that are only invoked by the JVM, in particular life cycle-related methods such as `ClassLoader::loadClass`. For each invocation, we checked whether there was a call site for this method in the parent method (i.e., the method of the parent vertex in the CCT) and if this was not the case, we tagged the method with `#nocallsite`. Closely related to this are methods that were called from system threads calling back into application code. Examples are invocations of `Object::finalize` and user interface event handlers. There are a number of system threads that can be recognised by name, and we used a special tag `#systemthread` to tag the roots of the CCTs generated for these threads. We tracked the following threads: `Signal Dispatcher`, `AWT-EventQueue-0`, `Reference Handler`, `AWT-Shutdown`, `Finalizer` and `DestroyJavaVM`. Note that the naming of these system threads depends on the particular JVM implementation used in the experiments as it is not defined by the JVM specification. We categorise the invocations tagged with either `#nocallsite` or `#systemthread` as *system (SYS)*. Table 1 lists the tagged invocation patterns and their respective categories.

Note that there might be multiple possible causes that a method is not reachable in the SCG. If an invocation corresponds to a static analysis false negative, there might be multiple tagged invocations on the path connecting it to the root, offering multiple explanations of why the respective method in unreachable. In fact, this does not necessarily indicate that this classification yields false positives as there might actually be multiple root causes that prevent the static analysis from computing a method as reachable.

4 RESULTS

4.1 Overview

We conducted experiments to measure the recall of various static call graph construction techniques with respect to different oracles. This led to a combinatorial explosion in the number of possible experiments: there are three types of static analyses (context-insensitive, context-insensitive with reflection support and context-sensitive), three possible oracles (constructed from built-in, generated and combined test cases), and the additional parameter whether to run the analysis in library or whole program (super jar) mode. This implies that 18 computationally expensive experiments had to be conducted and reported for each of the 31 programs, making both the execution and reporting challenging. To deal with this, we prioritised experiments and proceeded as follows. We first measured the recall of the baseline context-insensitive (base) analysis with respect to the oracles provided by built-in, generated and combined test cases, for both the library and the superjar configuration, the results answer RQ1 and are reported in Section 4.2. The impacts of context sensitivity (RQ2) and reflection support (RQ3) were then assessed and are reported in Sections 4.3 and 4.4. False negatives are further investigated in detail in Section 4.5 in order to answer RQ4. For RQs 2-4, we restricted the experiments to always use the combined set of (generated and built-in) tests, and the library analysis mode.

We report the run times of the respective experiments in Table 2. While the analysis of performance was not one of our research questions, performance did have an impact on our methodology, and matters as it is part of the trade-off that is being made when choosing a static analysis. Note the high cost of running the instrumented tests (not taking into account the already very high cost of generating tests, reported in [16]), and of running the static analysis with reflection support, with only 20 experiments avoiding time outs (set to 6 hours) [6] [7]. For all experiments, Java 1.8.0_144-b01 (Java HotSpot(TM) 64-Bit Server VM, build 25.144-b01, mixed mode), running on a Ubuntu OS was used. The heap size of the JVM was set to 16GB for the CCT recording, 256GB for the CCT reduction and 384GB for the static analyses.

[6]For the following programs, the analysis with reflection support did not time out: *checkstyle-5.1, commons-collections-3.2.1, informa-0.7.0-alpha2, findbugs-1.3.9, fitjava-1.1, javacc-5.0, jena-2.6.3, jFin_DateMath-R1.0.1, jfreechart-1.0.13, jgrapht-0.8.1, jrat-0.6, jrefactory-2.9.19, marauroa-3.8.1, nekohtml-1.9.14, openjms-0.7.7-beta-1, oscache-2.4.1, pmd-4.2.5, quartz-1.8.3, trove-2.1.0, velocity-1.6.4*

[7]The timeout of 6 hours chosen is at the upper end of the time outs used in related work: [36, 51] – 90 mins, [37, 39, 57] – 3 h, [21, 27] – 4 h, [28] – 6 h, [29] – 7 h.

Table 2: Experiment run times

analysis	programs analysed	median	max
CCT recording	31	5h 5mins	76h 4mins
CCT processing	31	12h	144h
SCG construction (base, lib)	31	3mins	31mins
SCG construction (base, super)	31	3mins	39mins
SCG construction (ctx-sens, lib)	31	8mins	3h 54mins
SCG construction (refl, lib)	20	1h 36mins	5h 42mins

4.2 The Recall of Static Program Analysis

To answer RQ1 we measured the recall of the base static analysis. The recall values for the context-insensitive baseline analysis are depicted in Figure 5. While in general the recall values (combined tests, the static analysis uses the lib setup) were high with a median of 0.884 , the "unsoundness" gaps were still significant, indicating that the static analysis typically misses around 11% of the known reachable methods. We also computed the recall with respect to the oracles obtained by the built-in and generated tests separately. The recall with respect to the oracle obtained with built-in tests was significantly lower (median 0.859) than the recall obtained using the generated test oracle (median 0.904). This suggest an interesting characteristic of built-in tests – they are better in penetrating code that uses dynamic language features than the generated tests. Note that this result was obtained with tests generated with one particular test generation framework – *evosuite* . The likely explanation is that test case generators (at least *evosuite*) have to deal with similar problems as static analysis to reason about dynamic language features as intended by the programmer.

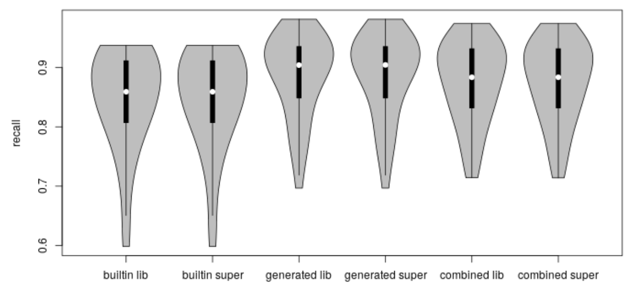

Figure 5: Recall of the base static analysis with respect to oracles obtained by executing built-in, generated and combined tests, for both lib and superjar mode

Figure 5 also indicates that there is no significant difference between the library and the superjar analysis mode. This indicates that dynamic language features are not used at component boundaries. We note however that there are programming patterns that do exactly this, but none of the program in the data set uses them. The use of a plugin-like model used in JDBC 4 with service locators is such a model [7, Section 9.2.1].

We also investigated whether the false negatives are methods declared in core Java (methods are declared in classes within `java.*` packages), extended Java (other official packages that are part of the Java Runtime Library, such as `javax.*`, `org.omg.*`), Java private (`sun.*`, `com.sun.*`, `com.oracle.net`) or application-defined (everything else, including application and third-party library packages). The average percentages of false negatives in the respective categories are as follows: 22.25% Java core, 10.51% Java extended, 46.50% Java private and 21.47% application. The high number of methods defined in Java-private classes stands out, this is consistent with the results of the cause analysis discussed in Section 4.5. The number of methods in application classes missed by the static analysis is still significant.

4.3 The Impact of Context-Sensitivity

In order to answer RQ2 we measured the recalls with respect to the oracle created by executing all tests for both the base (context-insensitive) analysis and a context-sensitive analysis as described in Section 3.4. The results are depicted in the second column of Figure 6, the median recall is 0.880 . It turns out that gaining precision (i.e., eliminating false positives) has very little impact on the recall. I.e. there are very few false negatives that were covered by the false positives of the less precise context-insensitive analysis.

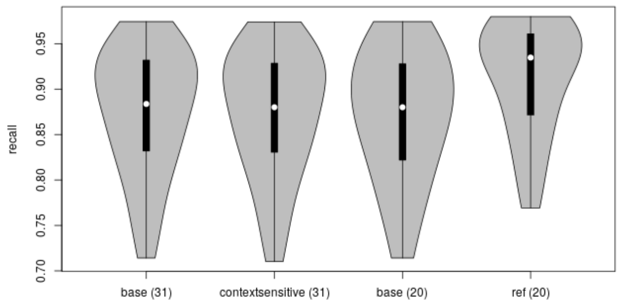

Figure 6: Recall of base vs context-sensitive analysis and reflection (ref) analysis. The numbers in brackets indicate the size of the data set used, the base analysis data are provided for both the full data set (column 1) and the reduced data set (column 3)

4.4 The Effectiveness of Dynamic Language Feature Support in Static Analysis

To answer RQ3 we compared the recall obtained by the base analysis with the analyses with reflection support being enabled. This allowed us to measure the effectiveness of state-of-the-art support for reflection and similar dynamic language features. Unfortunately, the additional reasoning *doop* has to perform is resource-intensive and timed out for several programs, as detailed in Table 2. Therefore, the results summarised in columns 3 and 4 in Figure 6 were obtained with a smaller dataset only consisting of 20 programs [8]. In general, the reflection support in *doop* is very effective – the median recall increases significantly from 0.884 to 0.935 .

4.5 Quantifying the Causes of Unsoundness

In order to answer RQ4, we removed tagged vertices from the CCTs and measured the percentage of false negatives (with respect to a given static analysis) still reachable as described in Section 3.7. The results for the base analysis are shown in Figure 7. This figure summarises the percentages of false negatives that can be explained by the presence of the respective class of language features across the dataset. The figure uses the aggregated categories, also showing statistical variation. Details are shown in Table 3. It turns out that dynamic invocations are only a minor source of false negatives, in particular the presence of `Method::invoke` can only explain less than 10% of the false negatives. Invocations triggered by methods invoked by the JVM and different types of dynamic allocations can explain the majority of false negatives. Note that the data set consists mainly of older programs, and features such as lambdas are likely to be under-represented [9]. The only programs where we found false negatives caused by dynamic access are *wct-1.5.2* and *guava-21.0*. Other categories that have overall little impact are allocations when objects are deserialised, dynamic proxies, and `Unsafe::getObject`. However, a significant part of programs uses dynamic proxies and `Unsafe::getObject`. More generally, we detected at least some usage of each of the features / patterns investigated when executing the programs in the data set.

Table 3 also contains the detailed classification of the false negatives left when running the static analysis with reflection support. The base analysis for the reduced data set is also included in order to make the base and the reflection data comparable. Figure 8 shows the variation of recall values across the data set. We observe that reflection support addresses a significant share of false negatives caused by `Method::invoke`. It addresses all false negatives caused by dynamic proxies (invocation handlers) in 3/5 programs, and all false negatives caused by allocation via deserialisation (although there were only 2 programs in this category). For the system category, the percentages increase, indicating that *doop* reflection support is relatively ineffective for these categories.

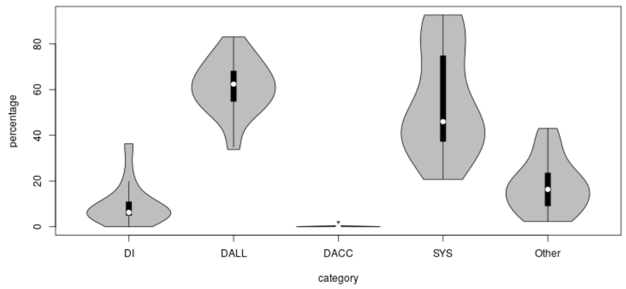

Figure 7: Causes of false negatives in the base static analysis

[8] *checkstyle-5.1, commons-collections-3.2.1, informa-0.7.0-alpha2, findbugs-1.3.9, fitjava-1.1, javacc-5.0, jena-2.6.3, jFin_DateMath-R1.0.1, jfreechart-1.0.13, jgrapht-0.8.1, jrat-0.6, jrefactory-2.9.19, marauroa-3.8.1, nekohtml-1.9.14, openjms-0.7.7-beta-1, oscache-2.4.1, pmd-4.2.5, quartz-1.8.3, trove-2.1.0, velocity-1.6.4*

[9] [16] contains an overview of the language features used by the *xcorpus* programs

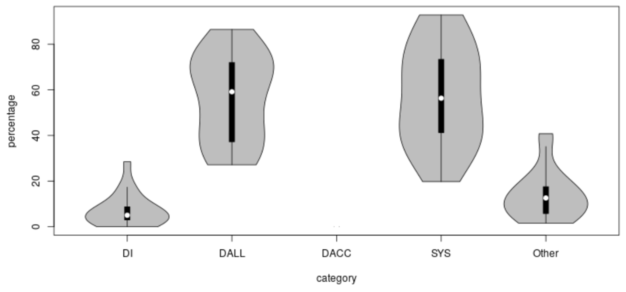

Figure 8: Causes of false negatives in the static analysis with reflection support

Table 3: Detailed classification of false negatives for the base analysis of the full dataset, and the base and reflection analysis on the partial dataset of 20 programs for which reflection analysis succeeded in percent (# - number of programs with more than one false negative in this category).

category (tag)	base (31)			base (20)			reflection		
	avg	stdev	#	avg	stdev	#	avg	stdev	#
#method.invoke	8.08	8.88	29	7.14	8.44	18	4.22	4.53	16
#lambda	0.10	0.16	16	0.10	0.18	10	0.01	0.04	1
#handler.invoke	1.45	1.98	24	1.73	2.42	14	2.98	4.13	14
#dynproxy.invoke	0.42	0.80	14	0.21	0.56	5	0.26	0.81	2
#class.newinstance	21.36	14.31	29	19.62	14.12	18	20.64	17.03	18
#constr.newinstance	5.97	6.41	28	4.76	5.53	17	5.61	6.82	17
#deserialize	0.01	0.06	3	0.02	0.08	2	0	0	0
#unsafe.getobject	0.15	0.30	9	0.18	0.35	6	0.27	0.52	6
#nativeallocation	40.00	12.67	31	41.81	12.7	20	36.24	14.41	20
#field.get	0.08	0.40	2	0	0	0	0	0	0
#nocallsite	52	22.51	31	47.27	19.34	20	52.49	20.57	20
#systemthread	2.57	3.25	31	3.41	3.8	20	4.82	4.88	20
other	17.94	11.39	31	17.96	11.14	20	13.83	10.74	20

We then analysed the percentage of false negatives left after all tagged vertices were removed from the CCTs, this is the number of uncategorised false negatives presented in the last columns labelled *Other* in Figures 7 and 8. It represents the (in)completeness of our automated cause analysis. We manually analysed a sample of false negatives in the *Other* category, aiming for a confidence level of 95% and a confidence interval of 5%. This yielded a sampling size of 373 for the base and 344 for the reflection analysis. We then randomly extracted the respective number of CCT paths from the respective CCT root to an uncategorised false negative, and inspected them. It turns out that they are dominated by a single pattern we refer to as *double-reflective factory*, which we discuss in some more detail next. This accounted for 54.4% of the uncategorised false negatives in the base analysis and 51.5% of the uncategorised false negatives in the analysis with reflection support. There are some other patterns we detected, we omit the detailed discussion for space reasons.

The double-reflective factory is a particular use of the factory design pattern [24] in conjunction with reflection, used to manage character sets. To illustrate this, consider the stack trace caused by an invocation of `System.out.println()` in Listing 4. Using both the base and the reflection analysis, `encodeLoop` is unreachable in the statically computed call graph. The encoder is created by the `Charset` (`sun.nio.cs.UTF`) which is created via reflection (`Class::forName` and `Class::newInstance`) by a `CharsetProvider` (`sun.nio.cs.FastCharsetProvider`) which is in fact itself also created using reflection by a service loader from jar manifest meta data. This is a triple factory, with two of the factories using reflective allocation. This is a good example of framework complexity Java is known for. While our analysis tags the factories as dynamically allocated, it does not do this to the objects created in those factories using plain object allocation with `new`. While an extension of our mechanism to cover such cases is possible, we decided not to do this in the scope of this work as our cause analysis had reached a coverage we considered as sufficient to confidently answer RQ4.

```
sun.nio.cs.UTF_8$Encoder::encodeLoop(Ljava/nio/CharBuffer;Ljava/
        nio/ByteBuffer;)Ljava/nio/charset/CoderResult
java.nio.charset.CharsetEncoder::encode(Ljava/nio/CharBuffer;Ljava
        /nio/ByteBuffer;Z)Ljava/nio/charset/CoderResult
sun.nio.cs.StreamEncoder::implWrite([CII)V
sun.nio.cs.StreamEncoder::write
java.io.OutputStreamWriter#write([CII)V
java.io.BufferedWriter#flushBuffer()V
java.io.PrintStream::newLine()V
java.io.PrintStream::println(Ljava/lang/String;)V
net.sourceforge.pmd.util.designer.MyPrintStream::println(Ljava/
        lang/String;)V
```

Listing 4: Stacktrace created by the invocation of `PrintStream::println`

We also sampled the *nocallsite* and *systemthread* categories, focusing on false negatives defined in applications or third-party libraries. It turns out that for the base analysis, 9/31 programs have such false negatives in the *systemthread* category, and 28/31 programs have such false negatives in the *nocallsite* category. Using the same sampling procedure as described above, we found that 86% of the application false negatives in the *nocallsite* category are caused by static initialisers (`<clinit>` methods) invoked by the JVM. Another example are invocations of `Runnable::run` methods through native dispatch from `java.security.AccessController::doPrivileged`. Reflective method invocations are also classified in this category due to the native dispatch in `sun.reflect.NativeMethodAccessorImpl#invoke0`, in the sampling set, this accounted for 4.8% of cases. Sampling application-defined methods classified as *systemthread* reveals that those can all be explained by invocations of `finalize` in application classes in the `Finalizer` thread.

4.6 Threats to Validity

The test unreflection process described in Section 3.3 has limitations as *junit* features such as tests with rules and custom runners were ignored, i.e., not unreflected. This has reduced the coverage of the oracle.

Test flakiness [44] is a know issue that affects test outcomes, including coverage. We found small variations in test coverage in 14 of the 31 programs between executions. Figure 2 uses averages from five runs, the oracle used was generated by a single run. The reason for this decision is the high cost of oracle generation (see Table 2). In some cases, a slightly larger oracle could have been obtained by running the (instrumented) tests multiple times, and merging the constructed CCTs.

The classification method discussed in Section 3.7 can explain most, but not all false negatives. We addressed this by sampling and manual analysis.

The tagging of lambdas relies on naming patterns used by the OpenJDK compiler. There is a possibility that some of the library code within the analysis scope has been compiled with a different compiler using a different convention. This would have resulted in more false negatives not being classified, as mentioned above, this was addressed by sampling.

Tagging with `#nocallsite` relied on a static pre-analysis to collect the call sites in methods. For libraries, this depends on the library version used. There is a chance that in some cases programs use custom class loaders, choosing a different version of the class. This would have resulted in methods being incorrectly tagged as `#nocallsite`, and in an over-reporting of false negatives in this category.

5 CONCLUSION

We have studied the recall of static call graph construction, and the impact various analysis settings have on recall. The recall values we measured are an approximation obtained by substituting possible by known program behaviour recorded by executing high-coverage test suites. The results indicate that there are significant gaps in the statically constructed call graphs – many methods that are known to be reachable are missed. While state-of-the-art analysis with reflection support can significantly improve recall, its high cost renders it impracticable for many practical application. The results further indicate that some language features suspected in being a major cause of unsoundness (in particular `Method::invoke`) play only a minor role. The classification of static analysis false negatives hints at directions for future research to boost recall: to include the analysis of native methods and the JVM itself [10].

The fact that dynamic analysis reveals a significant number of false negatives in the static analysis also indicates that hybrid techniques can be very effective. In particular, generated tests can be used to discover program behaviour that is out of reach of static analyses. However, there are limitations: like static reflection analysis, test generation is expensive [16], and our study demonstrated that it is not as effective in discovering dynamic program behaviour as the manually written tests.

ACKNOWLEDGEMENTS

This project was supported by the Science for Technological Innovation National Science Challenge of New Zealand, project "Closing the Gaps in Static Program Analysis" (PROP-52515-NSCSEED-MAU). The work of the second author was supported by Oracle Labs, Australia. We would like to thank Francois Gauthier, Paddy Krishnan, James Noble and Yannis Smaragdakis for their valuable feedback.

REFERENCES

[1] [n. d.]. Invokedynamic Rectifier / Project Serializer. http://www.opal-project.de/DeveloperTools.html, accessed 14 Jan 2019.
[2] [n. d.]. Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks.
[3] Karim Ali and Ondřej Lhoták. 2013. Application-only call graph construction. In *Proc. ECOOP'13*. Springer.
[4] Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *Proc. ECOOP'13*. Springer.
[5] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. 2015. Combining type-analysis with points-to analysis for analyzing Java library source-code. In *Proc. SOAP'15*. ACM, 13–18.
[6] Glenn Ammons, Thomas Ball, and James R Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices* 32, 5 (1997), 85–96.
[7] Lance Andersen. 2006. JDBC™4.0 Specification. *JSR* 221 (2006), 1–126.
[8] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *Proc. ASE'12*. ACM.
[9] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA'96*. ACM.
[10] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA'06*. ACM.
[11] Eric Bodden. 2012. InvokeDynamic Support in Soot. In *Proc. SOAP'12*. ACM.
[12] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. ICSE'11*. ACM.
[13] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proc. OOPSLA'09*. ACM.
[14] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).
[15] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience* 34, 11 (2004), 1025–1050.
[16] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. XCorpus–An executable Corpus of Java Programs. *JOT* 16, 4 (2017), 1:1–24.
[17] Jens Dietrich, Li Sui, Shawn Rasheed, and Amjed Tahir. 2017. On the construction of soundness oracles. In *Proc. SOAP'17*. ACM.
[18] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *Proc. WODA'03*.
[19] Stephen Fink and Julian Dolby. 2012. WALA–The TJ Watson Libraries for Analysis. https://github.com/wala/WALA.
[20] Brian Foote and Ralph E Johnson. 1989. Reflective facilities in Smalltalk-80. In *Proc. OOPSLA'89*. ACM.
[21] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static Analysis of Java Dynamic Proxies. In *Proc. ISSTA'18*. ACM.
[22] George Fourtounis and Yannis Smaragdakis. 2020. Deep Static Modeling of invokedynamic. In *Proc. ECOOP'19*.
[23] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proc. ESEC/FSE'11*. ACM.
[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *Proc. ECOOP'93*. Springer.
[25] Brian Goetz. 2012. Translation of Lambda Expressions. http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html.
[26] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps Don't Lie: Countering Unsoundness with Heap Snapshots. In *Proc. OOPSLA'17*. ACM.
[27] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the heap: Ultra-scalable static analysis with heap snapshots. In *Proc. ISSTA'18*. ACM.
[28] Neville Grech, George Kastrinis, and Yannis Smaragdakis. 2018. Efficient reflection string analysis via graph coloring. In *Proc. ECOOP'18*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[29] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. In *Proc. OOPSLA'17*. ACM.
[30] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call graph construction in object-oriented languages. In *Proc. OOPSLA'97*. ACM.
[31] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. In *Proc. PLDI'94*. ACM.
[32] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Proc. CETUS'11*.
[33] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. 2017. Challenges for Static Analysis of Java Reflection-Literature Review and Empirical Study. In *Proc. ICSE'17*. IEEE.
[34] Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 1 (1979), 121–141.
[35] Ondrej Lhoták. 2007. Comparing call graphs. In *Proc. PASTE '07*. ACM.
[36] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. In *Proc. OOPSLA'18*. ACM.

[10] We note that *doop* models some native methods, including `Object::clone` and some methods in `java.lang.System`, `sun.misc.Unsafe`, `java.io.UnixFilesystem`, `java.lang.Thread`, `java.lang.ref.Finalizer`, and `java.security.AccessController`.
However, these models are still unsound and, as Grech et al. noted, such manual modelling "... is hard. Extra native operations get added in every release of the JDK and analysis authors typically do not keep up with them." [26].

[37] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proc. ESEC/FSE'18*. ACM.

[38] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing reflection resolution for Java. In *Proc. ECOOP'14*. Springer.

[39] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and Analyzing Java Reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 2 (2019), 7.

[40] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. The Java virtual machine specification: Java SE 8 edition, 2015. (2015). https://docs.oracle.com/javase/specs/jvms/se8/html/index.html.

[41] Jie Liu, Yue Li, Tian Tan, and Jingling Xue. 2017. Reflection Analysis for Java: Uncovering More Reflective Targets Precisely. In *Proc. ISSRE'17*. IEEE.

[42] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *CACM* 58, 2 (2015), 44–46.

[43] Benjamin Livshits, John Whaley, and Monica S Lam. 2005. Reflection analysis for Java. In *Proc. APLAS'05*. Springer.

[44] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proc. FSE'14*. ACM.

[45] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at your own risk: the Java unsafe API in the wild. In *Proc. OOPSLA'15*. ACM.

[46] Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. 1998. An empirical study of static call graph extractors. *ACM TOSEM* 7, 2 (1998), 158–191.

[47] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. NDSS'05*. Internet Society.

[48] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Proc. OOPSLA'07*. ACM.

[49] Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. 2019. Java reflection API: revealing the dark side of the mirror. In *Proc. FSE'19*. ACM.

[50] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. 2016. Call graph construction for java libraries. In *Proc. FSE'16*. ACM, 474–486.

[51] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proc. ISSTA'19*. ACM.

[52] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Proc. SOAP'18*. ACM.

[53] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[54] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. In *Proc. OOPSLA'19*. ACM.

[55] Olin Shivers. 1991. *Control-flow analysis of higher-order languages*. Ph.D. Dissertation. Carnegie Mellon University.

[56] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *Proc. APLAS'15*. Springer.

[57] Yannis Smaragdakis and George Kastrinis. 2018. Defensive Points-To Analysis: Effective Soundness via Laziness. In *Proc. ECOOP'18*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[58] Brian Cantwell Smith. 1984. Reflection and semantics in Lisp. In *Proc. POPL'84*. ACM.

[59] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features-A Benchmark and Tool Evaluation. In *Proc. APLAS'18*. Springer.

[60] Li Sui, Jens Dietrich, and Amjed Tahir. 2017. On the Use of Mined Stack Traces to Improve the Soundness of Statically Constructed Call Graphs. In *Proc. APSEC'17*. IEEE.

[61] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. In *Proc. OOPSLA'00*. ACM.

[62] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Proc. APSEC'10*. IEEE.

[63] Frank Tip and Jens Palsberg. 2000. Scalable Propagation-based Call Graph Construction Algorithms. In *Proc. OOPSLA'00*. ACM.