Total Recall? How Good Are Static Call Graphs Really?

Dominik Helm
ATHENE
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
helm@cs.tu-darmstadt.de

Sven Keidel
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany

Anemone Kampkötter
Department of Computer Science
Technische Universität Dortmund
Dortmund, Germany
anemone.kampkoetter@tu-dortmund.de

Johannes Düsing
Department of Computer Science
Technische Universität Dortmund
Dortmund, Germany
johannes.duesing@tu-dortmund.de

Tobias Roth
ATHENE
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
roth@cs.tu-darmstadt.de

Ben Hermann
Department of Computer Science
Technische Universität Dortmund
Dortmund, Germany
ben.hermann@cs.tu-dortmund.de

Mira Mezini
ATHENE
hessian.AI
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
mezini@cs.tu-darmstadt.de

ABSTRACT

Static call graphs are a fundamental building block of program analysis. However, differences in call-graph construction and the use of specific language features can yield unsoundness and imprecision. Call-graph analyses are evaluated using measures of precision and recall, but this is hard when a ground truth for real-world programs is generally unobtainable.

In this work, we propose to use carefully constructed dynamic baselines based on fixed entry points and input corpora. The creation of this dynamic baseline is posed as an approximation of the ground truth—an optimization problem. We use manual extension and coverage-guided fuzzing for creating suitable input corpora.

With these dynamic baselines, we study call-graph quality of multiple algorithms and implementations using four real-world Java programs. We find that our methodology provides valuable insights into call-graph quality and how to measure it. With this work, we provide a novel methodology to advance the field of static program analysis as we assess the computation of one of its core data structures—the call graph.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; *Dynamic analysis*; • **Theory of computation** → *Program analysis*.

KEYWORDS

Call Graph, Static Analysis, Dynamic Analysis, Precision, Recall

ACM Reference Format:
Dominik Helm, Sven Keidel, Anemone Kampkötter, Johannes Düsing, Tobias Roth, Ben Hermann, and Mira Mezini. 2024. Total Recall? How Good Are Static Call Graphs Really?. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3652114

1 INTRODUCTION

Static call graphs (CGs) are a prerequisite for all interprocedural static analyses. But state-of-the-art static CG analyses [5, 14, 15, 19, 30] suffer from *unsoundness* and *imprecision*. Since the quality of many downstream analyses depends heavily on CG analyses [4], it is important to assess the quality of CGs in terms of *precision* and *recall*. Precision is the percentage of calls in the static CG that can be executed at runtime. Recall is the percentage of calls executable at runtime that occur in the static CG. In order to measure these two metrics, a ground-truth CG is necessary. Constructing such a ground truth would entail capturing all possible program executions which is an undecidable problem [22]. In lack of a ground truth, researchers have utilized three kinds of alternative approaches.

Firstly, prior work relied on micro-benchmarks [20, 23, 26] consisting of small programs that exercise individual language features and respective hand-crafted CGs. Static CG analyses are executed on the benchmarks and their results compared against the hand-crafted ones to gauge how well language features are covered. However, such micro-benchmarks (by construction) lack insights into a CG's recall for real-world programs. A single commonly used, hard to analyze language feature like reflection might disproportionately affect the recall of static CGs of real-world programs. Secondly, prior work compared the size of CGs constructed by different algorithms [11, 25, 29]. A CG containing fewer calls than another one is typically considered more precise. Unfortunately, this differential measure fails to quantify the actual precision of CGs. A smaller CG is not necessarily more precise—the smaller size could also be a result of lower recall. Thirdly, a better approach to measure precision and recall is to approximate the ground truth by *dynamic baselines* [2, 16, 27, 28]. The program under analysis is executed and dynamic CGs are constructed from recorded execution traces, which then act as baselines for assessing static CGs.

Approaches relying on dynamic baselines involve decisions about the program entry points to execute and about the number of traces that are *enough* for a sufficiently exhaustive baseline. Some existing approaches rely on benchmark programs [1, 16], others use existing test suites [2, 28], or synthesized tests [27] to exercise the target programs. However, we lack a systematic investigation of the relationship between static CGs and dynamic baselines. Using a high-coverage dynamic baseline might yield an inaccurately low recall. This is because the construction of static CGs typically starts at specific entry points and a high-coverage dynamic baseline may contain calls not reachable from these points. Conversely, using a low-coverage dynamic baseline that lacks calls reachable from the entry points may yield inaccurately low precision.

These observations indicate that we need a systematic investigation of the relationship between static CGs and dynamic baselines to understand what makes dynamic baselines good approximations of a ground-truth CG. This is where this paper contributes to the state of the art in computing baseline CGs: We record dynamic baselines exclusively for *predetermined program entry points*, frame the creation of dynamic baselines from these points as an *optimization problem*, and *approximate the solution to this optimization problem* by systematically expanding the corpus of input data supplied to the entry points. This strategy enables us to (a) approximate a dynamic baseline from the same entry points used for computing static CGs, facilitating a more accurate comparison, and (b) make well-defined decisions about the values and number of inputs to use.

Specifically, we apply three techniques to generate inputs for the entry points: First, we assemble an input corpus from publicly available sources (e.g., test or fuzzing corpora). Second, we inspect the coverage of this corpus and manually add inputs to cover yet uncovered code that is reachable from entry points. Third, we expand the input corpus with a coverage-based fuzzer. The fuzzer generates new inputs based on existing inputs and adds them to the corpus if the execution with the new input covers new parts of the program. To our knowledge, we are the first to apply coverage-based fuzzing for this purpose.

In essence, our method allows us to construct dynamic baselines that systematically approximate the unattainable ground truth for static CGs, while staying within the theoretical bounds of reachability from the specified entry points. Notably, we show that the precision and recall metrics for a static CG measured relative to our dynamic baselines establish bounds for the hypothetical precision and recall measured against the unattainable ground-truth CG. These bounds improve as the input corpus gets expanded as described above, turning the question of good and sufficient inputs into an optimization problem.

We used our approach to benchmark eight CG analysis implementations in four static program analysis frameworks. on four distinct Java programs (`axion`, `batik`, `jasml`, and `xerces`) from *XCorpus* [9]. Our study yields two key findings: Firstly, the size of a static CG is not a reliable predictor of its quality. Secondly, theoretical characteristics of CG algorithms can be misleading and, for the same conceptual algorithm, the framework of choice may have a significant impact on the quality. Also, observations about the quality of static CGs on some programs may not generalize to other programs. As a consequence, the decision about a CG algorithm to use in a particular case should be based on actually measuring the precision and recall of specific implementations of specific algorithms, if possible from multiple frameworks, for that specific use case. This, in turn, reinforces the importance of reliable methods for measuring CG quality.

In summary, we make the following contributions:

- We systematically explore the relationship between static and dynamic CGs that approximate the ground truth for measuring precision and recall. Based on this exploration, we propose a novel method for capturing dynamic baseline CGs, which uses predetermined entry points and frames the search for an accurate baseline as an optimization problem (Section 3).
- We propose an approach for measuring precision and recall of static CGs that uses the new method for dynamic baselines (Section 4). Crucially, we propose a new method for generating an input corpus for entry points that enables to systematically explore the solution to the optimization problem.
- We use our method for constructing dynamic baselines to benchmark eight implementations of CG analyses on four Java programs from XCorpus. The study provides insights into the quality of current CG analyses and underlines the importance of well-grounded methods for measuring CG quality (Section 5).

2 STATE OF THE ART

There are three main approaches to assess CG quality: micro-benchmarks, CG size, and dynamic baselines.

Micro-Benchmarks. To study the soundness of static CGs, prior work relied on hand-crafted micro-benchmark suites [20, 23, 26]. The latter consist of small programs, crafted to exercise individual language features. The authors run static CG analyses on the benchmarks and check if the resulting CGs contain the expected calls. Such micro-benchmarks are effective at uncovering sources of unsoundness, but do not provide any indication of what the concrete recall and precision of CG analyses are. Specifically, it is unclear what impact an unsoundly or imprecisely handled language feature has on the CG of real-world programs. For example, the Python Call Graph analysis PyCG [23] passes 103 of 112 (92%) micro-benchmark tests, yet the recall measured on five real-world applications is only 70%. This demonstrates the mismatch between the number of false-negatives on a micro-benchmark and the recall measured on real-world apps for which the authors created reference CGs by hand in a time-consuming process.

Call-Graph Size. Some authors compare CGs by their relative sizes to draw conclusions on relative precision, based on the number of reachable methods or call edges: if one CG contains fewer calls than another one, it is considered more precise. Such quantitative measures are sometimes complemented by qualitative studies.

Ali and Lhoták [1] compare CG quality based on the number of call edges among other criteria, with smaller, supposedly more precise CGs considered better. Smaragdakis et al. [25] and He et al. [13] consider the number of call edges and sizes of points-to-sets to evaluate the precision of points-to analyses. Reif et al. [20] also compare the number of reachable methods to discuss CG precision when evaluating different CG construction frameworks. Lhoták [16] compares CGs based on the number of call edges and reachable methods, claiming that these are the usual features when comparing CG analysis precision and suggests supplementing such quantitative metrics with qualitative assessment. Tip and Palsberg [29] and Gutzmann et al. [12] examine different metrics about CGs, including the numbers of reachable methods and edges in the CG.

In Section 5.4, we show that CG size is not a reliable measure for CG quality.

Dynamic Baselines. Chakraborty et al. [7] measure the recall of JavaScript static CG analyses using a dynamic baseline. They manually exercise JavaScript GUI applications and record call edges, call-site targets, and reachable methods to identify dynamic language features causing unsound CGs. They use a single program execution each and recognize that this is problematic, with some code reachable only on repeated execution. Similarly, to evaluate static CGs, Luo et al. [17] consider a dynamic baseline for which they manually exercise programs to record executed methods, but not full CGs.

Both Lhoták [16] and Ali and Lhoták [1] use an executable benchmark to record a dynamic baseline. Sui et al. [27] propose recording a dynamic baseline to evaluate static CGs by using built-in and synthesized test cases from the *Xcorpus* [9] data set. Utilizing these tests, they exercise the analyzed programs to cover as many execution paths as possible, achieving a median branch coverage of ~55%. They explicitly consider only reachable methods to calculate recall. Similarly, Antal et al. [2] use a dynamic baseline constructed by executing tests to perform a comparative study of static and dynamic CG analyses, which they complement with a manual qualitative analysis to validate specific call edges. Ságodi et al. [28] also use a test-based dynamic baseline to investigate the differences between static and dynamic CGs. They argue that dynamic CGs captured via test execution cannot provide a reliable baseline because they fail to account for code segments that remain unexercised by the available tests.

Current approaches for constructing baselines all depend on a specific benchmark or test harness. We argue that using dynamic baselines constructed by running tests leads to imprecise precision and recall measurements (cf. Section 3.3). Dynamic call graphs constructed in this manner can both overestimate and underestimate the actual behavior of the program at the same time.

3 NEW METHOD FOR DYNAMIC BASELINES

We formally define the ground-truth CG and approximations thereof by dynamic baselines. On that basis, we reason about issues with approaches that approximate the ground truth via dynamic baselines captured by running tests and/or benchmarks and frame the construction of dynamic baselines as an optimization problem.

3.1 Ground Truth and Approximations

Formally, a ground truth CG is defined as follows:

Definition 3.1 (Ground-Truth CG). Let $\mathsf{Traces}(p)$ be the set of all execution traces of program p starting at entry points from a set E. The *ground-truth CG* contains all caller-callee edges in these traces:

$$\mathsf{GroundTruth}(p, E) = \left\{ \mathit{caller} \rightarrow \mathit{callee} \;\middle|\; \begin{array}{l} \mathsf{Call}(\mathit{caller}, \mathit{callee}) \in t, \\ t \in \mathsf{Traces}(p, E) \end{array} \right\}$$

Computing a ground-truth CG is undecidable, as programs can have infinitely many traces, of potentially infinite length. Thus, we need to approximate it by some baseline constructed by collecting calls from sampling dynamic program executions[1]:

Definition 3.2 (Dynamically-Sampled CG). Let Tr be a subset of all execution traces $\mathsf{Traces}(p)$ of program p starting at entry points from the set E. A *dynamically-sampled CG* of p contains all caller-callee edges in Tr:

$$\mathsf{Dynamic}(p, \mathsf{Tr}, E) = \left\{ \mathit{caller} \rightarrow \mathit{callee} \;\middle|\; \begin{array}{l} \mathsf{Call}(\mathit{caller}, \mathit{callee}) \in t, \\ t \in \mathsf{Tr} \subseteq \mathsf{Traces}(p, E) \end{array} \right\}$$

To obtain reliable measures of precision and recall, we aim for a dynamically-sampled CG that closely approximates the ground-truth CG. Closeness depends on two decisions: (a) entry points E used for the traces and (b) number of traces in Tr. In particular:

Observation 1: *Dynamically-sampled CGs should have a* high coverage *of the program parts* reachable from the entry points.

In the ground truth, a method m is considered reachable from an entry point E, if for any program execution starting at that entry point, the method is invoked at least once, i.e., $\exists t \in \mathsf{Traces}(p, E)$ s.t. $\exists \mathsf{Call}(\_, m) \in t$. In a (dynamic or static) CG computed for an entry point E, a method m is considered reachable, if m is part of that CG.

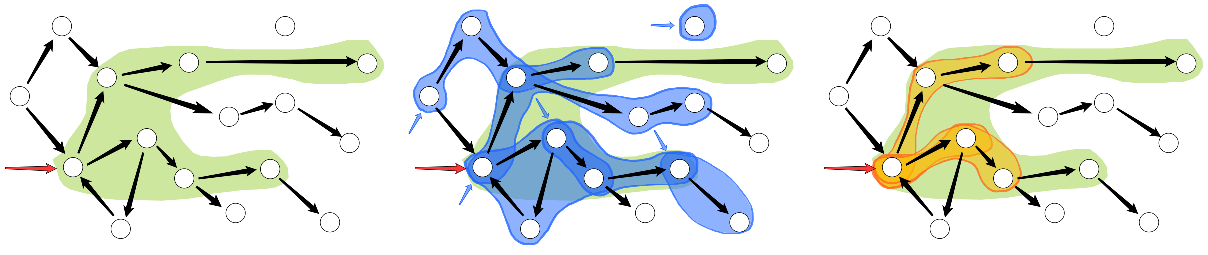
3.2 Approximation from Fixed Entry Points

When first deciding on the entry point, it is important to keep in mind that a good static CG is a good approximation of real program behavior. Typically, a program will be executed from only few entry points. These include main methods in applications [1], the public API of a library [20], or just the subset of a library's API actually used in a particular application. As entry points are application-specific, one should not choose them in arbitrary ways. This is especially important for CG algorithms like RTA and CFA, which compute object instantiations starting from the entry points. The baseline against which to assess the quality of call graphs, should thus comprise all possible program executions from these entry points, but exclude program executions not realizable from them. Static CG algorithms are often built on the assumption of a specific entry point. Disregarding this has led to problems when comparing static CGs to dynamic baselines [28].

Observation 2: *Considering proper entry points is vital to model real program behavior and assumptions of static CG analyses.*

Below, we assume a single entry point (red arrow in Figure 1) in line with static CG analyses that typically allow specifying one entry point (e.g., [29]). Extension to multiple entry points is trivially the union of execution traces starting at any of the entry points.

[1]It is infeasible to obtain this baseline by hand, because even trivial Java programs, e.g., a HelloWorld program, execute hundreds of methods [16] and more than 1000 calls.



(a) Call Graph and Ground Truth (b) Methods Covered by Tests (c) Methods Covered by Corpus

Figure 1: Ground Truth and Coverage Achieved by Testing and Our Corpus-based Approach

3.3 Approximation by Tests or Benchmarks

To generate traces, some researchers rely on existing test suites [2, 28] or additional synthesized tests [27] to exercise the target programs. Benchmarks and unit tests, however, constitute entry points that may not resemble typical use of a program. Notably, this is true for synthesized tests that test all methods of a program.

A test-based dynamic baseline $T = \text{Dynamic}(p, \text{Tr}, E')$ may have no relation to the actual ground-truth CG G from different entry points E, as Figure 1b depicts (blue arrows). Adding more test cases may not lead to more closely approximating G; on the contrary, it may lead to more dynamic call edges that are not part of G. Thus, it is unclear how unit tests should be constructed and how many should be used to get a T that is a good approximation of G.

Measuring S against a T sampled from unit tests could yield arbitrarily wrong results. In the worst case, a T from tests with high coverage metrics could cover many execution traces from unrelated entry points, but (almost) none of G. Measuring a very imprecise S against such T would report high precision, possibly close to 100%, if false-positive edges included in the static CG also appear in the dynamic sample. Conversely, measuring a high-recall S against a test-based T could result in a low recall measurement, possibly 0%: a precise S would not include edges outside G, so edges of T outside G would wrongly be considered missing, i.e., false-negative. Not much attention has been paid to this fact in previous work.

Observation 3: *Test-based dynamic baselines can yield arbitrarily wrong precision and recall measures.*

Other researchers record a single execution of a benchmark program [1, 7, 16]. A single trace, however, cannot be expected to have a high coverage of the program, some code might even only be executable on subsequent program executions [7] and missing edges render the dynamic baseline unreliable [28].

Observation 4: *A sufficiently high number of traces are needed to cover most of the actual program behavior in a dynamic baseline.*

3.4 Approximation by Input Corpus

We argue that dynamically-sampled CGs should be recorded by executing programs repeatedly from the entry point of real program executions and the same entry point as used for the static CGs.

A dynamically-sampled CG (D) captured from the entry point of the investigated static CG (S) is a subset of the unattainable ground-truth CG (G), i.e., $D \subseteq G$, as depicted in Figure 1c. However,

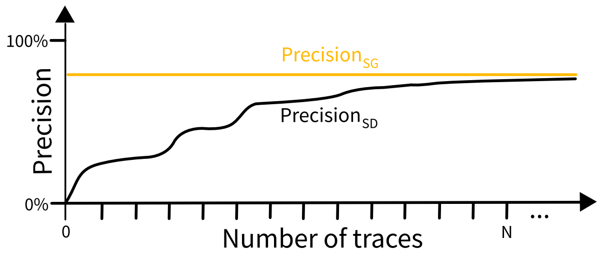


Figure 2: Precision Measure in Relation to Corpus Size

a systematic improvement of D is possible by expanding the corpus of inputs passed to the entry point with new inputs that lead to incorporating more of the previously uncovered segments of G into D (as shown in Figure 1c, where green areas transition to orange). Consequently, accurately measuring precision and recall of S becomes a matter of optimizing D. Precision and recall metrics of S measured based on D establish bounds of S's actual precision and recall that would be measured based on G. Optimizing D tightens these bounds. To the best of our knowledge, such bounds have not been discussed in the literature before. Proofs for the theorems listed below are given in the supplementary material.

First, the precision of S measured against D, Precision$_{SD}$, gives a lower bound to the precision measured against G, Precision$_{SG}$:[2]

Theorem 3.3 (Bound on Precision). [3]

$$\text{Precision}_{\text{SD}} = \frac{|S \cap D|}{|S|} \leq \frac{|S \cap G|}{|S|} = \text{Precision}_{\text{SG}} \qquad \square$$

This bound induces an optimization problem on D, illustrated by Figure 2: The measured Precision$_{SD}$ provides a tightening bound to the unattainable Precision$_{SG}$. With a small input corpus, only few execution traces with few call edges are observed and many edges in S are classified as false-positives. As the input corpus grows, more edges will be sampled, leading to a monotonic decrease in the number of edges classified as false-positive and, thus, a monotonic increase in the measured precision:

[2]We use Precision$_{\text{XY}}$ and Recall$_{\text{XY}}$ to denote the precision/recall of X measured against Y, with G, D, and S being abbreviations for GroundTruth (p,$\{e\}$), Dynamic (p,Tr,$\{e\}$) (with the same entry point e), and the static CG respectively.

[3]The proofs to all theorems can be found in the supplementary material accompanying this paper

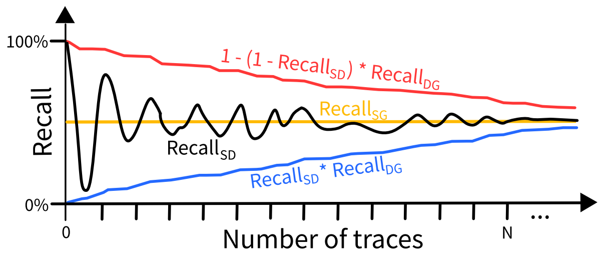


Figure 3: Recall Measure in Relation to Corpus Size

Theorem 3.4 (Precision Bound Increases Monotonically). *For all dynamic call graphs* $D_1 \subseteq D_2$,

$$\text{Precision}_{SD_1} \leq \text{Precision}_{SD_2} \qquad \square$$

Moving on, we observe that establishing a direct lower bound for recall is not possible. Recall_{SD} could exceed or fall short Recall_{SG}. These two measurements are ratios computed from different nominators and denominators, thus lacking an inherent relationship.

$$\text{Recall}_{SD} = \frac{|S \cap D|}{|D|} \gtreqless \frac{|S \cap G|}{|G|} = \text{Recall}_{SG}$$

However, we can establish bounds involving Recall_{DG}, which represents the recall or coverage achieved by D relative to G:

Theorem 3.5 (Bounds on Recall).

$$\text{Recall}_{SD} * \text{Recall}_{DG} \leq \text{Recall}_{SG} \leq 1 - (1 - \text{Recall}_{SD}) * \text{Recall}_{DG} \qquad \square$$

The theorem establishes lower and upper bounds for Recall_{SG}, which cannot be directly measured, because Recall_{DG} cannot be directly measured. But Recall_{DG} is the focus in the optimization process of expanding the input corpus, i.e., it can be systematically approached. The bounds on recall established above get gradually tightened as Recall_{DG} expands due to the growth of the input corpus as depicted in Figure 3:

Theorem 3.6 (Recall Bounds Tighten Monotonically). *For all dynamic call graphs* $D_1 \subseteq D_2$,

$$\text{Recall}_{SD_1} * \text{Recall}_{D_1G} \leq \text{Recall}_{SD_2} * \text{Recall}_{D_2G}$$
$$1 - (1 - \text{Recall}_{SD_2}) * \text{Recall}_{D_2G} \leq 1 - (1 - \text{Recall}_{SD_1}) * \text{Recall}_{D_1G} \qquad \square$$

The distance separating the bounds is $1 - \text{Recall}_{DG}$, tightening as Recall_{DG} is optimized. Thus, Theorem 3.5 also induces an optimization problem.

To recap, we advocate for the utilization of dynamically-sampled CGs, which get systematically refined to more accurately represent the unattainable ground-truth CG by leveraging input corpora.

Observation 5: *Input corpora for dynamic CGs can (and should!) be optimized to yield tight bounds on precision and recall measures.*

At this point, the question remains open, as to how to supply appropriate arguments to entry points to obtain meaningful program runs with comprehensive coverage, which has been identified to be challenging in previous work [27]. In the forthcoming section, we present a method to address this open question, along with the overall method to measure precision and recall by utilizing our new method for constructing dynamic call graphs.

4 OUR METHOD FOR ASSESSING STATIC CGS

Here, we present our architecture for using dynamically-sampled CGs introduced in 3 to assess the precision and recall of static CGs.

High-level Overview. Figure 4 gives an overview of our architecture and its components. A Java program to be analyzed is passed to a *static analysis*, which computes a static CG S, and to a *dynamic analysis*, which records the dynamically-sampled CG D. The resulting static and dynamic CGs are compared to compute precision and recall. This general approach is in line with previous research on dynamic baselines [2, 16, 27, 28]. A key methodological contribution is the addition of a *corpus generation* process, which produces a set of inputs for the selected entry point. This set of inputs is used for running the analyzed program to record D. In the following, we introduce our experimental infrastructure. Subsequently, we elaborate on individual components and discuss how we implemented them for enabling our study of static CGs for Java applications.

Experimental Infrastructure. We build our experimental infrastructure on top of Reif et al.'s [20, 21] JCG toolchain for Java CGs. It provides support for generating static CGs using different algorithms from OPAL 5.0.1 [10, 14], Soot 4.4.1 [30], WALA 1.5.7 [15], and Doop 4.20.14 [5] based on a program-specific configuration. It serializes the CGs to a JSON-based file format and supports some rudimentary comparison of CGs. We extend this toolchain, while reusing its file formats and its support for the program configuration and CG serialization.

4.1 Corpus Generation

The corpus generation process is responsible for generating inputs for running the target program to record D. With these inputs, starting from the selected entry point, the target program should reach as much dynamic program behavior as possible. We combine the following corpus generation techniques.

Existing Corpora We use existing publicly available corpora as a seed corpus of valid and invalid input samples.

Manual Extension We inspect the coverage resulting from the use of existing corpora, and manually create additional inputs that exercise new code paths.

Fuzzing We employ a fuzzer with the initial corpora to generate input covering more dynamic behavior, e.g., edge- and failure cases. Coverage-guided fuzzing increases Recall_{DG}, our dynamically-sampled CG D's coverage of the ground truth, while minimizing human effort to creating the corpora. It does so by instrumenting the target program to measure coverage of the execution, while randomly mutating the given corpus to discover new execution paths. Input data that leads to more coverage is added to the corpus and mutated further. Fuzzing improves D's approximation to G without over-approximation, as defined in Section 3.4, as the program is still executed from the specified entry point.

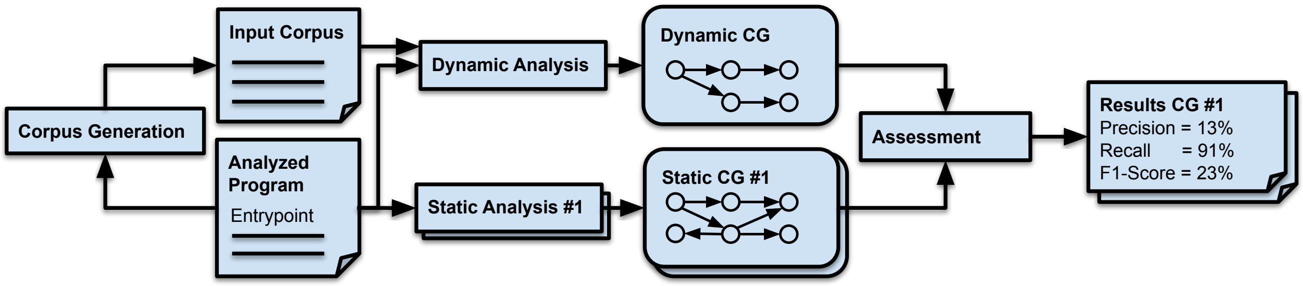


Figure 4: Architecture for Measuring Precision and Recall of Multiple Static Call-Graph Analyses.

In our empirical studies (Section 5), we employ Jazzer, a greybox fuzzer for Java programs using the widely-used LibFuzzer[4]. Greybox fuzzing, the forefront of fuzzing techniques [6, 31], mutates seed inputs to generate new ones. It prioritizes inputs that unveil yet undiscovered execution paths. While fuzzers are often used for vulnerability detection, we use Jazzer to achieve reasonable coverage of dynamic program execution. Thus, we do not require any input prioritization to discover specific execution paths first [31]. Note that our approach is not limited to greybox fuzzing and can be used with other fuzzing or non-fuzzing techniques. Combining all techniques works well by covering both valid (mainly from the initial corpora) and invalid (mainly from fuzzing) inputs.

As discussed in Section 3.4, generating an appropriate input corpus is an optimization problem: finding and adding inputs that exercise new code paths—be it automatically (e.g., by fuzzing for a longer period of time) or manually—will improve D's approximation of G, which, in turn, tightens the bounds on precision and recall.

Some programs are less amenable to this strategy of file-based input corpora combined with fuzzing, e.g., those with graphical user interfaces (GUIs). Still, our approach can be adapted to such programs by employing diverse corpus generation techniques, e.g., incorporating strategies like monkey testing [32] for GUIs.

4.2 Measuring Recall and Precision

Dynamic Analysis. The dynamic analysis records and serializes D, the dynamically-sampled CG. It executes the analyzed program on the input corpus, starting from configured entry points. The dynamic analysis intercepts every method invocation (i.e., every time execution of a method starts) and records the call's caller method, call site (program counter and, if available, line number), and callee method.

The recorded D is context-insensitive as are the static CGs that we assess. Still, D can be used to assess context-sensitive CGs directly or the analysis could be extended to record call strings.

We implemented the dynamic analysis as a Java Virtual Machine Tool Interface [18] agent, which enables automatic interception of runtime events. The agent is attached to the runtime process of the analyzed program, intercepts the `JVMTI_EVENT_METHOD_ENTRY` event, and transmits call edges via a TCP socket to JCG, where they are aggregated and serialized for assessment.

[4]Accessible at https://github.com/CodeIntelligenceTesting/jazzer and https://llvm.org/docs/LibFuzzer.html, respectively

Static Analysis. We want to assess precision and recall of different static CG analyses from different static analysis frameworks. These analyses are run on the target program with the same entry points configured. Using JCG toolchain, we support the OPAL, Soot, WALA, and Doop frameworks and serialize the CGs into JCG's file format for assessment. We compute static CGs for the following widely-used algorithms: Class-hierarchy analysis [29] (OPAL, Soot, WALA), rapid type analysis [3] (OPAL and WALA; Soot's RTA fails to analyze the full JDK 8 because of `MethodHandle` constants that it cannot interpret), and 0-CFA [24] (OPAL, WALA, Doop).

Assessment. Finally, D is used to assess the quality of static CGs S. This involves matching methods in both CGs based on their names and signatures. Call edges are matched using caller and callee methods and the program counter or line number of the call (not all frameworks capture both pieces of information). To facilitate this assessment, we extended JCG. The extension enables matching calls of S against D and subsequently computes precision, recall, and F1-measure, as detailed in Section 3.

5 EMPIRICAL STUDIES

In this section, we assess our approach and employ it to study CGs from different static analysis frameworks. Specifically, we aim to answer the following research questions:

RQ1 Can our approach create high-quality dynamically CGs?
RQ2 How do different corpus generation techniques influence the quality of the dynamically-sampled CGs?
RQ3 How can the quality of static CGs be assessed?

With each RQ, we also provide takeaways for both developers and users of static CG analyses to better assess CG quality.

5.1 Study Setup

Choice of Programs to Analyze and Corresponding Entry Points. The subject of our study are programs from the XCorpus [9], a curated corpus of 76 Java programs, that fulfill the following criteria:

- Can be executed on a single machine without any third party components running. This rules out programs like *Colt*, which requires a compute cluster.
- Do not rely on a GUI, network traffic, or interactive user input for core functionalities, ruling out applications like *Apache Tomcat.*
- Provide a choice of entry point that can cover a substantial part of the program. This rules out libraries like *Apache Commons.*

Table 1: Selected Benchmark Programs

Program	Description	Lines of Code	Entry Point
Axion	SQL database	24 113	`org.axiondb.tools.Console.execute`
Batik	SVG toolkit	179 129	`org.apache.batik.apps.rasterizer.SVGConverter.execute`
Jasml	Java Compiler	5 595	`com.jasml.decompiler.JavaClassParser.parse`
Xerces	XML library	192 110	`org.apache.xerces.jaxp.DocumentBuilderFactoryImpl.parse`

- Consume a structured input format (i.e., grammar-based, not, e.g., raw image data) that has existing corpora and is suitable for mutation-based fuzzing; this excludes e.g., the 3D rendering library Sunflower.
- Are from different application domains and use different input formats, e.g., from XML processors *Apache Xalan* and *Apache Xerces* we only picked one, *Apache Xerces*.

The criteria are designed to guarantee that an automated process, utilizing a predetermined input corpus and automated fuzzing, effectively exercises substantial parts of the program. Moreover, they aim to ensure that analyzed programs present a wide range of challenges for static CGs.

The four programs that do fullfil the criteria are shown in Table 1.[5] For these programs, we define entry points around main methods. The latter are the intended entry points into the respective applications and also starting points for static CG analyses. If there is a single main method, we constructed an entry-point method similar to this main method. The constructed entry point properly sets up and clears resources (e.g., temporary files) and avoids code that requires user interaction. For example, the main method of the SQL database Axion provides an interactive command line to execute SQL commands. We chose an entry point close to that main method which parses SQL commands from a file and executes them. This exercises the core functionality of Axion while at the same time avoiding unrelated user interaction (i.e., reading SQL queries from the command line). If an application has multiple main methods, we tested each to find one whose execution covers significant parts of the code base. In cases where no main methods were available, we delved into the documentation to identify core use cases. Subsequently, we constructed an entry point for the use case that exercises more portions of the code base.

Execution Harness. For each of the selected programs, we developed a concise execution harness with less than 100 lines of code. These harnesses retrieve inputs from the input corpus, initialize essential classes needed to execute the entry point, and subsequently invoke the entry point for each input. Their small size ensures that these harnesses exert minimal influence on the resulting CGs. Moreover, they are integrated into both dynamic and static analyses to maintain consistency for comparisons.

Corpus Generation. For each of the input data formats, we collected publicly available corpora of valid and invalid data. For the SQL database Axion, we extracted SQL commands from the test suites of SQLite, Postgres, and Clickhouse. For the SVG library Batik, we took SVGs from Mozilla's SVG test suite, icons from fonts.google.com, icon set Linea from icons8.com, responsive design vector icons from pixelbuddha.net, and icon set icon-works from fontsquirrel.com. For the Java compiler Jasml, we obtained Java 1.4 class files from four projects in the XCorpus. For the XML library Xerces, we used XML files from the libxml2 and Mozilla test suites and the SVG files for Batik. We obtained some of these inputs from fuzzing corpora publicly available on Github.[6]

Next, we employed standard coverage metrics to evaluate the extent to which our corpus exercises a substantial portion of the code base accessible from the entry point. This involved identifying packages, classes, and methods reachable from the entry point but not exercised by the current input corpus. For instance, Axion's utilization of the non-standard command `LIST SYSTEM TABLES` to generate a summary of internal tables remained unexplored within our initial corpus, which focused solely on standard SQL commands. For each missing class or method, we looked at their names and, if necessary, source code to identify inputs that would exercise the class or method and added these inputs to the corpus. Subsequently, we checked whether the missing class or method was now covered. We worked on this step until no more clear opportunities for extending the input corpus were found.

After crafting input corpora, we executed the fuzzer *Jazzer* to generate additional inputs, often edge and failure cases. We executed the fuzzer repeatedly in one hour intervals, until coverage approached a plateau and no longer increased significantly.

Choice of Static Call-Graph Algorithms. For the study, we selected four algorithms, where each one is available in multiple frameworks: *Class-Hierarchy Analysis* (CHA) [8] considering only the class hierarchy, *Rapid-Type Analysis* (RTA) [3] algorithm refining CHA by considering possible instantiations of types, and *Control-Flow Analysis* (CFA) [24], which incorporates pointer information. We opted for the context-insensitive variant of CFA (0-CFA), which joins results of all call sites of the same method. We studied these algorithms in four state-of-the-art analysis frameworks for Java: OPAL 5.0.1, Soot 4.4.1, WALA 1.5.7, and Doop 4.20.14. Among those, Doop only supports pointer-based CG algorithms, i.e., CFA.

Call-Graph Construction. We constructed both static and dynamic CGs as depicted in Figure 4. We used Adoptium OpenJDK 1.8.0_342-b07 that worked with all static analysis frameworks. We provided 400 GB of heap space and set a timeout of 3 hours. We configured all frameworks to analyze the full JDK for comparability. For Soot, we were unable to generate RTA and 0-CFA CGs: these analyses crashed while analyzing the JDK due to `MethodHandle` constants Soot cannot interpret. This leaves us with a total of eight algorithm implementations for static CGs: three versions of CHA

[5] The lines of code were measured with Tokei (https://github.com/XAMPPRocky/tokei).

[6] https://github.com/strongcourage/fuzzing-corpus/

Table 2: Dynamic Call-Graph Coverage

	Coverage			
Program	**Instruction**	**Branch**	**Method**	**Class**
Axion	56%	58%	53%	86%
Batik	38%	27%	31%	53%
Jasml	80%	65%	81%	98%
Xerces	15%	12%	12%	17%

Table 3: Metrics for Reachable Methods in `com.jasml`

	OPAL			WALA			Soot	Doop
Metric	**CHA**	**RTA**	**0-CFA**	**CHA**	**RTA**	**0-CFA**	**CHA**	**0-CFA**
Precision	95.1%	96.0%	96.4%	95.9%	95.5%	96.4%	96.0%	96.4%
Recall	100%	100%	100%	99.5%	100%	100%	100%	100%
CG Size	225	223	222	222	224	222	223	222

(OPAL, WALA, Soot), two of RTA (OPAL, WALA) and three of 0-CFA (OPAL, WALA, Doop). Moreover, WALA produced no 0-CFA CG for Batik within the given timeout.

5.2 Quality of Dynamic Call Graphs

As discussed in Section 3.4, comparing static CGs S against a CG D that is dynamically sampled from a single entry point provides a lower bound on precision and tightening bounds on recall as D covers more of the ground truth G. Coverage is thus an important indicator of the quality of D. In this section, we assess the coverage of our D (**RQ1**). We measured coverage with *JaCoCo*[7] 0.8.10, running the execution harness described in Section 5.1 on the final input corpus after fuzzing. Table 2 shows the instruction-, branch-, method-, and class coverage for each analyzed program.

According to our definition of the coverage of D (cf. Section 3.1), we are only interested in covering program parts that are reachable from the given entry points. This explains why coverage for some programs can seem low. These programs may contain many packages that are not coverable from the selected entry points. For example, Batik contains a large package `org.apache.batik.apps.svgbrowser` with 217 classes and 966 methods, which cannot possibly be covered from our entrypoint that rasters SVGs to PNGs. Xerces provides the package `org.apache.html.dom` with 61 classes and 710 methods, concerned with HTML-specific functionality, which is not reachable from our entrypoint that parses XML files. Furthermore, Xerces provides a number of classes and methods for traversing and iterating XML documents, which are never invoked while parsing. In contrast, other key packages have high coverage. For example, Batik's package `org.apache.batik.dom.svg`, the core package for the internal model of the SVG format, consists of 163 classes and has 91.4% class coverage, `org.apache.xerces.impl` containing the core Scanner implementations achieves 85%.

We find that our approach achieves high coverage of the program parts reachable from the selected entry points, while avoiding spurious coverage as, e.g., synthesized tests might have. This deliberate avoidance helps us identify imprecise static CGs, which might encompass program parts not within the entry point's reach.

This is also supported by the calculated precision, recall, and size of different CG algorithms using our D. Consider for illustration the metrics for reachable methods in Jasml (Table 3). Every algorithm except WALA CHA found all methods in D (100% recall); also almost all methods found by the algorithms are part of D (96% precision). Since static CGs represent a (supposed) over-approximation of G, while D under-approximates G, a near perfect agreement of all eight S and D suggests that D should be very close to the unattainable G. Hence, in Table 3 we show by example that the dynamic CG D is similar, indeed almost equal, to the static CG S to illustrate that the dynamic CG D can be a valuable approximation, regardless of the kind of CG algorithm, although Jasml may not have too many sophisticated programming constructs.

We conclude that it is feasible to create input corpora achieving high coverage of the analyzed program by manually extending existing corpora and employing a fuzzer to further improve upon them. For programs where these techniques can not easily be applied, e.g., programs with graphical user interfaces, other techniques might be applicable to generate high-quality Ds, such as monkey testing. Overall, to assess the quality of CGs, one should use as reference a D that is recorded on an input corpus optimized for high coverage.

Takeaway 1: *Dynamically-sampled CGs based on generated input corpora can serve as a basis for assessing static CG quality.*

5.3 Effect of Input Generation Techniques

In Section 4.1, we explained how we used various methods to get the coverage values mentioned in Section 5.2. Now, we look into how well these methods work when used alone or together (**RQ2**). This exploration helps us to recognize which methods are more effective overall and whether we really need all of them. It also gives us ideas for future studies on dynamic baselines. This way, we and other researchers can figure out where to focus efforts among these different methods.

To this end, we calculate the coverages of D for each generation technique. The setup is analogous to Section 5.2, with the only distinguishing factor being the actual set of inputs used. We report the resulting coverages for each project in four configurations:

Corpus We only use a publicly available seed corpus as input.
Extension We use the seed corpus as before, but with additional inputs that have been selected manually from other public sources, or hand-crafted after inspecting the program.
Fuzzing We only use fuzzing for generating inputs, without any seed corpus to start with.
Combined We combine all techniques as reported in Section 5.2.

In Table 4, we present the branch coverages obtained from various techniques across all programs. Using only a public seed corpus results in coverages ranging from 20% to 58% out of combined coverages of 27% to 65%. Expanding this corpus manually improves coverage for all programs except Xerces, adding up to 6 percentage points. Fuzzing, when used alone, does not generate good quality Ds, achieving coverages between 1% and 7% out of 27% to 12%. Yet, it plays a crucial role in enhancing the overall coverage when

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

Table 4: Branch-Coverage for Generation Techniques

	Technique			
Program	**Corpus**	**Extension**	**Fuzzing**	**Combined**
Axion	45%	51%	26%	58%
Batik	20%	25%	1%	27%
Jasml	58%	60%	27%	65%
Xerces	10%	10%	7%	12%

combined with other techniques, as it increases the coverage of the extended seed corpus by up to 7 percentage points (Axion).

We see that each step significantly increases the coverage and thus the quality of D.

Takeaway 2: *Public corpora, manual extension, and fuzzing are each necessary for crafting a high-coverage input corpus.*

5.4 Comparing Static Call Graphs

It is common to compare the precision and recall of static CGs S by measuring their size, the number of reachable methods or call edges (cf. Section 2). The common assumption is that smaller CGs imply higher precision [29]. In this section, we evaluate whether this is a sensible shortcut to using a dynamic baseline (**RQ3**). We also reflect upon some other aspects that need to be taken into account in order to meaningfully compare static CGs to each other.

We measure precision and recall of the eight CG algorithm implementations in the four considered frameworks using our dynamically-sampled Ds as baselines. We compute precision and recall for both reachable methods and edges. Furthermore, we separately compute values for the whole CG, including dependencies and the JDK, and for a filtered version that only contains methods/edges inside each program's core application package. This allows a comparison of CG algorithms both on the large and difficult to analyze code base of the JDK, and on the actual application. For every algorithm implementation *and* target program, we obtain a total of four measures (methods/edges and core package/whole CG each) for precision and recall. Each time, we also compute the *size of* S, i.e., its number of methods or edges, respectively.

5.4.1 Implications of Call-Graph Size. If static CG size was a predictor for precision, we would expect that a smaller CG indicates higher precision. Our data shows that this is not always the case; a smaller static CG can also be the result of lower recall. Consider Figure 5, which shows precision and recall on *all edges*, i.e., edges of the whole CG, for implementations of 0-CFA: OPAL's CG for Xerces contains over 5000 fewer edges than Doop's, however its precision is lower by 0.5%, with the missing edges lowering recall. Similar observations can be made for Doop vs. WALA on Jasml, for OPAL vs. Doop on Batik, and for OPAL vs. Doop on Axion—each time the smaller CG exhibits lower (or equal) precision compared to the larger one. Finally, the data presented in Table 3 also shows a smaller CG (WALA CHA) that has lower precision compared to larger ones (e.g., Soot CHA), albeit on a very small scale.

If the static CG size was a predictor for recall, we would expect that a larger CG indicates higher recall. This is often the case, as evidenced by Figure 5; but again not always. For example, Doop's 0-CFA for Jasml in Figure 5 has 2.5K edges more than the OPAL's 0-CFA, yet, recall is lower by almost 30 percentage points. In Figure 6, we observe similar results when comparing the values for size and recall of OPAL's and WALA's CHA on edges of the whole CG.

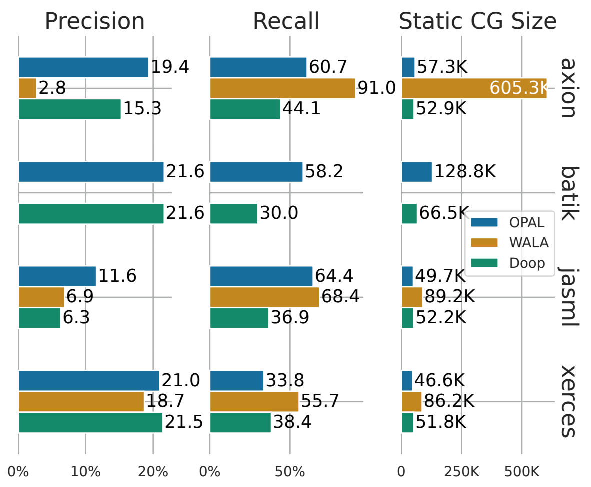


Figure 5: Smaller Graphs with Lower Precision, Higher Recall (0-CFA, Edges incl. JDK)

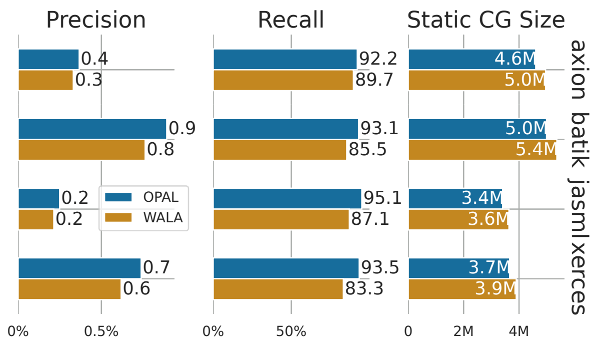


Figure 6: Larger Graphs with Lower Recall (CHA, Edges incl. JDK)

To sum up, our study indicates that merely comparing the sizes of CGs is not enough to assess their relative precision or recall accurately. Neither are smaller CGs necessarily more precise, nor do larger CGs always exhibit higher recall. This is especially true when the CGs are derived from different static analysis frameworks (cf. Figure 6). It is thus essential to explicitly measure precision and recall using high-quality dynamic baselines—ones optimized to closely align with the ground truth.

Takeaway 3: *CG size is not a reliable predictor of CG precision nor recall.*

5.4.2 Effects of Mode of Measurement. Existing works often rely on comparing either reachable methods or call edges to calculate

Table 5: Batik Metrics for All Edges versus All Methods

		Precision		Recall		Size	
Algorithm		Methods	Edges	Methods	Edges	Methods	Edges
CHA	OPAL	7.3%	**0.9%**	**97.6%**	**93.1%**	**194.1K**	5.0M
	WALA	**9.0%**	0.8%	91.6%	85.5%	147.1K	**5.4M**
	Soot	8.7%	0.7%	96.5%	81.1%	160.8K	5.2M
RTA	OPAL	**40.1%**	**9.1%**	63.4%	59.5%	22.8K	313.8K
	WALA	10.5%	1.1%	**92.8%**	**87.5%**	**127.3K**	**3.9M**
0-CFA	OPAL	44.6%	**21.6%**	62.3%	**58.2%**	**20.2K**	**128.8K**
	Doop	**48.8%**	**21.6%**	**65.4%**	30.0%	19.3K	66.5K

precision and recall (cf. Section 2) without discussing the effects of these respective choices. From this lack of discussion, we derive that researchers assume that there are no relevant differences when measuring on methods as opposed to edges. Because a method is deemed reachable regardless of the number of incoming call edges, this assumption is questionable. Our results underline this.

Consider for illustration the results for Batik calculated on reachable methods, respectively edges in Table 5 (similar trends were seen across all projects in our benchmark). For CHA, WALA demonstrates the highest precision in terms of methods (9.0%), while OPAL excels in edge precision (0.9%). Interestingly, OPAL generates the largest CHA CG in terms of methods but the smallest in terms of edges. For 0-CFA, Doop surpasses OPAL by four percentage points in method precision, yet their edge precision remains comparable. Although Doop exhibits slightly higher method recall than OPAL, this reverses when considering edges—where OPAL nearly doubles Doop's edge recall. This suggests that Doop misses many call edges, which is hidden in the measure for methods as the methods are still found to be reachable by at least one other edge.

This suggests that looking only at reachable methods (or only edges) is not sufficient to assess CG quality. In particular, both precision and recall on edges are often significantly lower than on methods. CG algorithms showing promising quality on methods may thus be significantly worse regarding edges. Results for both need to be provided when evaluating CG algorithms; for some applications reachable methods may be more important than edges (e.g., vulnerable method detection), and vice versa (e.g., taint analysis).

Takeaway 4: *Looking only at a single mode of measurement misses important insights.*

Takeaway 5: *Precision and recall in terms of call edges are often significantly lower than in terms of reachable methods.*

5.4.3 *Effects of Concrete Implementations.*

Users of CGs might decide for a CG analysis by selecting an algorithm from the implementations in their preferred framework based on its theoretical or perceived characteristics, e.g., RTA as a more precise algorithm than CHA. Our study reveals that concrete implementations of CG algorithms may have surprising effects on the actual precision and recall.

Take the example of WALA in Table 3: RTA and 0-CFA show perfect recall (100%), but CHA falls a bit short with 99.5% recall. This is unexpected because CHA, while being the least precise, is assumed to be the algorithm with the highest recall among the three kinds of algorithms. Also, according to the definitions of RTA and

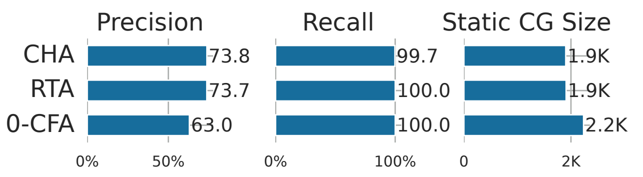


Figure 7: More Complex Algorithms with Lower Precision (Axion Core Package Methods, WALA)

CHA algorithms, RTA CGs should be a subset of CHA CGs, which means RTA's recall should not be higher than CHA's. Note that Table 3 is not suitable for drawing conclusions about differences in CG algorithms. Another surprising observation is that allocating additional time and resources to compute a seemingly more accurate CG can sometimes lead to the opposite outcome. For instance, when examining WALA's performance on Axion (Figure 7), we observe that the more sophisticated the algorithm, the less precise the results become. Specifically, CHA attains 73.8% precision while 0-CFA achieves only 63.0% precision.

The results for Xerces' core package shown in Figure 8 reveal another insight on specific implementations. Xerces is used via JDK methods using reflection to instantiate Xerces classes given in a system property (Listing 1). None of the 0-CFA implementations can resolve this reflection, resulting in empty CGs (an extreme corner case). Additionally OPAL's on-the-fly RTA considers classes instantiated only when an instantiation is found in a method already considered reachable. This misses the reflective instantiation of `org.apache.xerces.jaxp.DocumentBuilderFactoryImpl` when `newDocumentBuilder()`, again resulting in an empty CG. WALA's RTA, using a different design, considers the class instantiated and achieves 100% recall. This highlights how decisions on the implementation of an established algorithm can lead to large differences in resulting CGs, even for the same conceptual algorithm. As a side note, this shows that developers of Java analysis frameworks should strive to support reflection as best as they can, if high recall is desired. In this corner case, a single reflective invocation (combined with system properties) lead to completely invalid results.

```
System.setProperty(
"javax.xml.parsers.DocumentBuilderFactory",
"org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
javax.xml.parsers.DocumentBuilder parser =
DocumentBuilderFactory.newInstance().newDocumentBuilder();
parser.parse(new ByteArrayInputStream(input));
```

Listing 1: Entrypoint for Xerces

As theoretical characteristics of CG algorithms can be misleading and, for the same conceptual algorithm, the framework of choice may have a significant impact, one should compare the results of different CG implementations, if possible from multiple frameworks, instead of relying on perceived theoretical properties.

Takeaway 6: *Concrete implementations of static CG algorithms may have unexpected effects on precision and recall.*

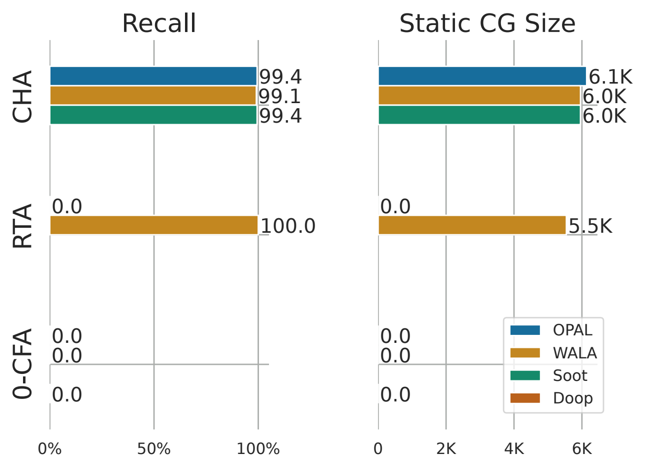


Figure 8: Empty Graphs due to Reflection (Xerces Core Package Methods)

The example of Xerces also shows how static CG quality assessed on one program may not generalize to other programs. When examining CGs, researchers select a specific set of programs to study, often chosen from a curated corpus. Findings may be confined to the chosen set. One should thus evaluate on diverse programs to reason about CG quality. For users, when choosing a CG algorithm and analysis framework for a specific use case, it is crucial to verify validity and usefulness of existing claims about their quality.

Takeaway 7: *Observations about the quality of static CGs on some programs may not generalize to other programs.*

5.5 Threats to Validity

5.5.1 Internal Validity. We measure recall and precision of static CGs using a dynamic CG D as the reference baseline, an under-approximation of the unattainable ground-truth CG G. If the D is not a good approximation of G, then precision and recall measured using D will deviate from the actual precision and recall that would be measured by using G. However, as explained in Section 3.4, this is a problem of optimizing D's coverage of G, Recall_{DG}, to ensure tighten the bounds between the measured and true values. We perform this optimization by extending corpora both manually and using state-of-the-art fuzzing to ensure a good coverage of G.

Our results might also be affected the setup of the static analysis frameworks. To ensure fair comparison, we made all frameworks analyze the entire JDK, even though their default configurations exclude different JDK packages, and enforced a timeout of 3 hours. As a result, we could not generate CGs for all algorithms for all programs. However, both measures are necessary to ensure fair comparison between different implementations of a each algorithm.

5.5.2 External Validity. For our four benchmark programs, we selected one entry point each. While we ensured these entry points cover the programs' core use cases, findings might not generalize other entry points. Instead of analyzing multiple entry points for one program, we considered assessing a more diverse set of programs more valuable to gain broader insights.

To obtain D, we use fuzzing, non-deterministic process, so results may vary slightly depending on the resources spent. We applied fuzzing until no new paths were found to minimize this risk.

6 CONCLUSION

Call-graph quality is crucial for many static analyses. Prior work often relied on the number of reachable methods to measure precision, or micro-benchmarks to assess recall. Where a dynamic baseline was used, it was derived from test cases and may thus not be representative of actual program execution.

We showed that construction of a *good* dynamic baseline is an optimization problem when entry points are considered. This yields bounds on precision and recall that approach the unattainable real values as the baseline is improved. We proposed a methodology to compute such baselines, employing public corpora, manual extension, and fuzzing for optimization. Using this, we recorded dynamic CGs for four Java programs and used them to assess statically generated CGs from four popular static analysis frameworks.

Our studies showed that CG size is not a reliable predictor for precision or recall. We also found that considering just reachable methods misses important insights revealed by also looking at call edges. Finally, we observed that the quality of CGs from a single framework or algorithm can vary widely. To be certain of the quality of a concrete static CG for a given use case, one cannot rely on any proxy measure, but instead has to evaluate against a good dynamic baseline, yielding guaranteed bounds on precision and recall.

7 DATA AVAILABILITY

Proofs, scripts, and evaluation data are included in supplementary material. An artifact[8] provides them together with the full input corpora.

ACKNOWLEDGMENTS

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 their joint support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES

[1] Karim Ali and Ondrej Lhoták. 2012. Application-Only Call Graph Construction. In *ECOOP 2012 - Object-Oriented Programming* (Beijing, China) *(ECOOP'12)*. Springer, 688–712. https://doi.org/10.1007/978-3-642-31057-7_30
[2] Gábor Antal, Péter Hegedűs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. 2023. Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools. *IEEE Access* 11 (2023), 25266–25284. https://doi.org/10.1109/ACCESS.2023.3255984
[3] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Jose, CA, USA) *(OOPSLA'96)*. ACM, 324–341. https://doi.org/10.1145/236337.236371
[4] Eric Bodden. 2018. The Secret Sauce in Efficient and Precise Static Analysis: The Beauty of Distributive, Summary-Based Static Analyses (and How to Master Them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, The Netherlands) *(SOAP'18)*. ACM, 85–93. https://doi.org/10.1145/3236454.3236500
[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*

[8] https://doi.org/10.5281/zenodo.10888531

(Orlando, FL, USA) *(OOPSLA'09)*. ACM, 243–262. https://doi.org/10.1145/1640089.1640108

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, TX USA) *(CCS'17)*. ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020

[7] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. 2022. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs. In *36th European Conference on Object-Oriented Programming* (Berlin, Germany) *(ECOOP'22)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.3

[8] 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* (Åarhus, Denmark) *(ECOOP'95)*. Springer, 77–101. https://doi.org/10.1007/3-540-49538-X_5

[9] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. XCorpus–An executable Corpus of Java Programs. *Journal of Object Technology* 16, 4 (2017), 1:1–1:24. https://doi.org/10.5381/jot.2017.16.4.a1

[10] Michael Eichberg and Ben Hermann. 2014. A software product line for static analyses: the OPAL framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (Edinburgh, United Kingdom) *(SOAP'14)*. ACM, 1–6. https://doi.org/10.1145/2614628.2614630

[11] David Grove and Craig Chambers. 2001. A Framework for Call Graph Construction Algorithms. *ACM Transactions on Programming Languages and Systems* 23, 6 (2001), 685–746. https://doi.org/10.1145/506315.506316

[12] Tobias Gutzmann, Antonina Khairova, Jonas Lundberg, and Welf Löwe. 2009. Towards Comparing and Combining Points-to Analyses. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation* (Edmonton, Canada) *(SCAM'09)*. IEEE, 45–54. https://doi.org/10.1109/SCAM.2009.14

[13] Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *36th European Conference on Object-Oriented Programming (ECOOP'22, Vol. 222)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 30:1–30:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.30

[14] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. 2020. Modular Collaborative Program Analysis in OPAL. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE'20)*. ACM, 184–196. https://doi.org/10.1145/3368089.3409765

[15] IBM. 2024. WALA - Static Analysis Framework for Java. http://wala.sourceforge.net/. [Online; accessed 11-March-2024].

[16] Ondvrej Lhoták. 2007. Comparing Call Graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (San Diego, CA, USA) *(PASTE'07)*. ACM, 37–42. https://doi.org/10.1145/1251535.1251542

[17] Linghui Luo, Goran Piskachev, Ranjith Krishnamurthy, Julian Dolby, Eric Bodden, and Martin Schäf. 2023. Model Generation For Java Frameworks. In *2023 IEEE Conference on Software Testing, Verification and Validation* (Dublin, Ireland) *(ICST'23)*. IEEE, 165–175. https://doi.org/10.1109/ICST57152.2023.00024

[18] Oracle. 2024. JVM(TM) Tool Interface. https://docs.oracle.com/en/java/javase/20/docs/specs/jvmti.html. [Online; accessed 12-March-2024].

[19] 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* (Seattle, WA, USA) *(FSE'16)*. ACM, 474–486. https://doi.org/10.1145/2950290.2950312

[20] 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* (Beijing, China) *(ISSTA'19)*. ACM, 251–261. https://doi.org/10.1145/3293882.3330555

[21] 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* (Amsterdam, The Netherlands) *(SOAP'18)*. ACM, 107–112. https://doi.org/10.1145/3236454.3236503

[22] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366. https://doi.org/10.2307/1990888

[23] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *43rd IEEE/ACM International Conference on Software Engineering* (Madrid, Spain) *(ICSE'21)*. IEEE, 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

[24] Olin Shivers. 1988. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, GA, USA) *(PLDI'88)*. ACM, 164–174. https://doi.org/10.1145/53990.54007

[25] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, TX, USA) *(POPL'11)*. ACM, 17–30. https://doi.org/10.1145/1926385.1926390

[26] 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 *Programming Languages and Systems* (Wellington, New Zealand) *(APLAS'18)*. Springer, 69–88. https://doi.org/10.1007/978-3-030-02768-1_4

[27] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE'20)*. ACM, 1049–1060. https://doi.org/10.1145/3377811.3380441

[28] Zoltán Ságodi, Edit Pengő, Judit Jász, István Siket, and Rudolf Ferenc. 2022. Static Call Graph Combination to Simulate Dynamic Call Graph Behavior. *IEEE Access* 10 (2022), 131829–131840. https://doi.org/10.1109/ACCESS.2022.3229182

[29] Frank Tip and Jens Palsberg. 2000. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, MN, USA) *(OOPSLA'00)*. ACM, 281–293. https://doi.org/10.1145/353171.353190

[30] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON'99)*. IBM Press, 13.

[31] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Network and Distributed Systems Security Symposium* (San Diego, CA, USA) *(NDSS'20)*. Internet Society, 2329–2344. https://doi.org/10.14722/ndss.2020.24422

[32] Thomas Wetzlmaier, Rudolf Ramler, and Werner Putschögl. 2016. A Framework for Monkey GUI Testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation* (Chicago, IL, USA) *(ICST'16)*. IEEE, 416–423. https://doi.org/10.1109/ICST.2016.51

Received 16-DEC-2023; accepted 2024-03-02