Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs

Michael Reif
TU Darmstadt, Germany

Florian Kübler
TU Darmstadt, Germany

Michael Eichberg
TU Darmstadt, Germany

Dominik Helm
TU Darmstadt, Germany

Mira Mezini
TU Darmstadt, Germany

ABSTRACT

Call graphs are widely used; in particular for advanced control- and data-flow analyses. Even though many call graph algorithms with different precision and scalability properties have been proposed, a comprehensive understanding of sources of unsoundness, their relevance, and the capabilities of existing call graph algorithms in this respect is missing.

To address this problem, we propose Judge, a toolchain that helps with understanding sources of unsoundness and improving the soundness of call graphs. In several experiments, we use Judge and an extensive test suite related to sources of unsoundness to (a) compute capability profiles for call graph implementations of Soot, WALA, DOOP, and OPAL, (b) to determine the prevalence of language features and APIs that affect soundness in modern Java Bytecode, (c) to compare the call graphs of Soot, WALA, DOOP, and OPAL – highlighting important differences in their implementations, and (d) to evaluate the necessary effort to achieve project-specific *reasonable* sound call graphs.

We show that soundness-relevant features/APIs are frequently used and that support for them differs vastly, up to the point where comparing call graphs computed by the same base algorithms (e.g., RTA) but different frameworks is bogus. We also show that Judge can support users in establishing the soundness of call graphs with reasonable effort.

CCS CONCEPTS

• **General and reference** → *Evaluation*; • **Software and its engineering** → Polymorphism.

KEYWORDS

call graph construction, static analysis, soundness

ACM Reference Format:
Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330555

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330555

1 INTRODUCTION

Call graphs (CG) are the foundation of inter-procedural analyses and many algorithms for constructing CGs exist [3, 4, 8, 12, 33, 36, 42, 45]. The focus of that research has mainly been on precision and/or scalability [19, 23, 45], thereby often covering only standard (non-)virtual method calls. Other (problematic) language features, e.g., Serialization or Java's reflection API, are ignored; the developers deliberately accept so-called *soundy* [28] CGs.

One reason for deliberately accepting unsoundness is the trade-off between soundness and precision/scalability. Another potential reason is the trade-off between the development costs for supporting such language features or specific APIs and the perceived value of doing so. In the end their support is only relevant if they are used in applications. For example, support for Java 7's invokedynamic only became relevant after Java 8. The latter's lambda expressions are compiled using invokedynamic instructions.

Given that all CG algorithms are soundy to varying degrees, the question is, how to assess their capabilities in practice? What is the impact of soundiness when analyzing real applications? The question is highly relevant, since the occurrence of the ignored features in real software can have a devastating impact on the constructed CGs. This impact depends on the locations of uncovered language features in the project and, hence, is best assessed in a project-specific way. For instance, Xalan's main method uses reflection in combination with system properties. An implementation of, e.g., Rapid Type Analysis (RTA) [8] that does not cover these features would only contain calls to the reflection API; missing the application methods. As a result, the CG would reach only a fraction of the methods it should actually reach.

A second question is, how do existing static analysis frameworks (Soot [46], WALA [21], DOOP [37], and OPAL [15]) compare in terms of costs and capabilities of their CG algorithms? The question is relevant when deciding which algorithms to use, especially, since preliminary studies [30, 35] have shown that CG implementations vary more widely than expected. The study presented in Sec. 4 of this paper reveals that due to differences in implementation decisions and set of supported features the RTA-based CGs of Soot, WALA, and OPAL have between 3195 and 75817 reachable methods (for XCorpus' jasml), a factor of up to 23 times.

Unfortunately, we lack methods and tools to answer these questions in a systematic way. That is, we have analyses for software, but we lack means for systematically analyzing these analyses, so as to understand the capabilities of CG algorithm implementations w.r.t.

supported language features and core APIs and possible sources of unsoundness when analyzing a specific application.

This understanding is a prerequisite to enable understanding and reproducing the results of inter-procedural static analyses. Without such an understanding, comparing static analyses building on top of CGs computed by different frameworks is incoherent.

This paper contributes in two ways to advancing the state-of-the-art.

First, we provide a toolchain, called Judge, for analyzing CG algorithms with respect to the language features they cover in both general terms as well as in a project-specific manner. Judge uses a test suite to build profiles of algorithms under investigation. Given a profile and the features of an application for which we want to construct the CG, Judge finds and documents sources of unsoundness in the application.

Second, we use Judge to conduct a comprehensive study of the capabilities of CGs constructed by Soot, WALA, DOOP, and OPAL. More specifically, we answer the following research questions: (RQ1:) Which language and API features are used how frequently by which kind of code? (RQ2:) How do Soot, WALA, DOOP, and OPAL compare to each other w.r.t. runtime costs and feature support? (RQ3:) Which CG algorithms are suitable for a specific application kind? (RQ4:) Given support for manually tuning the entry-points considered by an algorithm, how much effort is necessary to increase the soundness of a CG to an acceptable level.

We analyzed the top 50 Maven libraries, the XCorpus [13], the top 15 Android projects and five programs of each of the following JVM hosted languages: Kotlin [16], Groovy [32], Clojure [20], and Scala [24]. Not only does the study provide information about the prevalence of the advanced features in the wild; it also can guide the creation of benchmark suites for testing CG algorithms. For example, we observe that the usage of advanced features that affect the soundness of the CGs differs significantly when comparing the XCorpus with the current top 50 libraries found on Maven; the latter makes use of more features and also use those features that are found in both corpora more frequently.

The remainder of the paper is organized as follows. Section 2 presents Judge. Section 3 shortly presents the test suite. Our study is presented in section 4. Related work is discussed in section 5 and the paper is concluded in section 6.

2 EVALUATION TOOLCHAIN

Figure 1 depicts the building blocks and the workflow of Judge for analyzing CG algorithms. Judge's input is (a) a test suite comprising a test case for each language feature with respect to which we want to analyze the soundness of the CG algorithms under investigation and (b) a project for which we want to investigate the project-specific unsoundness of CG algorithms.

The upper part runs all CG algorithms on the test suite and computes profiles reporting whether the algorithms *passes* the tests or not. The lower part of the workflow computes the CG for the input project with different algorithms and in parallel evaluates the prevalence of the features under investigation in the project code. Given the CG of a project P constructed by algorithm AL, the occurrence of the features $FSET$ under investigation in P's code, $FSET$, and the AL's profile, Judge reports potential sources of unsoundness of AL in P's CG.

2.1 Call Graph Algorithm Profiles

Judge supports the analysis of the various CG algorithms offered by the frameworks: 4xWALA, 4xSoot, 1xDOOP, and 1xOPAL. The approach's first part (Steps 0, 1, and 2 in Figure 1) computes for each algorithm a profile which lists for each considered language feature whether it is supported or not.

To construct the profile, the approach uses *CG test cases*, each testing one specific feature that is relevant when constructing CGs. A test case consists of a minimal, executable program that uses the feature and Java annotations that specify the expected edges. Depending on the test case, we either specify a specific call edge or an indirect call. For instance, the test case for the *resolution of trivial reflective calls* contains a minimal, executable program that performs a reflective call, where the String that identifies the call target is directly specified. Here, the annotation specifies an *indirect call* target—we only expect that the target method is eventually called, i.e., the CG does not have to (but may) contain a direct edge from the call site of the reflective call to the method invoked finally. Test cases related to standard mono-/polymorphic calls, on the other hand, specify the expected *(direct) call edge*.

There are two classes of test cases. The first class consists of basic test cases that can be created using Java code. These are defined in markdown files (.md) that contain a high-level description of the test case along with the source code.

The second class consists of test cases that cannot be generated by the Java 8 compiler. These *Advanced Test Cases* (cf. Figure 1) are manually compiled using another compiler (e.g. Java 10 or Scala), created via bytecode engineering, or by replaying code evolution scenarios. The study of the JVMSpec led us to test cases that represent valid bytecode but cannot be generated by the Java compiler. For example, the JVM supports so-called MethodHandle Constants which are primarily intended to be used by other JVM-hosted languages. Furthermore, due to code evolution it may happen that an interface SuperI defines a default instance method m and its subinterface SubI a corresponding static method m. That is, both methods have the exact same signature and only differ in the access modifier (e.g., static). Such bytecode is legal and works reliably, but cannot be created using Java source code.

Overall, we define 122 test cases and, therefore, investigate 122 features which we grouped in 23 categories (cf. Table 1).

The profile constructions works as follows. We parse and extract the respective test cases and then compile them (Step 0). Given the set of all tests cases—which includes the advanced test cases—and using one .jar per test case, we run all 10 CG algorithms for each test case (Step 1) and then check if the CG algorithm supports the respective feature (Step 2). The latter parses the test jars' annotations and checks for each found annotation whether the CG contains the respective edge.

2.2 Querying for Features

To understand the prevalence of features affecting the soundness of CGs (cf. Step 3/blue area in Figure 1), Judge uses Hermes [34].

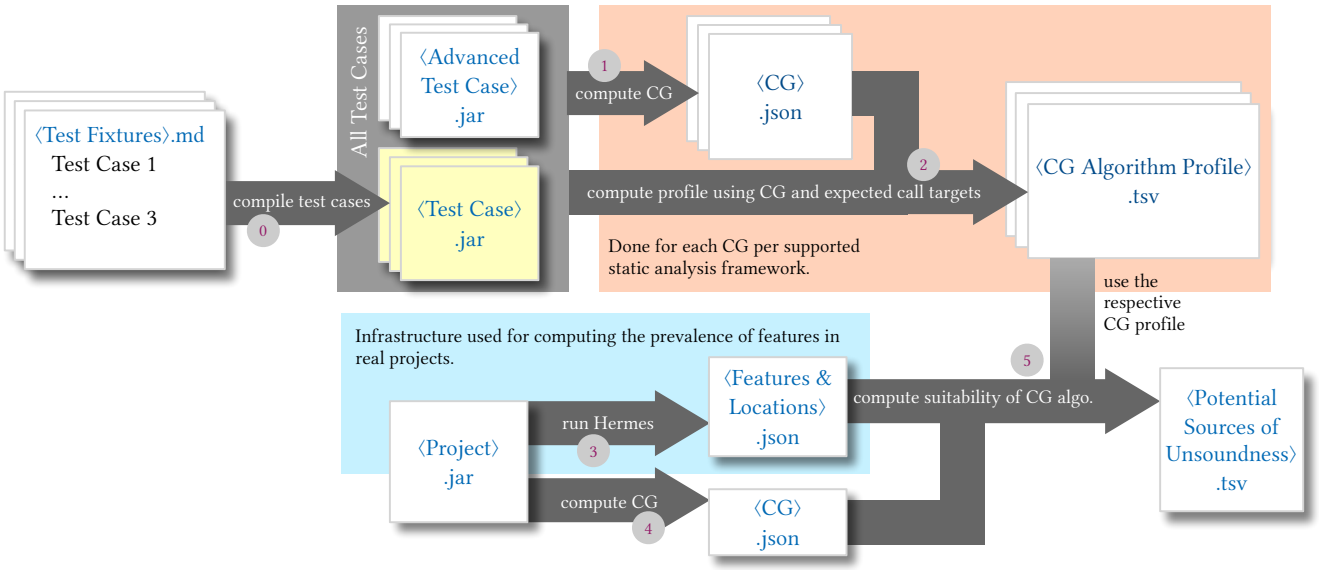


Figure 1: Call Graph Analysis Toolchain—Judge.

Table 1: Overview of the Test Suite.

Category	Abbreviation	# Test Cases
Classloading	CL	4
Dynamic Proxies	DP	1
Interface Default Methods	J8DIM	6
Static Interface Methods	J8SIM	1
Java 8 invokedynamics	MR/Lambda	11
JVM Calls	JVMC	5
Library Analysis	LIB	5
Trivial Reflection	TR	9
Locally Resolveable Reflection	LRR	3
Context-sensitive Reflection	CSR	4
Method Handles	MH	9
Class.forname Exceptions	CFNE	4
Non-virtual Calls	NVC	6
Serialization	Ser	9
Externalizable	ExtSer	3
Lambda Serialization	LamSer	2
Signature Polymorphic Methods	SPM	7
Static Initializers	SI	8
TYPES	-	6
Unsafe	-	7
Virtual Calls	VC	4
Java 9/10 Features	J9+	2
Non-Java Bytecode	NJB	6
Total		122

The latter executes code queries against a large code base and then produces reports on the queries' findings.

Each query is an analysis that checks if a specific feature is found in a given code base. The result is a report that lists the locations (in terms of the instructions' program counters) that use a feature along with the Hermes feature id.

We developed Hermes queries to derive Hermes features that map all test case ids to Hermes feature ids. All queries perform a most-conservative intra-procedural analysis. Ergo, test cases that require an inter-procedural analysis, e.g., test cases related to reflective calls that test if a framework is able to track strings across method call boundaries to (soundly and precisely) resolve reflective call targets, are only partially covered. Writing queries for these test cases would be subject to false positives and false negatives; the query would require information about the flow of strings in the application and no such analysis exists that is sound and precise. Therefore, it would be impossible to use those queries to reliably identify code locations that are sources of unsoundness.

However, for these test cases we write queries that determine that the local analysis is inconclusive and then flag the method accordingly. Such queries often handle multiple test cases by reporting that a finding belongs to one of multiple test cases, i.e., the query reports an id consisting of all test cases the finding may matches. For example, test cases of context-sensitive reflection are grouped because the query cannot distinguish where the method's parameter originates from.

Hence, the queries *only* derive 107 features for 122 test cases. Altogether, we developed 15 queries for Hermes.

2.3 Project-specific Call Graph Analysis

For the project-specific evaluation of an algorithm, we compute its CG for the project (Step 4 in Figure 1). Additionally, we use Hermes to find the locations of all features that may affect the soundness (Step 3 in Figure 1). Finally, the computed CG is used to determine all reachable methods that use unsupported features

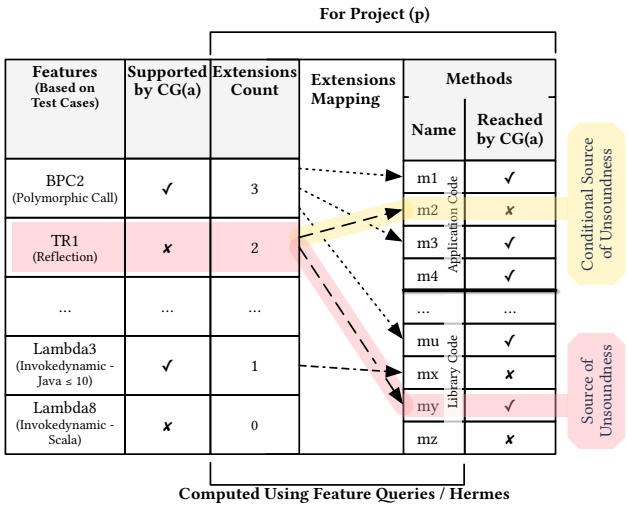


Figure 2: Project-specific call graph matching.

(Step 5 in Figure 1). This enables the identification of the *initial* sources of unsoundness.[1]

Figure 2 illustrates the project-specific assessment of a CG algorithm. The first two columns are project agnostic and represent the CG algorithm's profile: the first one lists Hermes' features ids (which map to the respective test cases); the second one identifies a feature as being supported or not. Columns three to six are project specific: Column three (Extensions Count) shows how often a feature was found by the respective Hermes query—in our case, the project contained three polymorphic calls, two reflective calls, one Java invokedynmic instruction, and zero Scala invokedynamics. The fourth column represents the mapping between the occurrences of a feature (column 3) and its locations/methods (column 5). Finally, column six shows whether the methods where the features were found are reachable from the constructed CG—i.e., are an *immediate source of unsoundness*—or not.

With respect to the reflection usage of method my, we make two observations: 1) the CG algorithm does not support the resolution of reflective method calls and 2) method my is already reachable. Hence, this reflection usage in my is a *source of unsoundness* because it knowingly leads to missing call edges. The reflective usage in method m2 is—in contrast—not reachable according to the current CG and is so far a *conditional source of unsoundness*; i.e., it would be another source of unsoundness if the method would be reached. In other words, conditional sources of unsoundness are potentially relevant because the impact of known unsoundly handled features on the constructed CG remains unknown.

3 TEST SUITE

In the following, we discuss our test suite by first giving a high-level overview of the test categories (cf. Table 1[2]) before we discuss individual test cases.

[1]Sources of unsoundness are always only potential sources of unsoundness because we do not check whether the instructions themselves are reachable.
[2]The test suite is published along Judge: https://bitbucket.org/delors/jcg/.

3.1 Test Categories

Classloading: Using a java.lang.ClassLoader it is possible to load and use a specific class in multiple (incompatible) versions. **Direct Calls**: Non-virtual method calls, i.e., constructor calls, **super** calls, **private** method calls and **static** method calls. **Dynamic Proxies**: Java's Dynamic Proxy API creates (via runtime bytecode engineering) type safe proxy classes which will then forward the calls—using Java reflection—to a previously specified handler class. **Java 8 Polymorphic Calls**: Java 8 added the possibility to define concrete instance and static methods in interfaces. **JVM Calls**: Calls of those methods that are (only) done by the JVM due to some event, such as calling start on a Thread. In that case the JVM will eventually call the Thread's run method. **Lambdas and Method References**: Lambda and method reference (e.g., String::length) based invocations (as introduced with Java 8). **Library Analysis**: As discussed in [33], the target of a method call in a library may require call-by-signature resolution when computing CGs *just* for the library. **Polymorphic Calls**: Virtual (interface) method calls as already available before Java 8. **Trivial Reflection**: Usage of the classical reflection API (java.lang.reflect.*) where the call target is immediately available (e.g., Class.forName("XYZ")). **Locally Resolvable Reflection**: Usage of the classical reflection API where an intra-procedural control-/data-flow analysis is required to resolve the call targets. **Context-sensitive Reflection**: Usage of the classical reflection API where an inter-procedural control-/data-flow analysis is required to resolve the call targets. **Method Handles**: Reflective calls using the java.lang.invoke.* APIs and Java 7's MethodHandle API. **Serialization**: When objects are (de)serialized, the JVM will call the respective (de)serialization methods. **Signature Polymorphic Methods**: Signature polymorphic method calls w.r.t. java.lang.MethodHandle's invoke and invoke-Exact methods [18] (In these cases the method descriptor used at the call site does not have to match the signature of the called method). **Static Initializers**: When a class is used for the first time, its static initializer will be invoked by the JVM. **Types**: Type casts and **instanceof** checks can be performed using language features or using java.lang.Class' API. **Unsafe**: Using Java's sun.misc.Unsafe API [29] direct memory manipulations using Java-level code is possible. **Java 9/10**: Features added with 9 and 10, such as **private** interface methods. **Non-Java Bytecode**: Legal JVM bytecode that cannot be created using Java, but which was added to the JVM to support other JVM hosted languages such as Clojure, Groovy, Kotlin, or Scala.

3.2 Custom Native Methods

We did not add explicit test cases related to custom native methods because none of the frameworks support cross-language analyses. Nevertheless, we developed a Hermes query to find respective calls and (always) flag them as potential sources of unsoundness.

3.3 Test Case Design

For systematically designing the test suite, we studied the Java Virtual Machine Specification (JVMSpec) [18] and the Java core APIs (java.*). When designing the test cases, we tried to ensure that a test case will only succeed if the algorithm explicitly supports the respective feature. This is, however, not possible in all cases; some test cases are simply supported due to an algorithm's inherent

imprecision. For example, some of the test cases related to Type Narrowing or the Unsafe API *just* manipulate references and can therefore negatively affect soundness in those algorithms that are points-to information based. If those algorithms do not model the effects of, e.g., the Unsafe API, the points-to information will be incorrect—potentially leading to unsound results. CG algorithms, such as Class-Hierarchy Analysis (CHA), that just rely on the type information found in the bytecode handle related scenarios in a sound manner; they just assume all subclasses.

4 THE STUDY

We perform four experiments to answer our research questions: (RQ1) how prevalent are the language and API features; (RQ2) how do the frameworks compare to each other; (RQ3) which framework is best suited for which kind of code base; (RQ4) how much effort is necessary to get a sound call graph.

4.1 Setup

All measurements are done using WALA 1.5.0, Soot 3.1.0, OPAL's develop branch [2], and DOOP's master branch [1]. From WALA we use the following algorithms: WALA_{RTA}, $\text{WALA}_{\text{0-CFA}}$, $\text{WALA}_{\text{N-CFA}}$[3], $\text{WALA}_{\text{0-1-CFA}}$—all configured with the FULL reflection option. WALA requires to specify packages to be excluded from the analysis. For the comparative analysis (Experiment 2 (see 4.3)) we excluded no package, whereas for the experiment related to RQ3 we use the predefined *Java60RegressionExclusions* to ensure termination. For all Soot call graphs (Soot_{CHA}, Soot_{RTA}, Soot_{VTA}, and $\text{Soot}_{\text{SPARK}}$ [26]) we use the options: *safe-forname* and *safe-newinstance*. This options make Soot consider all types as instantiated when Class.forName or Class.newInstance is used. We could not use *types-for-invoke* due exceptions being thrown [41]. Furthermore, we use *include-all* to ensure that no packages are filtered. Our library test cases are additionally started with *library:signature-resolution* and *all-reachable* to make use of Soot's capabilities to analyze library code. DOOP_{CI}'s call graph is set to be *context-insensitive* with *classical-reflection* turned on. For OPAL_{RTA}, we use the standard configuration.

All test cases w.r.t. libraries are started with the respective library entry points. We perform all experiments on a server with two Intel Xeon E5-2620 CPUs and 64 GB RAM.

4.2 Experiment 1

Our corpus for analyzing the prevalence of language and API features (RQ1) includes the XCorpus [13], the top 50 libraries from Maven Central [31] (from July 2018), the top 15 apps from Google's Playstore (from January 2018), plus five Clojure [20], Groovy [32], Kotlin [16], and Scala [24] projects.

Table 2 visualizes the results using a heatmap. It shows the relative frequency of each feature (cf. Feature column) within each corpus. We include the OpenJDK column as a separate corpus because most corpus projects are built upon it and, hence, partially use its features. A feature's relative frequency is color coded using a logarithmic scale as shown in the legend of Table 2. Slightly yellow boxes (■) identify unused features and red boxes (■) those found in ≥ 5% of all methods; we chose 5% because only 7 features occur in more than 5% of all methods. Features used in no corpus (e.g., Groovy invokedynamics, or the serialization of lambdas) and always soundly resolved features (e.g., standard poly-/monomorphic call) are not included.

[3]We use N=1 throughout the whole evaluation.

■ *All the API and language features supported by Java up to version 7 are used widely across all code bases.*

The most frequently used feature that was introduced with Java ≥ 8 is the call of static interface methods (*J8DIM6*). 12% of all methods of the top 50 Maven projects use them; Scalatest [22] is responsible for ≈ 90% of all uses. Clojure and Android code have not yet adapted Java 8 call semantics. Other Java 8 features, e.g., MethodHandle constants, are rarely used; primarily by the Nashorn library.

■ *Support for Java 8 is a must, given the frequent use of Java 8 call semantics features in modern code (J8DIMX), unless one analyzes only Android or Clojure code.*

Serialization-related functionality (*Ser3-7,9, ExtSer*) and Java's Reflection API (cf. *TR, LRR, CSR*) are both used with medium frequencies; also in modern code.

■ *Supporting classical Reflection and Serialization is strongly recommended, independent of the source code's age.*

Many features (e.g., method references *MR*), Java's MethodHandle API (*MH*), native methods (cf. *native*), or Java's Unsafe API (cf. *Unsafe3-7*) occur with varying frequency and not in all corpora.

■ *Support for many features is only required in specific scenarios.*

■ *The distribution of the feature usage is very different for the XCorpus when compared to the JDK 8 and/or the other corpora, therefore its representativeness for evaluating CG construction algorithms is limited. In particular, the usage of the Lambdas and the MethodHandle API increases, when we compare its usage frequency in the XCorpus vs. the top 50 Maven libraries.*

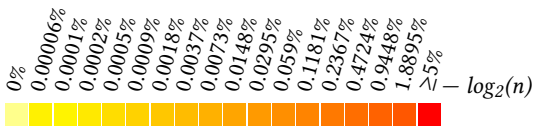
4.3 Experiment 2

In this experiment we compare different CG algorithms. We first compute each algorithm's feature profile. Next, we construct the CGs for five XCorpus projects (*jasml*, *javacc*, *jext*, *ProGuard*, and *sablecc*) to assess the CGs size and construction times. We select these projects because they all have (I) well-defined main classes, and (II) can be processed by at least one CG algorithm of each framework. We run all CG generators once on all five projects including the Java Runtime Environment 1.6_30 from DOOP's benchmark project [38]. The later is chosen to attain comparability w.r.t. the runtime; we set a timeout to 90 minutes.

Computing Call Graph Algorithm Profiles. Table 3 summarizes the computed algorithm profiles. The first column shows the test categories. Columns two to ten show for each test category the individual test results per CG algorithm. A cell's symbol indicates whether all (●), some (◐), or none (○) of the tests succeeded; the numbers represent the number of succeeded vs. all tests.

Table 3 shows that basic language features like static initializers (*SI*), (non-)virtual calls (*(N)VC*), and type casts (*TYPES*) are well

Table 2: Feature Prevalence across different corpora.



Feature	OpenJDK 8	XCorpus	Top50Maven	Scala	Groovy	Kotlin	Clojure	Android
J8DIM1								
J8DIM2								
J8DIM3								
J8DIM4								
J8DIM5								
J8DIM6								
JVMC1								
JVMC2								
JVMC3								
JVMC4								
JVMC5								
Lambda1								
Lambda2								
Lambda3								
Lambda4								
Lambda5								
Lambda6								
Lambda7								
LIB3								
LIB4								
LIB5								
MR1								
MR2								
MR3								
MR4								
MR5								
MR6								
MR7								
Native								
Ser1								
Ser3								
Ser4								
Ser5								
Ser6								
Ser7								
Ser8								
Ser9								
ExtSer1								

Feature	OpenJDK 8	XCorpus	Top50Maven	Scala	Groovy	Kotlin	Clojure	Android
ExtSer2								
ExtSer3								
SI1								
SI2								
SI3								
SI4								
SI5								
SI6								
SI7								
SI8								
MH1.1								
MH1.2								
MH2								
MH3								
MH4								
MH5								
MH6								
MH7								
MH8								
TR1								
TR2								
TR3								
TR4								
TR5								
TR6								
TR7								
TR8								
TR9								
LRR1								
LRR2								
CSR1+CSR2								
LRR3+CSR3								
Unsafe1								
Unsafe2								
Unsafe3								
Unsafe4								
Unsafe5								
Unsafe6								
Unsafe7								

supported. Except of two static initializer cases: the first one is not supported by $\text{Soot}_{\text{SPARK}}$, DOOP_{CI}, and WALA and the second one is not supported in WALA. *SI4* models a case when a Java 8's interface's static initializer must be called. An unexpected behavior is shown by $\text{WALA}_{\text{N-CFA}}$. It can only handle type casts that are performed using Java's explicit cast and **instanceof** APIs, but does not support built-in operators, i.e., **instanceof** or type casts of the form (String)o;.

Serialization-related methods (*Ser*) are not well supported by WALA and DOOP, are slightly better supported by SOOT and are best supported by OPAL (≈ 50%). The methods (in particular: readObject and writeObject—which will be called by the JVM) must be considered when object (de-)serialization occurs in reachable Methods.

Java 8 language features, such as default methods (*J8DIM*), lambdas, and method references (*MR*) are mostly correctly handled by WALA and OPAL but not supported by Soot and DOOP. Furthermore, OPAL is the only framework that supports the new method handle API (*MH*) and signature polymorphic methods (*SPM*).

As Table 3 shows, support for Java's reflection API varies, but all—except of $\text{WALA}_{\text{N-CFA}}$—provide at least some support. Moreover, Soot's reflection options enable it to resolve all advanced reflection test cases (*LRR* and *CSR*); calls to Class.newInstance are resolved to all initializers in the project.

Table 3 shows that only the basic algorithms: Soot_{CHA}, Soot_{RTA}, WALA_{RTA}, and OPAL_{RTA} *support* Java's Unsafe API as well as the Dynamic Proxy API. Here, the imprecision of CHA/RTA benefits the support of those two APIs.

Only OPAL supports non-Java bytecode (*NJB*) and Java 9/10 features (*J9+*).

Performance Comparison. The performance results are shown in Table 4. Column one lists the project, column two gives the number of all methods including the JDK and column three the number of project methods. The remaining columns list for each CG algorithm the number of reached methods and the GG's construction times for each algorithm.

OPAL is the fastest framework; All of WALA's context-sensitive CGs timed out; DOOP's has the slowest call graph generator that finished in time, followed by $\text{WALA}_{\text{RTA}}$and Soot. The CG constructed by RTA algorithms of Soot, WALA, and OPAL vary extremely. This is partially due to the different handling of basic virtual methods calls which all handle sound, but with very different precision. Other reasons are the supported features as well as the different usage of cast information.

```
1 Collection c1 = new LinkedList();
2 Collection c2;
3 if(cond){ c2 = new ArrayList(); } else { c2 = new Vector(); }
4 c2.add(null); // Call site
5 Collection c3 = new HashSet();
```

Listing 1: Precision Example

Listing 1 explains part of the difference. The three local variables c1, c2, and c3 are assigned different subtypes of Collection, namely LinkedList, ArrayList, Vector, and HashSet. The call on line 4 is then resolved differently. WALA considers all instantiated subtypes of Collection. Soot computes an upper type bound for c2 and the call is

Table 3: Support of language features and core APIs of Soot, WALA, OPAL, and DOOP's call graphs.

Category	$Soot_{CHA}$	$Soot_{RTA}$	$Soot_{VTA}$	$Soot_{SPARK}$	$WALA_{RTA}$	$WALA_{0\text{-}CFA}$	$WALA_{N\text{-}CFA}$	$WALA_{0\text{-}1\text{-}CFA}$	$OPAL_{RTA}$	$DOOP_{CI}$
CL	◐ 4/6	◐ 4/6	◐ 4/6	◐ 3/6	◐ 4/6	◐ 4/6	◐ 2/6	◐ 4/6	◐ 4/6	◐ 4/6
DP	● 1/1	● 1/1	○ 0/1	○ 0/1	● 1/1	○ 0/1	○ 0/1	○ 0/1	● 1/1	○ 0/1
J8DIM/J8SIM	◐ 3/7	◐ 3/7	◐ 3/7	◐ 3/7	● 7/7	● 7/7	● 7/7	● 7/7	● 7/7	◐ 3/7
MR/Lamdbas	○ 1/11	○ 1/11	○ 0/11	○ 0/11	● 11/11	◐ 10/11	◐ 10/11	◐ 10/11	● 11/11	○ 1/11
JVMC	◐ 4/5	◐ 4/5	◐ 3/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5
LIB	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 1/5	◐ 1/5	◐ 1/5	◐ 1/5	◐ 2/5	○ 0/5
TR	◐ 4/9	◐ 4/9	◐ 4/9	◐ 4/9	◐ 3/9	◐ 6/9	○ 0/9	◐ 6/9	● 9/9	◐ 3/9
LRR	● 3/3	● 3/3	● 3/3	● 3/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	◐ 1/3	◐ 2/3
CSR	● 4/4	● 4/4	● 4/4	● 4/4	○ 0/4	○ 0/4	○ 0/4	○ 0/4	◐ 1/4	◐ 0/4
MH	◐ 3/9	◐ 3/9	◐ 1/9	○ 0/9	◐ 2/9	○ 0/9	○ 0/9	○ 0/9	● 9/9	◐ 1/9
CFNE	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	◐ 3/4	● 4/4	● 4/4	● 4/4
NVM	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4
Ser	◐ 1/9	◐ 1/9	○ 0/9	○ 0/9	○ 0/9	○ 0/9	○ 0/9	○ 0/9	◐ 5/9	○ 0/9
ExtSer	● 3/3	● 3/3	◐ 1/3	◐ 1/3	◐ 1/3	◐ 1/3	◐ 1/3	◐ 1/3	● 3/3	◐ 1/3
LamSer	◐ 1/2	◐ 1/2	○ 0/2	○ 0/2	○ 0/2	○ 0/2	○ 0/2	○ 0/2	◐ 1/2	○ 0/2
SPM	○ 0/7	○ 0/7	○ 0/7	○ 0/7	○ 0/7	○ 0/7	○ 0/7	○ 0/7	● 7/7	○ 0/7
SI	● 8/8	● 8/8	● 8/8	◐ 7/8	◐ 7/8	◐ 6/8	◐ 6/8	◐ 6/8	● 8/8	◐ 7/8
TYPES	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	◐ 2/6	● 6/6	● 6/6	● 6/6
Unsafe	● 7/7	● 7/7	○ 0/7	○ 0/7	● 7/7	○ 0/7	○ 0/7	○ 0/7	● 7/7	◐ 0/7
VC	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4
J9+	○ 0/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	◐ 2/3	○ 0/3
NJB	○ 0/6	○ 0/6	○ 0/6	○ 0/6	◐ 3/6	◐ 3/6	◐ 3/6	◐ 3/6	◐ 4/6	○ 0/6
sum	67/122	67/122	51/122	47/122	67/122	65/122	56/122	58/122	102/122	42/122

Table 4: Comparison of algorithms w.r.t. call graph size and runtime.

Project	#Methods		$Soot_{CHA}$		$Soot_{RTA}$		$Soot_{VTA}$		$Soot_{SPARK}$		$OPAL_{RTA}$	
	all (incl. JDK)	project	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time
jasml	160 564	265	12 184	18 s	12 134	75 s	8 012	17 s	10 356	22 s	3 195	13 s
javacc	162 484	2 185	13 035	22 s	12 986	97 s	8 863	22 s	9 752	17 s	4 222	12 s
jext	163 569	3 270	34 604	97 s	34 470	697 s	20 259	97 s	20 605	73 s	15 705	15 s
proguard	165 797	5 498	36 425	84 s	36 256	647 s	20 928	100 s	28 912	136 s	7 771	11 s
sablecc	162 670	2 371	14 138	18 s	14 088	104 s	9 687	24 s	12 101	24 s	4 932	11 s
average				47.8 s		324 s		52 s		54.4 s		12.4 s

Project	#Methods		$WALA_{RTA}$		$WALA_{0\text{-}CFA}$		$WALA_{N\text{-}CFA}$		$WALA_{0\text{-}1\text{-}CFA}$		$DOOP_{CI}$	
	all (incl. JDK)	project	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time
jasml	160 564	265	75 817	362 s	timed out		timed out		timed out		14 149	579 s
javacc	163 484	2 185	76 643	399 s	timed out		timed out		timed out		14 952	618 s
jext	163 569	3 270	79 513	411 s	timed out		timed out		timed out		27 194	1 698 s
proguard	165 797	5 498	80 240	465 s	timed out		timed out		timed out		18 205	949 s
sablecc	162 670	2 371	77 607	460 s	timed out		timed out		timed out		15 774	680 s
average				419.4 s	-		-		-			904.8 s

thus resolved to all subtypes of AbstractList. OPAL computes union and intersection types and determines that c2 can either be an ArrayList or a Vector. For this example, WALA would add four, Soot three, and OPAL two call edges on Line 4.

■ *The last observation indicates that it does not make sense to compare the results of static analyses that build upon CGs from different frameworks, even if we use the implementations of the same algorithm across frameworks.*

In summary, all frameworks support different features and exhibit different performance. OPAL is the most recent framework and supports more of the recently added Java features and APIs

than the other frameworks. Advanced features for which solutions were proposed in literature [11, 17, 39]—such as DOOP's dynamic proxy support—are not enabled by default. In addition, the performance consequences from supporting rather hard-to-support features [11, 28](e.g. context-sensitive reflection)—which are generally not precisely supportable—are not well understood.

■ *From the observations above, we conclude that it is not possible to relate a CG's feature completeness to its runtime costs and its size. A CG's suitability always needs to be analyzed in the context of a specific problem (domain).*

4.4 Experiment 3

We assess Judge's suitability for project-specific evaluations using XCorpus' Xalan project. Xalan is a mid-sized project with a well-defined main class, for which we were able to run all CG algorithms within a 90 minutes limit.[4] Xalan also uses features not handled by *any* CG implementation.

Table 5 shows an excerpt of the evaluation's results. The column *#Locations* shows whether a specific (un)used feature is prevalent in Xalan or in the JDK. Furthermore, it shows for each CG algorithm the reachable methods (*#RM*), its runtime, how many feature locations are reachable within the call graph, and whether the respective feature is supported.

Soot's CG algorithms are the only ones that handle all context-sensitive reflection in a sound manner. This resulted in the biggest call graphs whose computation also required much longer than those of WALA and OPAL.

However, all CGs contain methods that use unsupported features (○), i.e., miss edges and are thus unsound. Though, OPAL's CG reaches the least number of *sources of unsoundness*, we also observe that OPAL's CG only contains 49 (≈ 0.3%) methods from Xalan. WALA's RTA call graph in contrast touches ≈ 50% of all methods. A detailed investigation using Judge, starting from the identified sources of unsoundness, reveals that this is due to a single unsupported feature related to Java reflection. The cause is a helper method (findProviderClass(...)) in Xalan's ObjectFactory—it expects a class name as a parameter and loads the class via reflection. Soot and WALA are configured to act conservatively and, therefore, consider *all* available classes as *instantiable* when a Class.newInstance call is performed. As result, they add a call edge to *all* class' constructors which enables them to reach a large portion of methods within Xalan but also introduces a large amount of imprecision; as a manual analysis revealed.

■ *The experiment shows that even for mid-sized programs, such as Xalan, CGs contain methods that use unsupported features and are thus unsound. Unsupported features can have a devastating effect as OPAL's poor coverage of Xalan demonstrates.*

4.5 Experiment 4

The experiments so far investigated the level of unsoundness of CGs due to incomplete feature/API coverage by CG construction algorithms. Whether unsoundness is tolerable or not depends on the use case. In this experiment, we consider use cases, where unsoundness cannot be tolerated, or, at least, needs to be minimized. An example for such a use case is vulnerability analysis. To cover such use cases, OPAL provides a mechanism for manually specifying entry points that are taken into consideration by the call graph algorithm. This mechanism can be used together with Judge, which provides assistance with analysing reachable methods that use unsupported features/APIs to understand the expected effect on the CG.

The goal of the experiment is to get an intuition of the effort needed to manually turn an unsound CG to a *reasonably sound* one. The subject was Xalan's CG produced by OPAL_{RTA}, which is unsound due to incomplete coverage of the reflection API. OPAL_{RTA} is used as it is most feature complete (cf. Table 3), hence, we expect to minimize the manual effort. What *reasonably sound* means depends in general on the use case. In this experiment, we consider a CG as being reasonably sound if it contains at least all results also found by dynamic analyses. We perform two dynamic analyses: (a) JVM profiling to log which methods are executed and (b) the dynamic analysis tool Tamiflex [11] for resolving reflective calls to record dynamic edges. Whereas we use the JVM profiling to check whether all executed methods are reachable in the CG, we use Tamiflex to examine whether the CG includes all reflective call edges that have been reported. Hence, when the CG contains both, we consider it reasonably sound. We profile Xalan using exemplary input and Tamiflex to record call targets of reflective calls and then iteratively use Judge along with OPAL_{RTA}'s mechanism to configure additional entry-point methods and types that must be considered as instantiated by the CG algorithm[5]. This way, we increase soundness manually step by step.

The initial CG covered 30% of all methods reported by a profiling run using exemplary input. None of the methods reported by TamiFlex were included. The analysis took ≈ 1,5 hours and required to analyze 10 reflective call sites, configure 17 types as instantiable, and configuring 50 additional entry points. As a result, the CG covered 121 of 198 methods reported by TamiFlex. The remaining methods are related to code that is generated at runtime. Furthermore, the CG covered 1500 of 1653 methods (≈ 91%) when compared to the profile run; all non-reachable methods belong to the JDK. At this state, we consider the CG reasonably sound.

■ *The experiment indicates that the effort involved in manually increasing the soundness of CGs is high even for mid-sized projects and despite good tool support, i.e., manual correction is not proper compensation for better algorithms that automatically construct sound CGs.*

4.6 Discussion

In the following, we summarize the implications of our study for both developers of CG implementations and static analysis researchers that use the latter. Thereby we highlight, how Judge helps them to make more informed decisions, to reason about potential limitations of their tools and the root causes thereof, and

[4] Please recall that we configured WALA to exclude several packages such that its algorithms terminate.

[5] The configuration of instantiated types is required since we are using a RTA CG which does not capture reflectively instantiated types.

Table 5: Excerpt from the Project-specific Evaluation for Xalan.

Feature	#Locations		$Soot_{RTA}$		$Soot_{VTA}$		$Soot_{SPARK}$		$WALA_{RTA}$		$WALA_{0\text{-}CFA}$		$OPAL_{RTA}$		$DOOP_{CI}$	
	#M	#M	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time
	16 389	251 239	58 560	2320s	28 248	322s	23 753	139s	15 343	15s	3 021	4s	6 834	22s	14 392	988s
	Xalan	JDK	#RF	#FS	#RF	#FS	#RF	#FS	#RF	#FS	#RF	#FS	#RF	#FS	#RF	#FS
TR2	28	288	25	○	10	○	7	●	7	○	1	●	1	●	2	○
Ser3	1	97	1	○	0	○	0	○	0	○	0	○	0	○	0	○
LRR1	2	84	15	●	10	●	10	●	2	○	1	○	1	●	11	●
CSR1	38	176	49	●	34	●	31	●	20	○	6	○	4	○	7	○
JVMC4	2	23	4	○	3	○	5	●	2	○	0	○	0	●	0	○
J8PC1	81	9799	1165	○	450	○	396	○	236	●	42	●	221	●	316	○
Lambda1	0	621	30	○	14	○	14	○	5	●	0	●	3	●	1	○

M=methods; RM=reachable methods; RF=reachable features; FS=feature support; ● indicates feature support and ○ an unsupported feature;

to set up empirical evaluations and ensure reproducibility of their results.

Implications for Framework Developers. Obviously, our experiments indicate that research on constructing high quality and practically useful CGs is still needed. We need new implementations that soundly cover features that are prevalent in real software, e.g., Java 8 call semantics. Furthermore, the implementations should support users in manually adjusting implementations and/or CGs, e.g., to integrate manually-defined parts of the graph in a project-specific way to handle encounters of unsupported features. Such a mechanism can help increase the soundness. So, users can specify call edges that solve the most significant soundness/precision issues.

Judge and our comprehensive test suite can be useful for implementors of CG construction algorithms in several ways. It helps figuring out where manual adjustments of the CG are needed. When implementing new or extending existing CG algorithms it helps investigating the usage of unsupported features/APIs in practice. Judge can also help to create representative benchmark suites w.r.t. their used API/language features, which enables well-founded research that (in)directly relies on CG algorithms. What is still missing and needed, however, is support for understanding design decisions pertaining to precision. It is, in any case, important that every CG algorithm implementation documents its design decisions w.r.t. to approximations and optimizations. Finally, given that the JVM, the Java language, and its bytecode keep evolving, our comprehensive test suite can be very useful as a regression test suite, which can be continuously enriched with new test cases for further domains/APIs that effect a CG's soundness by us or by users.

Implications for Static Analysis Researchers. The results of our study directly inform developers of client analyses if a framework suits their specific needs. Soot and DOOP can be used to analyze Android code as their feature profiles match well the feature profile of this domain, while OPAL and WALA support analyses targeting Java 8 applications. In any case, researchers developing new static analysis tools/frameworks, should clearly specify the employed CG implementation, in order to increase reproducibility of their results.

Judge is useful for static analysis researchers, too. It can be used to systematically evaluate CG implementations w.r.t. their suitability to serve as a foundations for building analyses for a certain application (class), as it can provide an overview of the (prevalence of) features that are used in that application (class), so as to pick the most sound CG for the specific needs. Even OPAL's broad feature/API support may be insufficient, if unsupported features, e.g., *CSR*, are used in the target applications. Knowing where the CG is unsound enables static analysis writers to understand whether a false negative originates from an unsound CG or is a problem of the analysis.

4.7 Threats to Validity

Internal threats to validity are the usage of incorrect test cases and/or Hermes queries. In that case, we may fail to identify the presence of language features/APIs that potentially cause unsoundness. To mitigate this threat, we thoroughly reviewed all our test cases and added a built-in verifier that checks if a test case is correctly annotated. In addition, the test cases and queries were developed by researchers with many years of experience in doing Java-based static analyses and were cross-checked by two further authors. A related threat is that we missed language and API features. To mitigate this threat, the implementations of the analyzed frameworks were studied carefully w.r.t. supported features. Furthermore, one author was responsible for constructing Java CG algorithms as part of his professional career and a second author has developed Java bytecode analyses for more than 15 years. Hence, the likelihood that we missed features is low.

An external threat is the usage of a non-representative corpus of programs. Our study has shown that an established corpus such as XCorpus is not representative for modern Java code, as it does not contain usages for many relevant features. Other established corpora, e.g., Qualitas, DaCapo, etc. are even older than the XCorpus. Therefore, we used 7 different corpora of reasonable sizes.

5 RELATED WORK

5.1 Call Graph Comparison

In earlier work [34, 35] we proposed an approach to compare CG implementations w.r.t. their support for features that affect the

soundness of the computed CGs. We reuse and extend our basic infrastructure to facilitate the described experiments. Moreover, compared to our previous work, we make multiple contributions. First, we make an extensive study of Java features that affect the soundness of CGs and their real-world prevalence—as asked by the Soundiness manifesto [28]. Second, we develop a set of 15 static analyses deriving 107 features relevant to sound CG construction, which enable building representative evaluation corpora and facilitate further studies. Third, we provide a more comprehensive comparison of four major Java static analysis frameworks. Fourth, with Judge, we provide a toolchain for analyzing CG algorithms for Java on arbitrary projects. At last, we contribute a tool-supported approach to reduce the unsoundness of a CG constructed by an unsound algorithm and an experience report of the costs to manually get closer to a sound CG.

Sui et al. [41] also compared Soot, WALA, and DOOP's CG implementations using a micro benchmark suite. They measure the recall and also the precision of the tested algorithms. We consolidate their benchmark suite with our test suite. However, neither was their goal to identify sources of unsoundness nor did they do related studies.

Murphy et al. [30] conducted a study where they compared CG generators for C. They found that CGs emitted by different tools vary for identical input programs and deem the barely understood practical effects of approximations as the problem's origin. Furthermore, they discuss how one should chose a CG generator and recommend to check its input constraints, its documented or implicitly made design decision, and its correctness w.r.t. one's needs. However, such information is generally not available and our approach is a significant step towards deriving such information automatically. Whereas they conducted a one-time empirical study, Judge supports the assessment of a CG's capabilities and provides project-specific information to enable an informed decision which CG algorithm to use.

Lai et al. [23] discussed CG construction for different kinds of Java code bases w.r.t. potential sources of unsoundness and precision. However, they solely focused on programs compiled from JVM-hosted languages such as OCAML, Jython, Scheme, Scala, or JRuby. They aim to describe the challenges that arise when constructing CGs for such programs and only used WALA for their analysis. For their study, they focused on minimal, artificial code examples and the identified sources of unsoundness were reflective calls and invokedynamic usage. Our study instead focuses on the frequency with which features that potentially affect a CG's soundness are used in real-world programs.

Lhoták [25] presented a tool that enables a manual, qualitative comparison between two CGs by first finding differences and then inspecting them. Whereas Lhoták's work is targeted towards debugging CG implementations, we compare supported features and APIs as well as their relevance with regard to a particular project. Also, a systematic identification of sources of unsoundness would not be possible if the compared CGs both miss some edges; in that case the graphs would be identical.

Other works presented CG algorithms or algorithm families [5, 6, 19, 36, 45], evaluated and compared them w.r.t their size, number of reachable methods, poly- and monomorphic call sites, and runtime. Unlike ours, their CG comparison solely focuses on the sizes of the CGs and their capabilities to resolve polymorphic calls and, therefore, on their precision. Complementary to their work, we enable an automated assessment of CGs w.r.t. their supported features and APIs and whether their implementation is suitable to be used on a specific project.

5.2 Benchmarking & Testing

Corpora, such DaCapo [10], the Qualitas Corpus [43, 44], or the XCorpus [13] are regularly used to evaluate static analyses such as CG algorithms or points-to analyses on real-world applications. However, the selection of programs that are added to such corpora is often guided by the *perceived* value of the projects or by technical factors such as *compilability* which are not principle-based approaches. This easily leads to corpora with questionable representativeness and, therefore, to subsequent research results that are most likely skewed. Our evaluation shows, e.g., that the programs from the most recent corpus, the XCorpus, use less features than the current top 50 Maven libraries.

Furthermore, micro benchmark suites like SecuriBench Micro [27], DroidBench [7], or PointerBench [40] also provide one unit test per feature/program construct and those tests can be used to ensure an implementation's correctness; if a test fails the developer can easily identify the reason why and which code was responsible. However, none of the test suites targets the identification of bugs/sources of unsoundness in CG algorithms as done by ours.

Nguyen Quang Do et al. [14] presented an automatic benchmark management system (ABM) for the generation of updatable corpora. After a developer specifies a query, a corpus is automatically mined from software repositories (e.g. GitHub). ABM does not assess the mined projects any further but features an integration with Hermes. Our queries could then be used to create a benchmark suite that covers relevant projects when constructing CGs. For example, if a specialized benchmark suite, e.g., to evaluate approaches that target reflective method invocations like [9, 11, 39] should be created, the respective queries can be used.

6 CONCLUSION

In this paper, we presented Judge for (1) the evaluation of language features and APIs that are relevant when building CG algorithms; (2) comparing CG algorithms; (3) evaluating how well-suited a specific algorithm is for a specific project kind, and (4) to facilitate the creation of project-specific sound CGs. Additionally, we performed extensive studies regarding the capabilities of four major Java static analysis frameworks and the prevalence of features that are not soundly handled. The results are discouraging. All frameworks lack support for many features frequently found in the wild and—even for standard mono-/polymorphic calls—produce vastly different CGs. This renders comparisons of static analyses which rely on CGs impossible and also considerably unsound.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported by the DFG as part of CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

REFERENCES

[1] [Online; accessed 20-January-2019]. Doop's commit id: cdc59ce71d6510198da396cf6a7d20d73c6466d9.

[2] [Online; accessed 20-January-2019]. OPAL's commit id: 3107c45c8a00de0e132691a6275d39b5a4aa415b.

[3] Karim Ali and Ondřej Lhoták. 2012. Application-only call graph construction. In *European Conference on Object-Oriented Programming*. Springer, 688–712.

[4] Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*. Springer, 378–400.

[5] Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2014. Constructing call graphs of Scala programs. In *European Conference on Object-Oriented Programming*. Springer, 54–79.

[6] Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2015. Type-Based Call Graph Construction Algorithms for Scala. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 9 (Dec. 2015), 43 pages.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[8] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. *ACM Sigplan Notices* 31, 10 (1996), 324–341.

[9] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. 2015. Static analysis of implicit control flow: Resolving Java reflection and Android intents (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 669–679.

[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 *ACM Sigplan Notices*, Vol. 41. ACM, 169–190.

[11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezin. 2011. Taming reflection. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. ACM Press, New York, New York, USA, 241.

[12] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.

[13] JB Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. XCorpus–An executable Corpus of Java Programs. (2017).

[14] Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. 2016. Toward an automated benchmark management system. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 13–17.

[15] Michael Eichberg, F Kübler, D Helm, M Reif, G Salvaneschi, and M Mezini. 2018. Lattice Based Modularization of Static Analyses. In *ISSTA Companion/ECOOP Companion*. ACM.

[16] Kotlin Foundation. [Online; accessed 24-August-2018]. Kotlin Language. https://kotlinlang.org/.

[17] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static Analysis of Java Dynamic Proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 209–220.

[18] J Gosling, B Joy, G Steele, G Bracha, A Buckely, and D Smith. 2018. *The Java Virtual Machine Specification, 2018*. Oracle Ameria.

[19] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746.

[20] Rich Hickey. [Online; accessed 24-August-2018]. *Clojure Language*.

[21] IBM. [Online; accessed 19-April-2018]. WALA - Static Analysis Framework for Java. http://wala.sourceforge.net/.

[22] Artima Inc. [Online; accessed 24-August-2018]. ScalaTest. http://www.scalatest.org/.

[23] Xiaoni Lai, Zhaoyi Luo, Karim Ali, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2015. *Evaluating Call Graph Construction for JVM-hosted Language Implementations*. Technical Report CS-2015-03. University of Waterloo, David R. Cheriton School of Computer Science.

[24] École Polytechnique Fédérale Lausanne. [Online; accessed 24-August-2018]. Scala Language. https://www.scala-lang.org/.

[25] Ondvrej Lhoták. 2007. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '07*. ACM Press, New York, New York, USA, 37–42.

[26] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *International Conference on Compiler Construction*. Springer, 153–169.

[27] B. Livshits. [Online; accessed -August-2018]. SecuriBench Micro. https://suif.stanford.edu/~livshits/work/securibench-micro/.

[28] 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. *Commun. ACM* 58, 2 (2015), 44–46.

[29] 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 *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 695–710.

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

[31] MvnRepository. [Online; accessed 15-July-2018]. Maven - Popular Projects.a. https://mvnrepository.com/popular.

[32] The Apache Groovy project. [Online; accessed 24-August-2018]. Groovy Language. http://groovy-lang.org/.

[33] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. 2016. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 474–486.

[34] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. 2017. Hermes: assessment and creation of effective test corpora. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 43–48.

[35] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM, 107–112.

[36] Olin Shivers. 1988. Control flow analysis in scheme. In *ACM SIGPLAN Notices*, Vol. 23. ACM, 164–174.

[37] Yannis Smaragdakis. [Online; accessed 23-August-2018]. DOOP - Framework for Java Pointer and Taint Analysis. https://bitbucket.org/yanniss/doop/.

[38] Yannis Smaragdakis. [Online; accessed 23-August-2018]. DOOP Benchmarks. https://bitbucket.org/yanniss/doop-benchmarks/.

[39] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *Asian Symposium on Programming Languages and Systems*. Springer, 485–503.

[40] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:26.

[41] 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 *Asian Symposium on Programming Languages and Systems*. Springer, 69–88.

[42] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. *Practical virtual method call resolution for Java*. Vol. 35. ACM.

[43] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 336–345.

[44] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S Bigonha. 2013. Qualitas. class Corpus: A compiled version of the Qualitas Corpus. *ACM SIGSOFT Software Engineering Notes* 38, 5 (2013), 1–4.

[45] Frank Tip and Jens Palsberg. 2000. *Scalable propagation-based call graph construction algorithms*. Vol. 35. ACM.

[46] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.