Received 18 November 2022, accepted 8 December 2022, date of publication 14 December 2022, date of current version 22 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3229182

RESEARCH ARTICLE

Static Call Graph Combination to Simulate Dynamic Call Graph Behavior

ZOLTÁN SÁGODI, EDIT PENGŐ, JUDIT JÁSZ, ISTVÁN SIKET, AND RUDOLF FERENC
Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary

Corresponding author: Judit Jász (jasy@inf.u-szeged.hu)

This work was supported in part by the European Union Project Within the Framework of the Artificial Intelligence National Laboratory under Grant RRF-2.3.1-21-2022-00004; in part by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Project Implemented through Funding Scheme under Grant TKP2021-NVA; and in part by the University of Szeged Open Access Fund under Grant 5894.

ABSTRACT Call graphs are fundamental for many higher-level code analyses. The selection of the most appropriate call graph construction tool for an analysis is not always straightforward and depends on the purpose of the results' further usage. The choice of call graph construction tool has a great effect on the following tasks' execution time, memory usage, and result quality. This research compares the resulting static and dynamic Java call graphs to assist in the selection of the most appropriate tools. Static call graphs, as their name suggests, are constructed by static analysis, based on the source code or the bytecode, without executing tests or any code parts. This means that the project can be analyzed in its early stages and with fewer resources, but there is concern that this will result in less accurate, noisier graphs since the dynamic behavior of the programs will be estimated by static algorithms. Inaccuracies can greatly affect analyses based on call graphs. On the other hand, dynamic call graphs are created during the actual execution of the program. The calls that are included as edges in the graph are exactly those that were executed during the run, so you can expect the result to be more accurate. However, dynamic analysis requires more resources and the execution of code via test cases which provide high test coverage. In this work, we investigated the relationship between dynamic and static call graphs. Is the graph generated by dynamic analysis really better? Can static graphs approximate or even complement dynamic call graphs with sound results? In order to find the answers to these questions, we compared the results of five static and one dynamic analyzer. They were evaluated on three projects of different sizes and test coverage. We included in the comparison a merged graph created by ourselves by combining different static analyzer outputs. Not only did we compare static graphs to the dynamic results, we also validated the calls in a dynamic graph and found that these graphs could mislead the user. The results show that dynamic graphs should be considered good, although not a golden standard since they contain phantom calls, calls that are not present in the source code. Such calls are not limited to synthetic calls. Static analyzers could not be applied without consideration either, but a combination of static call graphs does tend to contain similar calls to the dynamic graphs with no phantom calls.

INDEX TERMS Call graph, dynamic analysis, java, static analysis.

I. INTRODUCTION

Call graphs are directed graphs that contain the methods of a program as nodes. If method A calls method B it is represented by a directed edge from node A toward node B in the graph. In software quality assurance there are many techniques that rely on call graphs. It is essential that the call graphs used for these quality assurance methods are suitable for the purpose as different call graphs highly impact the results [30]. For this reason, it is important to study the differences between call graphs produced by different approaches. In this study, we discuss Java call graphs, so the following

The associate editor coordinating the review of this manuscript and approving it for publication was Hui Liu.

statements refer mostly to Java-based call graphs. Call graphs can be generated in two ways, by static or dynamic analysis. Static analysis is based on the source code or the byte code, therefore, unlike dynamic analysis, it does not require the execution of the program. This is both an advantage and a disadvantage. Static analysis can be performed easier and faster, but dynamic analysis can detect exactly which calls have been executed. A perfect example is the case of unreachable code. A static analyzer might add the calls starting from the unreachable code part making its result less sound, however, the dynamic analyzer would not add that part since unreachable code is never executed and thus cannot be reached by the analyzer. Detecting a call is achieved by code instrumentation in most cases. Instrumentation adds code to the program that helps to trace which methods are calling each other at runtime. This usually affects the performance of the program negatively. Another disadvantage of the approach is that only those calls will appear in the resulting call graph that have occurred during execution. Therefore, in order to get a more exhaustive call graph, large test coverage is needed for the analyzed project. Based on the above presumptions, a naive approach is that although dynamic analysis is costly, the produced call graphs are better than static ones [34], [39]. Neville et al. [17] stated that static analyses are unsound and this unsoundness is relevant for every language that has reflection, native code or dynamic loading. Languages based on JVM are highlighted considering this problem. We test this hypothesis (i.e. that dynamic analysis produces better call graphs) by answering the following research questions:

- **RQ1: Is it possible to approximate dynamic call graphs using only static call graphs?**
- **RQ2: Is it possible to cover the dynamic call graphs using static call graphs but with more soundness?**
- **RQ3: Can dynamic call graphs be considered a "golden standard" for call graphs?**

As can be seen, RQ1 and RQ2 are very similar but they differ in a major factor. Static tools can generate a complete graph that covers all the edges between nodes that are present in both graphs. So RQ1 asks if it is possible to find similar static nodes and edges as dynamic ones, while RQ2 asks if it is possible to find those similar nodes and mostly those nodes without any "additional" nodes.

To answer these questions, we investigated the similarity between the two types of call graphs produced by static and dynamic analysis. We have already shown in an earlier work [23] that static call graphs themselves can be very different. In that work, we evaluated 6 open-source call graph-building tools on multiple larger-scale, open-source projects. The results showed that the structure of the generated call graphs is strongly influenced by the capabilities and the processing method of the static analyzer tools. In the current study, we have included a dynamic analyzer for the call graph comparison. The analyses were carried out on three Java projects that vary in size, test coverage, and their usage of Java-specific features. The raw results of these comparisons are available in an online appendix, which also contains the static call graph generators and the source code of the call graph comparator program. The appendix is available at this link: `https://doi.org/10.5281/zenodo.7003920`. The source of the dynamic analyzer is located in a separate repository available at `https://github.com/sed-szeged/java-instrumenter`.

The rest of the paper is organized as follows. In Section II, we discuss the related work. Section III introduces the five static analyzer tools altogether with 6 different settings that we used for our research, while the dynamic analyzer (instrumenter) is described in Section IV. Section V briefly describes the tool we have developed to pair and compare call graphs. Our heuristic algorithm to generate new call graphs based on already existing ones is included too. In Section VI, we evaluate the analyzers on three Java projects. Section VII lists the threats to validity, while Section VIII summarizes our work.

II. RELATED WORK

Call graphs are the basis of many software analysis algorithms, such as control flow analysis, program slicing, program comprehension, bug prediction, refactoring, bug-finding, verification, security analysis, and whole-program optimization [9], [16], [44], [46]. The precision and recall of these applications depend largely on the soundness and completeness of the call graphs they use. Moreover, call graphs can be employed to visualize the high-level control flow of the program, thus helping developers understand how the code works. There are several studies about dynamic call graph-based fault detection, like the work of Eichinger et al. [15], who created and mined weighted call graphs to achieve more precise bug localization. Liu et al. [27] constructed behavior graphs from dynamic call graphs to find non-crashing bugs and suspicious code parts with a classification technique.

Regardless of whether the examined language is low-level and binary or high-level and object-oriented, call graph construction can always lead to some difficulties [5], [33]. A call graph is accurate if it contains exactly those methods and call edges that might get utilized during an actual execution of the program. However, in some cases, these can be hard to calculate. For example, if several call targets are possible for a given call site, then a deeper examination is needed to determine which ones to connect as precisely as possible. This examination can be done in a context-dependent or context-independent manner; naturally, the choice influences the generated call graph [18]. Context-dependent methods are more accurate in return for greater resource usage. To mitigate the resource demands of such methods, the analysis of the programs often only starts from the `main` method or a few entry points instead of starting from every method. This might result in a less accurate call graph. To improve the accuracy of context-independent methods, the following algorithms can be used for object-oriented languages: *Class Hierarchy Analysis (CHA)* [11], *Rapid Type Analysis (RTA)* [5], *Hybrid Type Analysis (XTA)* [42], *Variable Type Analysis (VTA)* [40].

Another important question during call graph creation is the handling of library calls [2]. Including library calls not only makes the call graph bigger, it also requires the analysis of the libraries, which can be quite resource-consuming. However, the exclusion of library elements may cause inaccuracies when developers implement library interfaces or inherit from library classes. The analysis of library classes might involve private, inaccessible methods as well. Michael Reif et al. [33] discussed the problem that the often-used algorithms, such as *CHA* and *RTA*, do not treat libraries differently than applications. However, this can lead to inaccuracies, such as unnecessary edges or important edges being missed. The recommended algorithm in this work can reduce the number of call edges drastically depending on the analyzer and the project itself. The tools we selected for our comparison represent library calls and library methods at various levels of detail.

As mentioned in Section I, many comparative studies are available about call graph creation. Grove et al. [19] implemented a framework for comparing algorithms for call graph creation and assessed the results with regard to precision and performance. Murphy et al. [29] carried out a study similar to ours about the comparison of five static call graph creators for C. They identified significant differences in how the tools handled typical C constructs like macros. Hoogendorp gave an overview of call graph creation for C++ programs in his thesis [20]. Antal et al. [4] conducted a comparison of JavaScript static call graph creator tools. They collected five call graph builders and analyzed the handling of JavaScript language elements and the performance as well. As a result, they provided the characterization of the tools that can help in selecting the one that is most suitable for a given task. Tip et al. [42] tried to improve the precision of *RTA* by introducing a new algorithm. On average, they reduced the number of methods by 1.6% and the number of edges by 7.2%, which can be a considerable amount in the case of larger programs. Lhoták [26] compared static call graphs generated by Soot [37] and dynamic call graphs created with the help of the *J [36] dynamic analyzer. He built a framework to compare call graphs, discussed the challenges of the comparisons, and presented an algorithm to find the causes of the potential differences in call graphs.

In other papers, In this paper, we show these differences and, unlike in any other paper known to us, try to figure out how static results could approximate dynamic call graphs with high soundness in case dynamic call graphs can be taken as ground truth.

III. STATIC CALL GRAPH CONSTRUCTION TOOLS

We studied numerous static analyzer tools for Java to decide whether they could generate – or could be easily modified to generate – call graphs. We searched for widely available, open-source programs from recent years, which could analyze complex, real-life Java systems. We discarded many plug-in-based tools, as they produced only a visual output (e.g., CallGraph Viewer [7]), while other promising candidates were not robust enough on larger systems (e.g., Java Call Hierarchy Printer [6]). In some cases, the call graphs had to be extracted directly from the inner representation of the analyzer. However, we eliminated any tool that did not provide enough information to reconstruct the caller-callee relationships between compilation units without major development (e.g., JavaParser [10]). The descriptions of the five tools that met our selection criteria are presented below.

A. SOOT

Soot [37] is a widely used language manipulation and optimization framework developed by the Sable Research Group at McGill University. It supports analysis up to Java 9 and works on the compiled binaries. Although its latest official release was in 2012, the project is still active on GitHub, from where we acquired the 4.2.1 release, which was the latest version at the time. Soot has a built-in call graph creator functionality that can be parameterized with multiple algorithms. We employed the *CHA* algorithm and the *SPARK* analyzer with VTA enabled during construction. These algorithms differ in their handling of polymorphic calls. When using CHA, Soot creates the inheritance graph of a program, and during call graph construction only the inherited classes are used in polymorphic calls, thus reducing the number of infeasible calls. In the case of the VTA [40] algorithm, Soot creates an inner representation where the flow of object references can be followed. This information is used to resolve polymorphic method calls.

B. OPENSTATICANALYZER

OpenStaticAnalyzer (OSA) [12] is an open-source, multi-language static analyzer framework developed by the University of Szeged. It calculates source code metrics, detects code clones, performs reachability analysis, and finds coding rule violations in Java, JavaScript, Python, C#, and C/C++ projects. We extracted the call graph of our project by traversing its Abstract Syntax Tree[1] (AST) internal representation and collecting every available invocation information.

C. SPOON

SPOON [31] is an open-source, feature-rich Java analyzer and transformation tool for research and industrial purposes. It is actively maintained, supports Java up to version 9, and while several higher-level concepts (e.g., reachability) are not provided,,out of the box'', the necessary infrastructure is accessible for users to develop their own. SPOON performs a directory analysis of the source code and builds an AST-like metamodel, which is the basis for further analyses and transformations. Similar to the above-mentioned

[1] Abstract Syntax Tree represents the syntactic structure of the source code in a hierarchical tree-like form.

OSA implementation, the call information can be obtained by processing the AST-like inner representation of SPOON. The library is well-documented and provides a visual representation of its metamodel, which helped us thoroughly study its structure. We used the 7.0.0 version for our research.

D. WALA

WALA [45] is a static and dynamic analyzer for Java bytecode (supporting syntactic elements up to Java 8) and JavaScript. Originally, it was developed by the IBM T.J. Watson's Research Center; now it is actively developed as an open-source project. Similar to Soot, it also has a built-in call graph generation feature with a wide range of graph-building algorithms. We used the *ZeroOneContainerCFA* graph builder for our research, as it performs the most complex analysis. It provides an approximation of the Andersen-style pointer analysis [3] with unlimited object sensitivity for collection objects. The generator had to be parameterized with the entry points, from which the call graphs would be built. To make the results similar to the results of the other tools, we treated all not-private, non-abstract methods as entry points (instead of just the `main` methods). For other configuration options, we used the default settings provided in the documentation and example source codes.

E. ECLIPSE JDT

Eclipse JDT [14] is one of the main components of the Eclipse SDK [13]. It provides a built-in Java compiler and a full model for Java sources. We created a JDT-based plugin for Eclipse Oxygen that supports Java 10 code, to extract the call graph from the extensive, AST-like inner representation.

IV. DYNAMIC CALL GRAPH CONSTRUCTION

A dynamic call graph is a record of a program execution, therefore, they are built in an entirely different way from static call graphs [35]. Considering Java, there are two main methods to generate them: source code-based and bytecode-based instrumentation [8], [41]. In both cases, the instrumentation is done by inserting measurement probes into the source- or bytecode [24]. They both have drawbacks and benefits. For example, bytecode-based instrumentation does not require the source code or a separate build, it can be implemented more easily. On the other hand, source code instrumentation can be considered more general, as it does not depend on the bytecode version and the Java VM [21]. As shown by code coverage measurements, bytecode-based instrumentation tools may produce different results than source code-based ones. However, it was proven in Horváth's work [21] that with proper filtering settings the difference is insignificant.

We used a tool[2] developed at the University of Szeged to build dynamic call graphs that utilize a common Java agent-based, on-the-fly bytecode instrumentation approach to collect call chain information that can later be transformed into a call graph. During the instrumentation process, probes are inserted into all methods that are relevant to the analysis (i.e. they belong only to the selected module). These probes are guarding every method's entry and exit points i.e., they trigger every time the execution reaches a method call and when the execution leaves a method e.g., by reaching a *return* or a *throw* statement. Running these probes on every method entry and exit can have a huge impact on execution time and memory consumption. Therefore, selecting relevant methods is a key point since without this selection every method would be instrumented, which would result in an exploded call graph or sometimes even the analysis not being carried out due to resource limits. We configured the instrumentation tool to include all methods that are part of the production code of the actual project, and excluded test code, dependencies, and built-in Java libraries.

The instrumentation-based approach requires running the programs with pre-set values and settings that ensure that only one deterministic flow exists, provided that no randomness or any kind of failure like memory shortage occurs. For this, we used the developer-written tests that were included in the projects' sources. Typically, one test case executes only a small part of the possible control flow at a time, therefore, multiple executions of the program are needed for the most complete analysis. Every test case adds those calls to the call graph that were covered during the execution of that particular test case. If calls are covered by multiple test case executions then no extra information is added to the call graph. As the quality of the dynamic call graph depends on the test suit, we did not add new test cases or remove any of the test cases in order to maintain the project's original state. Although we did not add any test cases for better dynamic results, we selected the projects evaluated in this work carefully. The projects in this work reflect the importance of coverage as they differ to a great extent. We did this measurement with a commonly used tool, JaCoCo[3] using branch coverage since it is related to the number of possible execution paths and is still not too hard to compute.

We argue that test suites that have higher code coverage values are better for dynamic analysis, in other words, higher coverage means higher precision since we can gather more information. In this paper, the analyzed projects vary in coverage in order to show how much effect different levels of coverage have on the results.

V. CALL GRAPH COMPARISON AND COMBINATION

A purely manual comparison of the static and dynamic call graphs is not possible due to their size. Because of the increasing research interest in extremely large graphs and their widespread usage, there are a vast number of algorithms available for comparing general directed and undirected graphs [1], [25], [28], [43]. However, these methods cannot

[2]https://github.com/sed-szeged/java-instrumenter

[3]https://www.jacoco.org/jacoco/

be directly applied to call graphs, especially if they were produced by different analyzer tools. They might represent the same method in different ways, therefore, we developed a call graph comparison method based on node pairing. We started the development and implementation of this comparison algorithm in a previous work [32]. The method name-based pairing mechanism had to be refined in several steps due to the different operation of static analyzers. For the current research, we have enhanced this comparator tool to handle dynamic call graphs as well and developed other features to help evaluate the results.

Some examples of the information provided by the comparator tool:

- Summary table of the percentage of common calls.
- Graph-to-graph comparison of common and diverse calls.
- List of calls that are missing from one graph but are present in the other (where both graphs contain the methods associated with these calls).

For the purpose of this paper, we added extra functionality to this tool[4] thus it is possible to create new call graphs based on the other already loaded call graphs. This created graph is also included in our measurements. The created graph is a heuristically combined graph. In our previous work, we found that the algorithm CHA can explode graphs so we had to handle these cases. If a call site calls the same method multiple times in different classes after a limit it is considered to be a CHA explosion. For example, if a `foo` method calls the `clone` method of every class then it is considered a CHA explosion. For this research, the call limit was set to 5 but it could be tuned if necessary. If such an explosion is detected we delete every call in the SootCHA's graph and substitute it with another tool's calls from the same call site. We include at most two methods from the other graph. We perform an additional filtering step on this artificially created graph. SootCHA's graph is full of calls that are associated with static class initialization (`clinit`) blocks as the algorithm includes them every time a class is used. Therefore, we exclude the (`clinit`) nodes from the graph if only Java library calls are originating from them.

For better understanding, we include a diagram (Figure 1) about how the data flows in this process. We transform every graph into a uniform representation and we process those uniform graphs. We also create a graph (Uniform UnionGraph) in this uniform format that contains every node and edge from every static analyzer. The final calculations are done on the uniform graphs and in the following parts of this work a graph is meant to be in the uniform format.

VI. EVALUATION

We have evaluated the static and dynamic call graphs of three carefully selected Java systems. These systems are Joda-Time,[5] the Core module of Maven[6] (Maven-Core), and the Sorter module of the Sortpom[7] project. Joda-Time and Maven-Core are part of the Defects4J [22] dataset. Sortpom is an open-source project from GitHub.

TABLE 1. Properties of the analyzed projects.

	LOC	Test coverage
Joda-Time	28.7k	81%
Maven-Core	27.7k	39%
Sortpom	1.8k	96%

We chose these three systems because they differ significantly in size (Lines of Code, LOC), functionality, and test coverage. Table 1 summarizes the properties of the systems. The coverage values represent branch coverage.

Since dynamic analysis is highly dependent on test coverage, we have taken this feature into account in the selection process. As shown in Table 1, the Maven-Core project has low coverage, Joda-Time has medium-high coverage, while the Sortpom project has exceptionally high test coverage. Having various coverage values allows us to determine how it affects the results of the dynamic analysis.

The following three subsections compare the dynamic and static graphs obtained from these three projects. The RQs are also answered in the analyses.

A. JODA-TIME

As Table 1 showed, Joda-Time is a medium-sized project with relatively good test coverage. It was the de-facto standard date and time library before Java 8.

The comparison of the call graphs produced on Joda-Time is summarized in Table 2. The first line and the first column of the table depict the names of the analyzer tools. The static analyzers are highlighted in orange, the dynamic analyzer (DYN) in pink, and UnionGraph (UG) in blue. UnionGraph includes every graph created by every static analyzer. Therefore, it can be considered an upper bound on what static analyzers can detect in the dynamic analyzer's graph. Soot was run with two different configurations: we tested it with both the VTA (SootVTA) and CHA (SootCHA) algorithms.

The diagonal elements of Table 2 (depicted in bold) show the number of different calls found by each analyzer tool. Every other cell in a row is a percentage that displays how many percent of the given tool's calls was found by the tool in the column, respectively. For example, the value of 75.36 in the second column of the fourth row means that 75.36% of the calls of the OSA tool were found also by SootCHA (which is 7,530 calls). If we consider OSA's calls as the relevant elements, this value is the same as the recall but in

[4]https://doi.org/10.5281/zenodo.7003920

[5]https://github.com/JodaOrg/joda-time/tree/6ad133837a4c4f8199d00a05c3c16267dbf6deb8

[6]https://github.com/apache/maven/tree/cecedd343002696d0abb50b32b541b8a6ba2883f/maven-core

[7]https://github.com/Ekryd/sortpom/tree/0afd7ae3ecb2bf93be45a45723e467e868404173/sorter

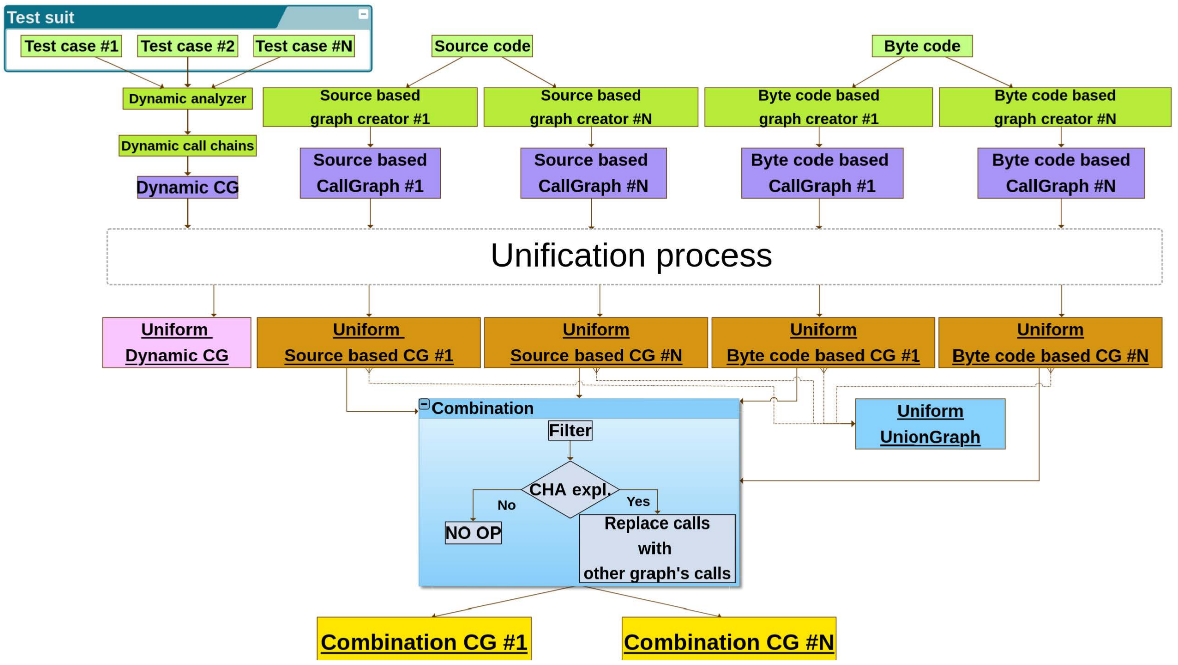

Test suit
Test case #1
Test case #2
Test case #N
Source code
Byte code
Dynamic analyzer
Source based graph creator #1
Source based graph creator #N
Byte code based graph creator #1
Byte code based graph creator #N
Dynamic call chains
Dynamic CG
Source based CallGraph #1
Source based CallGraph #N
Byte code based CallGraph #1
Byte code based CallGraph #N
Unification process
Uniform Dynamic CG
Uniform Source based CG #1
Uniform Source based CG #N
Uniform Byte code based CG #1
Uniform Byte code based CG #N
Combination
Filter
CHA expl.
No
Yes
NO OP
Replace calls with other graph's calls
Uniform UnionGraph
Combination CG #1
Combination CG #N

FIGURE 1. How we process the data and graphs.

TABLE 2. Joda-Time comparison results.

	SootCHA	SootVTA	OSA	SPOON	WALA	DYN	JDT	UG
SootCHA	**34,161**	39.33%	22.04%	22.61%	58.76%	26.84%	21.86%	100%
SootVTA	100%	**13,436**	32.67%	34.05%	60.46%	26.52%	32.38%	100%
OSA	75.36%	43.94%	**9,992**	99.99%	69.97%	50.81%	99.17%	100%
SPOON	75.81%	44.91%	98.08%	**10,187**	70.52%	49.86%	97.43%	100%
WALA	97.15%	39.32%	33.84%	34.77%	**20,660**	38.67%	33.55%	100%
DYN	93.21%	36.22%	51.62%	51.64%	81.23%	**9,836**	51.46%	94.62%
JDT	73.15%	42.60%	97.04%	97.20%	67.88%	49.57%	**10,211**	100%
UG	91.12%	35.84%	26.65%	27.17%	55.11%	24.82%	27.24%	**37,492**

a percentage format. In this comparison, we use these two terms equivalently.

Table 2 compares all static analyzers with each other, with the dynamic tool, and with UnionGraph as well. It can be seen that UnionGraph "finds" the most calls in the dynamic graph (94.62%), the second best is SootCHA with 93.21%. This result seems exceptionally good, however, note that SootCHA detects 34,161 edges, while the dynamic graph has only 10,211 edges. We can conclude that tools that do not use more advanced algorithms to deal with polymorphism other than simple static analysis (OSA, SPOON, JDT) have a relatively small coverage of the dynamic graph. WALA and SootCHA perform better. Although SootVTA uses the advanced VTA algorithm it achieves only 36.22% on the dynamic tool's graph. This is because Joda-Time is a library and the algorithm is entry-point-based.

With RQ1, we investigate whether it is possible to approximate dynamic call graphs with static call graphs only. Looking at the 93.21% result for SootCHA, the answer is yes. However, SootCHA found three times as many edges as the dynamic tool. The edge number explosion is caused by the CHA algorithm itself. If one of the test cases uses the `hashCode` method of a class, this call will appear as a single edge in the dynamic graph. In contrast, the CHA algorithm binds the `hashCode` method of every class in the class hierarchy. This can result in numerous call edges in the static graph that do not necessarily occur in reality.

RQ2 tests whether it is possible to cover the dynamic graph in such a way that the static graph is as accurate as possible. To answer this question, we used the combined graph described in Section V. Our goal was to construct a graph that contains as few edges as possible that are not present in the dynamic graph (more sound), but covers the edges that can be found (precise). The results of the graphs obtained by combining different static analyzers are summarized in Table 3. The combinations are SootCHA-WALA, SootCHA-SPOON, SPOON-WALA, and SootVTA-WALA. The table also includes the results of the original, unmodified graphs (WALA, OSA, SPOON, JDT, SootCHA, SootVTA, and UnionGraph). The Recall column is the link between Table 2 and Table 3 as the graph coverage we used is the percentage form of the recall. Since the precision value

TABLE 3. Precision, F1-score and Recall on Joda-Time.

Tool	Precision	F1	Recall
SootCHA-WALA	0.5242	0.6237	0.7696
SootCHA-SPOON	0.5411	0.6200	0.7258
SPOON-WALA	0.5822	0.5473	0.5164
WALA	0.3867	0.5240	0.8123
OSA	0.5081	0.5121	0.5162
SPOON	0.4986	0.5073	0.5164
JDT	0.4957	0.5050	0.5146
SootCHA	0.2684	0.4168	0.9321
UnionGraph	0.2482	0.3933	0.9462
SootVTA-WALA	0.4388	0.3744	0.3265
SootVTA	0.2652	0.3062	0.3622

is included in the graph comparison to answer RQ2, the F1-score[8] [38] was used as the primary measure, which is the harmonic mean of the precision and the recall. Therefore, the rows are in descending order by F1-score. We consider those edges of the static graphs to be true positives that are also found in the dynamic graph. Edges that are found only in the static graph are false positives, while edges that occur only in the dynamic are false negatives. The highest value in each column is highlighted in green. Naturally, UnionGraph sets an upper bound on the recall, so, in that column, it will always be the maximum.

It can be seen that the greatest F1-score results are obtained with combined graphs. The best is the filtered SootCHA-WALA combination. It has a 0.7696 recall on the dynamic call graph. Although this is lower than SootCHA's recall of 0.9321 (93.21% coverage), its precision is significantly higher (0.5242 instead of 0.2684, which is almost twice as high). The SPOON-WALA combination has the highest precision, but its recall value is relatively low. With this analysis, we found that it is possible to increase the accuracy of static graphs by heuristically combining them, but at the cost of a reduced recall.

1) QUALITATIVE ANALYSIS

Table 3 summarizes which static graphs best approximate the dynamic graph. Although the filtered SootCHA-WALA combination performed well, we checked which edges are missing from the static graph (false negative edges). The manual analysis revealed that the vast majority of these edges are associated with properties and Java language elements (e.g. synthetically generated access methods) that also significantly influence the construction of static graphs. These were described in more detail in our previous article [23]. Furthermore, the following two major groups of missing edges can be distinguished that are related to the dynamic execution. We refer to these calls as phantom calls, because, in reality, they are not executed in the form they are represented by the dynamic graph.

- Phantom calls from constructors (approximately 5% of all false positive edges)
- Phantom calls for `hashCode` (approximately 1.9% of all false positive edges)

Phantom calls from constructors are such calls that are present in the dynamic graph, but we have not been able to manually find a source code element for. Their occurrence is due to the operation of the JVM (class loading, optimizations).

Phantom calls for `hashCode` is a collective name. This refers to phantom calls that are caused by the dynamic tool's filtering mechanism. The dynamic tool filters out all Java library calls, so if there is a callback from the library to the analyzed code the intermediate calls will not be visible in the graph. For example, if a `foo` method of class `A` calls `HashMap`'s `add` method it will call `A`'s `hashCode` method, but because of the filtering, this will appear in the graph as if `foo` had called the `hashCode` method. Another example is the call of `equals` methods. In reality, these edges do not exist, so they do not appear in the static graphs.

These phantom edges are one of the main reasons why the recall value of UnionGraph is not 1.000, i.e. the graph coverage is not 100%. A dynamic graph has edges and nodes that static tools cannot find.

B. MAVEN-CORE

In this section, we analyze the Maven-Core project, which has a significantly lower test coverage (see Table 1) than the other two systems, to examine the effect of test coverage on the dynamic graph. The coverage between the static and dynamic graphs is shown in Table 4. Its structure is the same as in the case of Table 2.

The table shows that the dynamic graph contains very few, only 1,608 calls. In comparison, static analyzers find at least 4,000 edges. This significant difference was examined in the qualitative analysis to confirm that it was caused by the low test coverage. The effect of low test coverage can be clearly observed in the dynamic graph's column (column 7) as well. In this column, the values are significantly lower than in the case of Joda-Time. The dynamic graph covers only a very small part of the static graphs. For SootCHA it covers only 1.87% of the calls. This is not surprising, since the graph generated by the CHA algorithm contains 74,867 edges. However, despite this large number of edges in SootCHA's graph, it cannot cover 100% of the dynamic graph. It only covers 87% of it (the second column of the seventh row), which is a bit less than for Joda-Time. Although the dynamic graph can be considered small because of the low test coverage, it cannot be stated that this automatically ensures better coverage by static graphs. The coverage of the dynamic graph increased significantly for SootVTA, SPOON, and JDT compared to Joda-Time's results, but decreased for the others. The varied performance of static graphs can be explained by our previous article [23] on comparing static call graphs. In that study, we also evaluated the static call graph creator tools on Maven-Core and on Joda-Time. Although we used

[8]F1-score $= 2 \cdot \frac{precision \cdot recall}{precision + recall}$.

TABLE 4. Maven-Core comparison results.

	SootCHA	SootVTA	OSA	SPOON	WALA	DYN	JDT	UG
SootCHA	**74,867**	47.70%	3.01%	5.30%	5.65%	1.87%	5.44%	100%
SootVTA	99.96%	**35,725**	4.64%	7.53%	9.88%	2.68%	7.85%	100%
OSA	47.50%	34.87%	**4,749**	64.31%	40.64%	12.44%	61.21%	100%
SPOON	54.91%	37.21%	42.24%	**7,230**	34.20%	13.57%	80.17%	100%
WALA	99.51%	83.12%	45.43%	58.22%	**4,248**	20.43%	54.26%	100%
DYN	87.00%	59.51%	36.75%	61.01%	53.98%	**1,608**	62.19%	87.13%
JDT	58.19%	40.11%	41.56%	82.87%	32.96%	14.30%	**6,994**	100%
UG	92.99%	44.37%	5.90%	8.98%	5.28%	1.74%	8.69%	**80,508**

TABLE 5. Precision, F1-score and Recall on Maven-Core.

Tool	Precision	F1	Recall
SootCHA-WALA	0.2544	0.3937	0.8700
SootCHA-SPOON	0.2540	0.3932	0.8694
SootVTA-WALA	0.2261	0.3277	0.5951
SPOON-WALA	0.1993	0.3004	0.6101
WALA	0.2043	0.2964	0.5398
JDT	0.1430	0.2325	0.6219
SPOON	0.1357	0.2220	0.6101
OSA	0.1244	0.1859	0.3675
SootVTA	0.0268	0.0513	0.5951
SootCHA	0.0187	0.0366	0.8700
UnionGraph	0.0174	0.0341	0.8713

version 3.6.0 instead of the currently used 3.6.3, the trend that Maven-Core brings out the differences in static analyzers much better than Joda-Time was already apparent.

Table 5 shows the results of the combined graphs as in Table 3. These values are significantly lower than for Joda-Time because of the high number of false positive edges (i.e. edges that are only found in the static tools). Once again, the SootCHA-WALA pairing achieves the best F1-score. It reaches a 0.8700 recall on the dynamic graph, which is really close to the upper bound, the 0.8713 recall of UnionGraph.

1) QUALITATIVE ANALYSIS

Since static graphs do not always cover dynamic graphs as precisely as would be expected given the large edge difference, it was important to investigate which edges occur only in dynamic graphs and only in static graphs. First, we performed a manual check of the false negative edges (edges only in the dynamic graph). We found types similar to what Joda-Time has, i.e. 10% of the false negative calls were phantom calls for `hashCode` and another 10% were connected to synthetic methods (access methods). Moreover, we detected new types of phantom edges. The Maven project uses the Mockito Framework[9] for testing, which generates stubs and mock classes. Since the dynamic tool gets the bytecode from the class loader itself (online instrumentation), it will find these generated classes and their methods, so nodes that do not exist in the source code are added to the graph. Naturally, static analyzers cannot find these nodes and edges. Among the false negative calls, 25% of them were connected to the Mockito Framework. Other libraries and frameworks may also generate classes at runtime.

We also examined the false positive edges, i.e. the edges that occur only in static graphs. The study confirmed that the size difference in graphs is caused by those parts of the code that are not executed by the tests. That is, test coverage has a strong influence on the dynamic call graphs. In such cases, statically generated graphs give a better picture of the project. This answers RQ3, i.e. that dynamic graphs cannot be considered as the ''golden standard'' for call graphs.

C. SORTPOM SORTER

In addition to the two large projects, we also analyzed a small one, the Sorter module of the Sortpom project. Given its exceptionally high branch coverage, we expected results similar to that of Joda-Time. Table 6 confirms this assumption. The column of the dynamic graph shows that it also covers static graphs better than Maven-Core, although, in most cases, the extent of this is lower than for Joda-Time. Similar to the Joda-Time results (see Table 2) SootCHA and WALA achieve the highest coverage on the dynamic graph. UnionGraph reaches a score above 90%, but the other static graphs cover it reasonably well too. JDT is the only exception. The reason for this difference is that the coverage of methods is only 43.07%. In contrast, the second worst method coverage value (achieved by OSA) is 74.91%. JDT has 413 nodes in its graph, but when the Java library calls are removed, only 169 remain. This is less than the 267 methods found by the dynamic analyzer. When the impact of low test coverage is added to this, the graph coverage of only 16.91% becomes understandable.

Table 7 shows the F1-score and precision results on the Sortpom project. The greatest precision values are around 0.67, which matches what we expected based on the Jode-Time project perfectly. The highest F1-score is 0.7638, which even exceeds Joda-Time. It was achieved by the SootCHA-WALA combinatorial graph. Its recall (0.8863) is very close to its upper bound (0.9040). This characteristic is also similar to the results on Joda-Time.

1) QUALITATIVE ANALYSIS

Due to the large test coverage, it may seem surprising in Table 6 that the dynamic graph's edge count is significantly lower than the static ones'. While the dynamic graph has only 267 nodes and 343 edges, UnionGraph has 1,826 nodes

[9]https://site.mockito.org/

TABLE 6. Sortpom-Sorter comparison results.

	SootCHA	SootVTA	OSA	SPOON	WALA	DYN	JDT	UG
SootCHA	**2,265**	56.38%	15.41%	19.12%	29.01	13.69%	6.89%	100%
SootVTA	100%	**1,277**	24.35%	30.54%	48.63%	22.47%	11.67%	100%
OSA	22.83%	20.34%	**1,529**	25.70%	21.12%	11.45%	5.10%	100%
SPOON	75.83%	68.30%	68.83%	**571**	67.95%	40.28%	14.71%	100%
WALA	94.94%	89.74%	46.68%	56.07%	**692**	42.20%	23.55%	100%
DYN	90.38%	83.67%	51.02%	67.06%	85.13%	**343**	16.91%	90.38%
JDT	22.22%	21.23%	11.11%	11.97%	23.22%	8.26%	**702**	100%
UG	55.10%	31.06%	37.19%	13.89%	16.83%	7.54%	17.08%	**4,111**

TABLE 7. Precision, F1-score and Recall on Sortpom-Sorter.

Tool	**Precision**	**F1**	**Recall**
SootCHA-WALA	0.6711	0.7638	0.8863
SootCHA-SPOON	0.6734	0.7620	0.8776
SootVTA-WALA	0.6778	0.7454	0.8280
SPOON-WALA	0.6785	0.6745	0.6706
WALA	0.4220	0.5643	0.8513
SPOON	0.4028	0.5033	0.6706
SootVTA	0.2247	0.3543	0.8367
SootCHA	0.1369	0.2377	0.9038
OSA	0.1145	0.1870	0.5102
UnionGraph	0.0754	0.1392	0.9038
JDT	0.0826	0.1110	0.1691

and 4,111 edges. UnionGraph covers all the methods of the dynamic graph, so there are 1,559 methods that only occur in static graphs. It is worth examining which methods and calls are missing from the dynamic graph even with such high test coverage. Out of the 1559 methods, only 615 are strictly related to the Sortpom project. This is a huge difference compared to the dynamic graph, from which library calls are filtered. The next large category of discrepancies is caused by the test functions (349 methods). Although they are part of the Sortpom project, they are also filtered from the dynamic graph. The remaining methods include 69 lambda or nested class methods, which can be difficult to pair if the analyzers do not provide the correct line information [32].

It is also crucial to investigate which edges the static tools could not cover in the dynamic graph, i.e. why the recall is not 1.000, or in other words, why the SootCHA-WALA combination achieved a recall of only 0.8863. In this case, the uncovered dynamic calls' number was 39. The greatest part, 51% of these calls were phantom calls generated by the dynamic tool. An example of a phantom call is the edge pointing from method `processElement(Wrapper)` of class `ToStringOperation` of package `sortpom.wrapper.operation` to the `toString()` method of class `AlphabeticalSortedWrapper` of package `sortpom.wrapper.content`. In the source code, this line of code belongs to this call: `builder.append(baseIndent).append("elementContent=").append(element Wrapper).append("\n");` According to the static analysis, there is no `toString` call because `append` methods return with the `StringBuilder` object.

About 31% of the calls were connected to language-specific elements that are handled by static tools with varying efficiency and in different ways. An example of a language-specific call is the call from the `processOperation(HierarchyWrapperOperation)` method of the `HierarchyWrapper` class to the `processOtherContent(Wrapper)` method of the `HierarchyWrapperOperation` class. (Both classes are part of the `sortpom.wrapper.operation` package.) In the actual source code, this is a method reference provided for a `forEach` method. Soot generated a completely new bootstrap method for the method reference.

15% of the missing edges were removed during our heuristic combination mechanism when polymorphic edges were filtered out to handle the CHA explosion.

There was one call out of 39 that could not be matched to either the source code or the previous categories. It is a call from the `lambda$compareTo$3(Function,Map.Entry)` method of the `ChildElementSorter` class to its `lambda$compareTo$2(ChildElementSorter,Map.Entry)` method. The `ChildElementSorter` class is part of the `sortpom.wrapper.content` package, the `Map.Entry` and `Function` classes are part of the `java.util` package. The elements such as `compareTo`, `Function`, and `Map$Entry` are present in the source code, but it is unclear how the Java compiler and the JVM created this exact call.

VII. THREATS TO VALIDITY

We used one concrete dynamic analyzer in this work. We did not include any other dynamic analyzers, however, we studied them in detail. We found that they use the same API provided by JVM and they differ only in method filtering. The filtering mechanism of the dynamic tool we used is not limited to the fully qualified names of the methods, it employs the actual location on the disk as well. This way, it provides a more sophisticated way of filtering, for example, it can exclude packages from other submodules and test classes.

In this paper, we included only 3 Java projects and evaluated our heuristic algorithm on these. Although the projects were carefully selected to be different in size and test coverage, other projects might have produced a bit different graph-combination results. This is natural, because, as we have shown in a previous article [23], the efficiency of static call graph generators is greatly influenced by the language elements used by the project. Analyzing a project should start with analyzing the most used language features and choosing static analyzers accordingly.

The analyzed projects are libraries, meaning that they do not have explicit entry points in the code. This could have influenced the sophisticated call graph constructor algorithms such as Soot's VTA or WALA's *ZeroOneContainer-CFA* algorithm. However, this is not an issue, considering that the aim of this paper was not to compare different methods or to find the best possible combination of static analyzers but to investigate whether we could create a combination that may substitute dynamic call graphs. Similarly, the parameter of the heuristic combination algorithm was considered constant. We were not aiming for the optimal solution.

VIII. SUMMARY

In this research, we aimed to investigate the relationship between call graphs produced by dynamic and static analysis. We collected 5 open-source static analyzers. We used two configurations of Soot, thus we compared a total of 6 static call graphs with the dynamically generated call graph. The dynamic graph was created using a Java agent-based, on-the-fly bytecode instrumenter tool. This instumenter was executed on the test suite of the three Java projects used in the comparison, which have varying test coverages and sizes. In order to compare the dynamic and static graphs, we asked three research questions. The first two investigated whether static graphs can be used to approximate dynamic graphs. In the third RQ, we examined whether dynamic graphs can be considered superior to static graphs, i.e. whether they can be a "golden standard" for call graphs. The summary of the answers to the three research questions are the following:

- **RQ1 - Is it possible to approximate dynamic call graphs using only static call graphs?**: The three comparison tables (Tables 2, 4, and 6) show that the edges of the dynamic graph can be covered by static graphs up to approximately 90%. The best stand-alone static analyzer is Soot configured with the CHA algorithm. Naturally, the best coverage was achieved by UnionGraph, which contains all edges of all static graphs. This is the upper limit of coverage. Incomplete coverage is caused by edges that can only be found by the dynamic analyzer. The incomplete coverage is partly caused by edges that can only be found by the dynamic analyzer because they are caused, for example, by the operation of the JVM. These are called phantom edges. The other reason for the discrepancy is the way static analyzers work, which we discussed in our previous article [23]. With a high coverage of around 90%, it can be said that static graphs are suitable for approximating dynamic call graphs. However, the cost is that the size of the static graph is often several times larger than the dynamic graph. In the case of Joda-Time, the UnionGraph contains 37,492 edges, compared to 9,836 in the dynamic graph. For this reason, we examined in RQ2 whether good coverage can be achieved with static graphs that contain as few edges as possible.
- **RQ2 - Is it possible to cover the dynamic call graphs using static call graphs but with more soundness?**: To answer this question, we heuristically combined several static graphs and compared them to the dynamic graph. The aim of this heuristic combination was to reduce the number of unnecessary edges in static graphs. The graph of SootCHA was a good starting point since it covered the dynamic graph the best, but the CHA algorithm introduced a large number of polymorphic edges into the graph that might not be executed in reality. If a so-called CHA explosion is detected we delete every call in SootCHA's graph and substitute it with another tool's calls from the same call site. We tested several tool combinations, some did not include SootCHA. The F1-score was used to evaluate the performance of the combinations, as it is the harmonic mean of accuracy and recall (to which graph coverage can be related). In all of the cases, the SootCHA-WALA combination provided the best results. Although their dynamic graph coverage was reduced compared to the original SootCHA, the accuracy of the graphs was significantly increased. RQ2 can also be answered positively, i.e. it is possible to cover dynamic graphs with higher accuracy and without a significant decrease in recall.
- **RQ3 - Can dynamic call graphs be considered a "golden standard" for call graphs?**: To answer this question, we performed a qualitative analysis to find out which edges occur only in dynamic graphs and only in static graphs. The dynamic graph may miss edges because of insufficient test coverage. JVM actions, phantom calls, and improper filtering may cause the occurrence of edges that are not present in the project according to the source code. On the other hand, static graphs might contain unnecessary edges that are not executed in reality (e.g. due to the handling of polymorphism) and they miss some edges due to inaccuracies in their analysis. Based on these results, we concluded that neither dynamic graphs nor static graphs can be considered the "golden standard". It depends on the subsequent application which is more suitable for the purpose. Static call graphs might be better if there is no test suite or it is negligible. They might also be preferred if an excessive number of call-backs are present in the code that create the mentioned phantom calls.

REFERENCES

[1] L. A. Zager and G. C. Verghese, "Graph similarity scoring and matching," *Appl. Math. Lett.*, vol. 21, no. 1, pp. 86–94, Jan. 2008.

[2] K. Ali and O. Lhoták, "Application-only call graph construction," in *Proc. 26th Eur. Conf. Object-Oriented Program. (ECOOP)*, Berlin, Germany: Springer-Verlag, 2012, pp. 688–712.

[3] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. thesis, Dept. Comput. Sci., Univ. Copenhagen, Copenhagen, Denmark, 1994.

[4] G. Antal, P. Hegedus, Z. Toth, R. Ferenc, and T. Gyimothy, ''Static Javascript call graphs: A comparative study,'' in *Proc. IEEE 18th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2018, pp. 177–186.

[5] D. F. Bacon and P. F. Sweeney, ''Fast static analysis of C++ virtual function calls,'' *ACM SIGPLAN Notices*, vol. 31, no. 10, pp. 324–341, Oct. 1996.

[6] P. Badenski. (2022). *Call Hierarchy Printer GitHub Page*. Accessed: 2022. [Online]. Available: https://github.com/pbadenski/call-hierarchy-printer

[7] CallGraphViewer. (2022)/. *Callgraph Viewer Home Page*. Accessed: 2022. [Online]. Available: https://marketplace.eclipse.org/content/callgraph-viewer

[8] X. Chen, P. Du, and W. Srisaan, ''SimSight: A virtual machine based dynamic call-graph generator,'' Dept. Comput. Sci. Eng., Univ. Nebraska-Lincoln, Tech. Rep. TR-UNL-CSE-2010-0010, 2010.

[9] M. Christodorescu and S. Jha, ''Static analysis of executables to detect malicious patterns,'' in *Proc. 12th Conf. USENIX Secur. Symp. (SSYM)*, vol. 12. Berkeley, CA, USA: USENIX Association, 2003, p. 12.

[10] D. V. Bruggen, F. Tomassetti, N. Smith, and C. Maximilien. (2022). *Java-Parser for Processing Java Code Homepage*. Accessed: 2022. [Online]. Available: https://javaparser.org/

[11] J. Dean, D. Grove, and C. Chambers, ''Optimization of object-oriented programs using static class hierarchy analysis,'' in *Proc. Eur. Conf. Object-Oriented Program.*, M. Tokoro and R. Pareschi, Eds. Berlin, Germany: Springer, Aug. 1995, pp. 77–101.

[12] DSE University of Szeged. (2022). *OpenStaticAnalyzer GitHub Page*. Accessed: 2022. [Online]. Available: https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer

[13] Eclipse. (2022). *Eclipse Home Page*. Accessed: 2022. [Online]. Available: www.eclipse.org/eclipse/

[14] Eclipse JDT. (2022). *Eclipse JDT Home Page*. Accessed: 2022. [Online]. Available: http://www.eclipse.org/jdt/

[15] F. Eichinger, K. Böhm, and M. Huber, ''Mining edge-weighted call graphs to localise software bugs,'' in *Machine Learning and Knowledge Discovery in Databases*. Berlin, Germany: Springer, 2008, pp. 333–348.

[16] Y. Feng, S. Anand, I. Dillig, and A. Aiken, ''Apposcopy: Semantics-based detection of Android malware through static analysis,'' in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 576–587.

[17] N. Grech, G. Fourtounis, A. Francalanza, and Y. Smaragdakis, ''Heaps don't lie: Countering unsoundness with heap snapshots,'' 2019, *arXiv:1905.02088*.

[18] D. Grove and C. Chambers, ''A framework for call graph construction algorithms,'' *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 6, pp. 685–746, Nov. 2001. [Online]. Available: http://doi.acm.org/10.1145/506315.506316, doi: 10.1145/506315.506316.

[19] D. Grove, G. DeFouw, J. Dean, and C. Chambers, ''Call graph construction in object-oriented languages,'' in *Proc. 12th ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, 1997, pp. 108–124.

[20] H. Hoogendorp, ''Extraction and visual exploration of call graphs for large software systems,'' M.S. thesis, Faculty Math. Natural Sci., Univ. Groningen, Groningen, The Netherlands, 2010.

[21] F. Horváth, T. Gergely, Á. Beszédes, D. Tengeri, G. Balogh, and T. Gyimóthy, ''Code coverage differences of Java bytecode and source code instrumentation tools,'' *Softw. Quality J.*, vol. 27, no. 1, pp. 79–123, Mar. 2019.

[22] R. Just, D. Jalali, and M. D. Ernst, ''Defects4J: A database of existing faults to enable controlled testing studies for Java programs,'' in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2014, pp. 437–440.

[23] J. Jász, I. Siket, E. Pengő, Z. Ságodi, and R. Ferenc, ''Systematic comparison of six open-source Java call graph construction tools,'' in *Proc. 14th Int. Conf. Softw. Technol.*, 2019, pp. 117–128.

[24] T. Kempf, K. Karuri, and L. Gao, ''Software instrumentation,'' in *Wiley Encyclopedia of Computer Science and Engineering*, B. W. Wah, Ed. Hoboken, NJ, USA: Wiley, 2008.

[25] D. Koutra, A. Parikh, A. Ramdas, and J. Xiang, ''Algorithms for graph similarity and subgraph matching,'' Tech. Rep., 2011.

[26] O. Lhoták, ''Comparing call graphs,'' in *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2007, pp. 37–42.

[27] C. Liu, X. Yan, H. Yu, J. Han, and S. P. Yu, ''Mining behavior graphs for 'Backtrace' of noncrashing bugs,'' in *Proc. SDM*, 2005, pp. 286–297.

[28] O. Macindoe and W. Richards, ''Graph comparison using fine structure analysis,'' in *Proc. IEEE 2nd Int. Conf. Social Comput.*, Aug. 2010, pp. 193–200.

[29] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, ''An empirical study of static call graph extractors,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 2, pp. 158–191, Apr. 1998.

[30] V. Musco, M. Monperrus, and P. Preux, ''A large-scale study of call graph-based impact prediction using mutation testing,'' *Softw. Quality J.*, vol. 25, no. 3, pp. 921–950, Sep. 2017.

[31] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, ''SPOON: A library for implementing analyses and transformations of Java source code,'' *Software: Pract. Exper.*, vol. 46, no. 9, pp. 1155–1179, Sep. 2016.

[32] E. Pengő and Z. Ságodi, ''A preparation guide for Java call graph comparison: Finding a match for your methods,'' *Acta Cybernetica*, vol. 24, no. 1, pp. 131–155, 2019.

[33] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini, ''Call graph construction for Java libraries,'' in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 474–486.

[34] H. G. Rice, ''Classes of recursively enumerable sets and their decision problems,'' *Trans. Amer. Math. Soc.*, vol. 74, no. 2, pp. 358–366, 1953.

[35] B. G. Ryder, ''Constructing the call graph of a program,'' *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 3, pp. 216–226, May 1979.

[36] Sable *J. (2019). *Sable *J Home Page*. Accessed: 2022. [Online]. Available: http://www.sable.mcgill.ca/starj/

[37] Sable Research Group. (2022). *Sable/Soot GitHub Page*. Accessed: 2022. [Online]. Available: https://github.com/Sable/soot

[38] Y. Sasaki, ''The truth of the F-measure,'' *Teach Tutor Mater*, vol. 1, no. 5, pp. 1–15, Jan. 2007.

[39] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, ''On the recall of static call graph construction in practice,'' in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 1049–1060.

[40] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, ''Practical virtual method call resolution for Java,'' *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 264–280, Oct. 2000.

[41] D. Tengeri, F. Horvath, A. Beszedes, T. Gergely, and T. Gyimothy, ''Negative effects of bytecode instrumentation on Java source code coverage,'' in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2016, pp. 225–235.

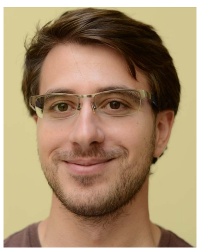
[42] F. Tip and J. Palsberg, ''Scalable propagation-based call graph construction algorithms,'' in *Proc. 15th ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, 2000, pp. 281–293.

[43] J. R. Ullmann, ''An algorithm for subgraph isomorphism,'' *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.

[44] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, ''Accurate static estimators for program optimization,'' *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 85–96, Jun. 1994.

[45] WALA. (2022). *WALA Home Page*. Accessed: 2022. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page

[46] K. Gallagher and D. Binkley, ''Program slicing,'' in *Proc. Frontiers Softw. Maintenance*, Sep. 2008, pp. 439–449.



ZOLTÁN SÁGODI is currently pursuing the Ph.D. degree. He has been working in research during his B.Sc. and M.Sc. studies. His Ph.D. topic is detecting vulnerabilities and faults in source code via static analysis and AI application.

He is currently working at the Department of Software Engineering, University of Szeged. Besides his research tasks, he takes his part in education. This means multiple courses and in his freetime, he takes effort into automatizing many of the educational tasks (e.g., creating and correcting exams, and preparation for the courses).

EDIT PENGŐ received the Ph.D. degree from the University of Szeged, in 2016. Her research interests include the quality assurance of source code through static analysis. Topics of her thesis include symbolic execution, comparison of static call graphs, and detection of primitive obsession code smell. The thesis defense process is currently underway. Besides research, she currently works at the Department of Software Engineering, University of Szeged, and takes part in research and development projects. She teaches C++ programming and information security.

JUDIT JÁSZ received the Ph.D. degree in computer science from the University of Szeged, in 2010. She is currently an Assistant Professor with the Department of Software Engineering, University of Szeged. Her research interests include static program analysis and bug prediction. In addition to research, she is actively involved in the department's teaching activities and research development projects.

ISTVÁN SIKET received the Ph.D. degree in computer science, in 2011. He is currently an Assistant Professor with the Department of Software Engineering, University of Szeged. His research interests include source code analysis, measurement, quality assurance, and bug detection. He has been participating in several research and development projects related to source code analysis and quality assurance.

RUDOLF FERENC received the Ph.D. degree in computer science from the University of Szeged, in 2005, and the Habilitation degree, in 2015.

He is currently an Associate Professor and acting as the Head of the Department of Software Engineering, University of Szeged. His research interests include static code analysis, metrics, quality assurance, design pattern and antipattern mining, and bug detection. He leads the Static Code Analysis Group, which develops tools for analyzing the source code of various languages. These tools calculate code metrics and detect coding issues and duplications. He has more than 100 publications in these fields with over 2000 citations. He is leading several research and development projects, which are related to quality assessment, improvement, and architecture reconstruction of software systems for major banks and software development companies in Hungary. He has been serving as the Program Co-Chair and the Program Committee Member at the major conferences in this field (ICSE, ICSME, ESEC/FSE, SANER, CSMR, WCRE, ICPC, SCAM, and FASE), since 2005.