Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java

Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini
Software Technology Group
Technische Universität Darmstadt
Germany
<reif,kuebler,eichberg,mezini>@cs.tu-darmstadt.de

Abstract

Call graphs are at the core of many static analyses ranging from the detection of unused methods to advanced control- and data-flow analyses. Therefore, a comprehensive understanding of the precision and recall of the respective graphs is crucial to enable an assessment which call-graph construction algorithms are suited in which analysis scenario. For example, malware is often obfuscated and tries to hide its intent by using Reflection. Call graphs that do not represent reflective method calls are, therefore, of limited use when analyzing such apps.

In general, the precision is well understood, but the recall is not, i.e., in which cases a call graph will not contain any call edges. In this paper, we discuss the design of a comprehensive test suite that enables us to compute a fingerprint of the *unsoundness* of the respective call-graph construction algorithms. This suite also enables us to make a comparative evaluation of static analysis frameworks. Comparing Soot and WALA shows that WALA currently has better support for new Java 8 features and also for Java Reflection. However, in some cases both fail to include expected edges.

Keywords call-graph construction, static analysis, soundiness

1 Introduction

Call graphs are a central data-structure required by many static analyses which range from the detection of unused methods [9] to advanced control- and data-flow analyses [3]. Hence, the algorithm used for constructing a call graph directly impacts a client analysis' results; often even to a larger extent than expected. A study, done by Murphy et al. [17], which compared different call graph extraction tools for the C language, revealed that the produced call graphs vary in more dimensions than they expected. Therefore, a comprehensive and competitive evaluation of existing call-graph algorithm implementations for Java is required to make it possible to assess which implementation is best suited for which specific type of project or analysis use case.

Native language features, such as the resolution of (non-)virtual method invocations, are in most cases very well supported [4, 7, 22]; they are the primary target of call-graph implementations. In contrast, the handling of core APIs (e.g., Unsafe, Reflection, or Serialization) as well as the support for special runtime events is often just an afterthought. Supporting such features is possible (reflection is discussed in [6, 15]; native methods in [13, 20]), but is often only addressed partially and limited to specific contexts [1, 2, 14]. Furthermore, as discussed by Reif et al. [18], a call graph's soundness also depends on the kind of the analyzed project, e.g., (extensible) library or closed command-line application, and how well it is supported by the algorithm.

The choice of the algorithm and the quality of its implementation therefore predetermines an analysis' precision, and recall (sometimes the term soundiness [14] is used in this context). Given that the precision is generally well understood, we will focus on the recall of existing call-graph implementations. I.e., we will focus on the calls that may be missed due to insufficient support for selected language features or core APIs. Getting a comprehensive understanding of the recall is important for several analyses – in particular for security focused ones. E.g., applications often hide their malicious intent by using Reflection and algorithms without appropriate support are therefore not suitable for analyzing such apps [10].

We propose a call-graph assessment suite for the systematic evaluation of call-graph algorithms. This suite can be used to test call-graph implementations in particular w.r.t. missing call edges due to unsupported features or programming bugs. We also used it to evaluate the two most widely used Java-analaysis frameworks Soot [23] and WALA [11] to gain a better understanding of their support for new(er) language features, selected core APIs, and their overall quality. The test suite systematically tests if call sites are resolved such that those methods, that will be executed at runtime, are actually part of the call graph; evaluating the precision is not in focus. The results show that standard language features are – as expected – very well supported. The support for core APIs instead, varies significantly and – surprisingly – also between the call-graph implementations belonging to the same framework. In some cases supposedly more precise algorithms seem to miss more edges than simpler ones.

ISSTA Companion/ECOOP Companion'18 , July 16–21, 2018, Amsterdam, Netherlands
2018. ACM ISBN 978-1-4503-5939-9/18/07...$15.00
https://doi.org/10.1145/3236454.3236503

In the next section 2 we will discuss the approach. After that, in section 3, we will present the language features and APIs covered by the proposed test suite. Our Study is presented in section 4. The paper concludes with a discussion of related work (section 5) and a conclusion (section 6).

2 Approach

The core idea is to have a wide range of small, focused test fixtures that – as far as possible – test a single relevant aspect related to call-graph construction. These test fixtures provide the *ground truth* and are used as input for the different call-graph algorithms.

Figure 1 provides an overview of the proposed approach. For each set of closely related test cases we use a single markdown (`.md`) file which contains all related tests (*<Test Fixtures Category>.md*). For example, we create one markdown file for each of the following categories: usages of Java Reflection, Java 8 language features, usages of `sun.misc.Unsafe`, or Serialization. Using markdown enables us to generate a concise, human-readable description of the test cases that also contains additional background information. Each test case consists of a small runnable Java program which uses a specific language feature and/or API along with a brief description of the unique features of the test case. Additionally, each test case contains one or more annotations to describe the expected call targets; i.e., to specify the *ground truth*.

The *Test Cases Extractor* parses the markdown files and retrieves the test cases, compiles them and bundles each one into a respective `.jar` file. After that, we use a *Framework Specific Test Adapter* to construct a call graph for each individual call-graph implementation. After construction, the graph is serialized to a common JSON-based representation. The last step is then performed by the *Call Graph Matcher*. It loads the call graph and compares the found call targets with those explicitly specified in the test cases. A short *Report* summarizes the results. Next, we provide more details regarding the individual steps.

Test Case Specification. Each markdown file is structured in the same way: The first level header (e.g., *Trivial Reflection* in Listing 1) identifies the test suite. A second level header (e.g., `TR1` in Listing 1) identifies a concrete test case. After the second level header comes the specification of the main class and a short description that is followed by multiple code snippets which – taken together – form an executable Java program. The first line of each test case is a Java comment that identifies the target file in which the code will be stored. In Listing 1 the test case `TR1` will be stored in the file `tr1/Foo.java`.

Annotating the Ground Truth. In order to detect missing call edges, a specification of the ground truth is required. We decided to use Java's annotations (cf. Line 13 in Listing 1) to specify the crucial call-graph edges that should be part

````
1   #TrivialReflection
2   The strings are directly available...
3   ##TR1
4   [//]: # (MAIN: tr1.Foo)
5   Test reflection with respect to static methods.
6   ```java
7   // tr1/Foo.java
8   package tr1;
9   import lib.annotations.callgraph.IndirectCall;
10  class Foo {
11      static String m() { return "Foo"; }
12
13      @IndirectCall(
14          name = "m", returnType = String.class,
15          line = 17, resolvedTargets = "Ltr1/Foo;" )
16      public static void main(String[] args) throws Exception {
17          Foo.class.getDeclaredMethod("m").invoke(null);
18  }}
19  ```
20  [//]: # (END)
21  ##TR2 ...
````

Listing 1. Reflection.md

of the call graph. Due to the decision, that all code snippets have to be executable programs it is sometimes necessary to perform multiple calls to achieve the required state. Hence, each method may contain multiple call sites. Therefore, we identify the relevant call sites using line numbers, the callee's name, as well as its return and parameter types. Currently, to avoid ambiguous call sites the fixtures are restricted to have only one method call with the same name per line of code.

We provide two annotations: First `CallSite` to specify direct call edges between two methods. This one is used for standard virtual method calls, constructor calls, static method invocations, and default method invocations (Java 8). The second one, `IndirectCall`, is used to specify indirect calls. Consider the reflective call m.invoke(**null**) in Line 17 (Listing 1). In this case the call graph may (also) contain call edges to the Reflection API and/or a call to the target method (`m` in the example); however, the representation of such calls is framework specific and to abstract over differences, e.g., how reflective calls, method references etc. are actually handled by the frameworks, we specify that we expect some path leading to the expected targets as shown in Lines 13-15.

Serialization of the call graphs. In the JSON representation (cf. Listing 2) of the call graphs each method is represented using its name, the parameter types, the return type, and the fully qualified name of its declaring class. A call site is represented by the caller method, the line number, the declared target method, and the set of computed target methods.

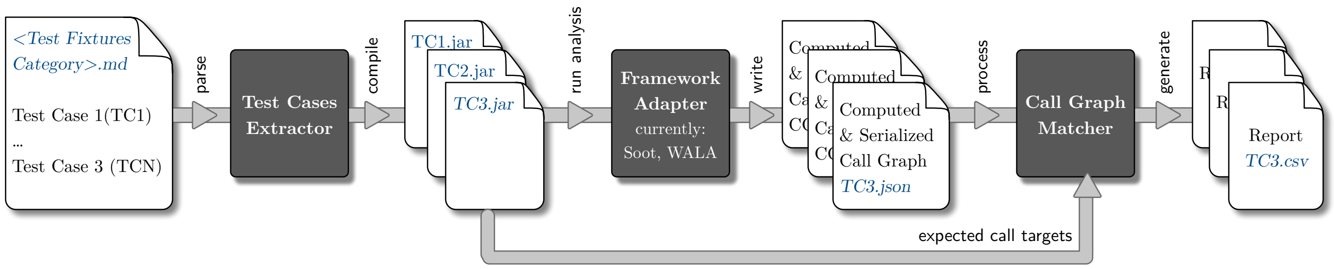

Figure 1. Call graph test cases specification, compilation, evaluation, and summarization.

```
{ "callSites": [
    { "declaredTarget": {
        "name": "getDeclaredMethod",
        "parameterTypes":
            ["Ljava/lang/String;","[Ljava/lang/Class;"],
        "returnType": "Ljava/lang/reflect/Method;",
        "declaringClass": "Ljava/lang/Class;" },
      "method": {
        "name": "main",
        "parameterTypes": ["[Ljava/lang/String;"],
        "returnType": "V",
        "declaringClass": "Ltr1/Foo;" },
      "line": 12,
      "targets": [ {
          "name": "getDeclaredMethod",
          "parameterTypes":
              ["Ljava/lang/String;","[Ljava/lang/Class;"],
          "returnType": "Ljava/lang/reflect/Method;",
          "declaringClass": "Ljava/lang/Class;" } ]
    },
    ...
  ]
}
```

Listing 2. Serialized Call Graph

Validating the Call Graph. Identifying missing call edges is done by iterating over all methods of a project and comparing the found call targets against the specified one. The presence of call edges related to indirect calls is done by performing a breadth-first search on the computed call graph; starting with the main method. The final report then lists missed calls/the failed test cases.

3 Test Suite

In the following, we present the different categories that are covered in the proposed test suite.[1]

Virtual Method Calls. At the core of Java are virtual methods. When such a method is called, the target is resolved depending on the runtime type of its receiver object. When the runtime type can not be determined precisely, a sound call-graph algorithm will over-approximate the receiver type and then determine the set of possible call targets.

Reflection. The Java Development Kit provides two APIs for reflection which allow to call *basically arbitrary* methods at runtime. Both APIs, `java.lang.reflect.*` as well as `java.lang.invoke.*` provide methods to 1) look up a method in a class by name and then 2) to invoke the method. Due to that layer of indirection and the fact, that visibility constraints can be bypassed, the resolution of reflective calls is challenging. The test cases in this category are divided in the following sub categories: *Trivial Reflection* contains test cases where all Strings are immediate parameters of the calls of the classic Reflection API (`...reflect.*`). Hence, neither control- nor data-flow analysis are required. *Trivial Modern Reflection* is similar to the previous one, but the tests cases use the Java 7 MethodHandle-API. *Locally Resolvable Reflection* defines test cases that require intra-procedural control- and data-flow analysis to resolve the respective calls. *Context-sensitive Reflection* requires inter-procedural control- and data-flow analysis to resolve the reflective calls.

Unsafe API. With `sun.misc.Unsafe` Java provides an internal API that allows direct memory manipulations from within Java code. This API is used by wide spread libraries [16]. Using the methods `compareAndSwapObject`, `putObject`, and `getObject`, objects can be put into or retrieved from fields. The test cases therefore test if the call graphs contain call edges to those virtual methods that are due to an unsafe field update. E.g., if a method m invocation occurs on a field of type `T` which is updated to an object of type `TSub` (with $TSub <: T$) via `Unsafe`, the call graph must contain an edge to $TSub.m$.

Static Initializer. In Java, every time a class is loaded by a class loader, a call to its static initializer is performed by the language runtime. Those calls are implicit and, therefore, must be explicitly modeled.

Serialization. Java's serialization mechanism allows to persistently store and retrieve objects using object serialization.

[1]The test suite is available at our repository: https://bitbucket.org/delors/jcg/.

To use this mechanism, classes must implement the interface `java.io.Serializable` or `java.io.Externalizable`. When (de-)serialization is used, the JVM potentially calls several overridable call back method(s). I.e., the call sites are not part of the Java code base.

Lambdas and Method References. Java 8 introduced two new language features: lambdas and method references. When these features are used in Java source code, they are compiled using the `invokedynamic` instruction. At runtime a so-called call site object is instantiated by the instruction and used to indirectly invoke the lambda method/the referenced method later on.

Java 8 Default Methods. Java 8 introduced default methods which are defined in interfaces and which have to be taken into account when resolving virtual method calls. We included test cases for virtual method invocations w.r.t. interface default methods and maximally-specific interface methods.

Type Narrowing. Type casts and `instanceof` checks can be performed using language features, but also using core Java APIs. We added several test cases that test both: API-based and language-feature-based type casts and `instanceof` checks.

4 Study

In the following, we describe how we evaluate Soot and WALA's call-graph implementations by applying the proposed test suite. The study is driven by the following two research questions.

RQ1 How do the call graphs of Soot and WALA compare with each other?

RQ2 What are the main sources of unsoundness in built-in call-graph implementations?

All measurements were done using WALA version 1.4.3 and Soot version 3.0.0[2]. We generated data for the following algorithms: Soot_{CHA}, Soot_{RTA}, Soot_{VTA}, $\text{Soot}_{\text{SPARK}}$, WALA_{RTA}, $\text{WALA}_{\text{0-CFA}}$, $\text{WALA}_{\text{N-CFA}}$[3], and $\text{WALA}_{\text{0-1-CFA}}$.

Table 1 summarizes the test result. The first column (Category) shows the different test categories. Columns two to nine show the individual test results for each call-graph implementation per test category. Every table cell shows a symbol and a pair of numbers where the symbol indicates whether all (●), some (◐), or no (○) tests succeeded (i.e., the expected call edge is part of the call graph). The numbers represent how many test cases succeeded compared to the total number of tests.

The results (cf. Table 1) show that basic language features like static initializers (SI), polymorphic calls (PC), and type cast (TYPES) are well supported. The only exception is a static initializer case which is covered neither by Soot nor WALA. This test case pertains a Java 8 feature where the static initializer of an interface must be called when a subclass of the interface is initialized and the respective interface defines a default method. A rather unexpected exception is the $\text{WALA}_{\text{N-CFA}}$ implementation which 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., the `instanceof` operator or type casts of the form `(String) o;`.

All call-graph implementations from Soot and WALA do not deal with serialization-related methods (SE). Those methods must be considered when object (de-)serialization occurs during call-graph construction, i.e., for instance `readObject` and `writeObject` are called by the runtime (JVM) and, therefore, must be included in the call graph.

Language features and APIs that were introduced by Java 8 are still mostly unsupported by Soot but are handled by WALA. WALA correctly resolves virtual calls (J8PC) w.r.t. Java 8 interfaces and default methods, Soot does not resolve any method invocation to an interface's default method. In contrast to WALA, Soot also lacks support for Lambdas as well as calls performed via method references (MR). The only method reference test case that WALA does not support concerns object creation.

Support for Java's reflection API varies between Soot and WALA's call graphs. Regarding the different levels of reflection usage, Table 1 shows that the support for trivial reflection (TR) is weak in Soot and better supported in WALA. An outlier is the $\text{WALA}_{\text{N-CFA}}$ implementation which does not support reflection at all.

Table 1 shows that Soot_{CHA}, Soot_{RTA}, and WALA_{RTA} are able to resolve all method calls related to Java's Unsafe API (`java.misc.Unsafe`). However, more advanced call-graph implementations are not able to detect those cases. The imprecision of cheap algorithms benefits the support of the Unsafe API.

Please note that $\text{WALA}_{\text{N-CFA}}$ implementation performs consistently worse than WALA's other implementation.

RQ1 - How do the call-graph implementations from Soot and WALA compare? Soot's call graphs support less Java language features and core APIs than WALA's call graphs. In particular, the language support for Java 8 is still completely missing in Soot. When newer Java versions (Java 8 or higher) are analyzed, WALA's call graphs, except the $\text{WALA}_{\text{N-CFA}}$ algorithm, are better suited. However, when analyzing older Java version (prior Java 8) Soot is a viable option. Another distinguishing feature is the resolution of reflective method calls where WALA's algorithms are able to resolve more test cases than Soot's call-graph implementations.

RQ2 - What is the main source of unsoundness in built-in call-graph implementations? All built-in call graphs, those from Soot and WALA, struggle with resolution of reflective method calls. Another unsoundness source pertaining

[2]It is a snapshot from Soot's nightly build: 16th April 2018 - 5:26 pm.
[3]We use N=1 throughout the whole evaluation.

Table 1. Support of language features and core APIs of Soot and WALA's call graphs.

Category	Soot$_{\text{CHA}}$	Soot$_{\text{RTA}}$	Soot$_{\text{VTA}}$	Soot$_{\text{SPARK}}$	WALA$_{\text{RTA}}$	WALA$_{\text{0-CFA}}$	WALA$_{\text{N-CFA}}$	WALA$_{\text{0-1-CFA}}$
SI	◐ 5/6	◐ 5/6	◐ 5/6	◐ 5/6	◐ 5/6	◐ 5/6	◐ 5/6	◐ 5/6
PC	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6
J8PC	◐ 3/6	◐ 3/6	◐ 3/6	◐ 3/6	● 6/6	● 6/6	● 6/6	● 6/6
Lamdbas	○ 0/5	○ 0/5	○ 0/5	○ 0/5	● 5/5	● 5/5	● 5/5	● 5/5
MR	○ 0/8	○ 0/8	○ 0/8	○ 0/8	◐ 7/8	◐ 7/8	◐ 7/8	◐ 7/8
TR	◐ 3/10	◐ 3/10	◐ 1/10	◐ 1/10	◐ 5/10	◐ 6/10	○ 0/10	◐ 6/10
LRR	○ 0/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	◐ 1/3	○ 0/3	◐ 1/3
CSR	◐ 2/6	◐ 1/6	◐ 1/6	◐ 1/6	○ 0/6	◐ 2/6	○ 0/6	◐ 2/6
TMR	○ 0/3	○ 0/3	○ 0/3	○ 0/3	◐ 1/3	○ 0/3	○ 0/3	○ 0/3
UNSAFE	● 3/3	● 3/3	○ 0/3	○ 0/3	● 3/3	○ 0/3	○ 0/3	○ 0/3
TYPES	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	◐ 2/6	● 6/6
SE	○ 0/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3	○ 0/3

Explanation of category abbreviations: SI = static initializer; PC = polymorphic calls; J8PC = Java 8 polymorphic calls; Lambdas = lambdas; MR = method references; TR = trivial reflection; LLR = locally resolvable reflection; CSR = context-sensitive reflection; TMR = trivial modern reflection (`java.lang.invoke.MethodHandle`); UNSAFE = `java.misc.Unsafe` API; TYPES = type cast API; SE = serialization;

to Soot's call graphs is the introduction of new language features. Several features that were introduced in Java 8[4] are not yet supported by Soot. Additionally, corner cases, e.g., object creation via method references or static initializers of interfaces, hinder the sound and correct implementation of call graphs. This shows the need for a call graph assessment suite to comprehend a call-graph implementation's strengths and weaknesses and provide a test suite for implementors.

4.1 Threats to Validity

The performed evaluation demonstrates the design and usefulness of a comprehensive test suite to assess sources of *unsoundness* in call graph implementations. The test suite is, however, not complete w.r.t. to all Java features, core APIs, or runtime (JVM) callbacks. For instance, test cases for JNI calls, Java 9 modules, class loading, and others are missing. Also other scenarios, such as the analysis of partial programs or software libraries are not discussed. However, the test suite already covers language features and APIs that are used in practice and, therefore, allows us to draw valid conclusions regarding the tested features and core APIs.

Since all test cases are manually annotated, there is a chance of annotation mistakes. To mitigate this risk, the programs were executed and all annotations and unexpected results were independently verified by two authors.

5 Related Work

Testing and benchmarking of static analyses to ensure correctness was always a concern of the program analysis community. While testing ensures correctness and benchmarking tries to establish a common baseline for a meaningful comparison–which mostly concerns the analysis' precision–our work targets the recall of call-graph implementations which are a fundamental building block for inter-procedural analyses. Benchmarks and corpora consisting of real-world applications, such as the DaCapo [5] benchmark suite, the Qualitas Corpus [21], or the XCorpus [8], are too big and, therefore, cannot be used to comprehend an implementation's unsoundness. Artificial benchmarks or test suites like DroidBench [3] provide test cases w.r.t. different analysis features and language-specifics. However, those test cases are engineered to target the client analysis' result and are not suitable to uncover flaws in call-graph implementations.

The empirical study conducted by Murphy et al. [17] compared several different call-graph extractors for the C language and found that the extracted call graphs varied in more different dimensions across the tools than expected. Given their results, they pointed out that the design space for call graphs raises several problems because the practical effect of approximations is not well understood. In their empirical study they focus on the comparison of complete call graphs, our approach – in contrast – assesses individual language features or APIs supported by a single call graph. Moreover, we do not only provide empirical results but also publish a test suit that can be used to implement soundy[5] call graphs.

In [12] Lhoták does a qualitative comparison between two different call graphs. He presented two tools where the first tool can be used to find differences between two given call graphs and the second tool can be used to inspect those differences. While those tools aim to comprehend the difference between two call graphs, for instance to examine a call graph's correctness or precision, our approach solely focuses on soundness aspects and also unveals features that are not yet supported by any algorithm.

[4]Java 8 was first released in March 2014.

[5]The knowledge about a implementation's weaknesses enables a reasoning about a client analysis' potential false negatives.

6 Conclusion & Future Work

In this paper, we discussed the design of a comprehensive and extensible call-graph assessment suite that enabled us to approximate the unsoundness for several call graphs computed by two famous frameworks: Soot and WALA. The evaluation revealed the weaknesses and strengths of Soot and WALA's built-in call graphs and that both vary significantly.

In the future, we are going to use Hermes [19] to quantify the influence of a missing feature on real-world applications and, therefore, enhance the understanding of the importance of Java language features, APIs, and runtime environment. Additionally, we will increase the number of test categories, test cases, and evaluated static analysis frameworks.

Acknowledgments

This work was supported by: the DFG as part of CRC 1119 CROSSING, the German Federal Ministry of Education and Research (BMBF), and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

References

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

[2] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. 2015. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 13–18.

[3] 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.

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

[5] 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.

[6] 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. https://doi.org/10.1145/1985793.1985827

[7] 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.

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

[9] Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. 2015. Hidden Truths in Dead Software Paths. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 474–484. https://doi.org/10.1145/2786805.2786865

[10] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 281–294.

[11] IBM. [n. d.]. WALA - Static Analysis Framework for Java. http://wala.sourceforge.net/. ([n. d.]). [Online; accessed 19-APRIL-2018].

[12] OndâĹĪrej 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. https://doi.org/10.1145/1251535.1251542

[13] Siliang Li and Gang Tan. 2009. Finding bugs in exceptional situations of JNI programs. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 442–452.

[14] 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.

[15] Benjamin Livshits, John Whaley, and Monica S Lam. 2005. Reflection analysis for Java. In *Asian Symposium on Programming Languages and Systems*. Springer, 139–160.

[16] 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. https://doi.org/10.1145/2814270.2814313

[17] 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.

[18] 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.

[19] 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.

[20] Gang Tan, Andrew W Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. 2006. Safe Java native interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, Vol. 97. 106.

[21] 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.

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

[23] 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.