Qualitative and Quantitative Analysis of Callgraph Algorithms for PYTHON

Sriteja Kummita
sriteja.kummita@iem.fraunhofer.de
Fraunhofer IEM
Germany

Goran Piskachev
goran.piskachev@iem.fraunhofer.de
Fraunhofer IEM
Germany

Johannes Späth
johannes.spaeth@codeshield.io
Department of Computer Science, Paderborn
University
CodeShield GmbH
Germany

Eric Bodden
eric.bodden@upb.de
Department of Computer Science, Paderborn
University
Fraunhofer IEM
Germany

Abstract

As one of the most popular programming languages, PYTHON has become a relevant target language for static analysis tools. The primary data structure for performing an inter-procedural static analysis is callgraph (CG), which links call sites to potential call targets in a program. There exists multiple algorithms for constructing callgraphs, tailored to specific languages. However, comparatively few implementations target PYTHON. Moreover, there is still lack of empirical evidence as to how these few algorithms perform in terms of precision and recall.

This paper thus presents eval.CG, an extensible framework for comparative analysis of PYTHON callgraphs. We conducted two experiments which run the CG algorithms on different PYTHON programming constructs and real-world applications. In both experiments, we evaluate three CG generation frameworks namely, CODE2FLOW, PYAN, and WALA. We record precision, recall, and running time, and identify sources of unsoundness of each framework.

Our evaluation shows that none of the current CG construction frameworks produce a sound CG. Moreover, the static CGs contain many spurious edges. CODE2FLOW is also comparatively slow. Hence, further research is needed to support CG generation for PYTHON programs.

Keywords: Static Analysis, Callgraph Analysis, Python, Qualitative Analysis, Quantitative Analysis, Empirical Evaluation

1 Introduction

A callgraph (CG) is the fundamental data structure that enables inter-procedural static program analysis and has been used as baseline for static client analyses over the last decades [18]. Using a CG, one can perform client analyses such as identification of unreachable methods, replacing dynamically dispatched function calls with their static counterparts [4], performing inter-procedural constant propagation [26], method inlining, or refactorings based on the class hierarchy [7]. The CG nodes represent functions from the given program whereas the edges model the function invocations (aka. call sites) to the respective functions (aka. call targets). The quality of the results of any inter-procedural client analyses heavily depends upon the precision and soundness of the generated CG. A CG is sound when it includes edges to all possible runtime call targets and is precise when it does not include edges to the call targets that are not possible at runtime. A perfectly accurate CG includes all runtime edges that are possible from every call site and, at the same time, no further spurious edge. However, the construction of such an accurate static CG is undecidable [24] and CG algorithms must resort to static approximations.

Despite the fact that PYTHON has become a very popular programming language, research on inter-procedural static analysis for PYTHON is rare. PYTHON’s wide-range of dynamic language features challenge the generation of static callgraphs (CGs). While in statically-typed languages such as Java, dynamic features reduce to dynamic dispatch and reflection, mostly

1 TIOBE index - https://www.tiobe.com/tiobe-index/
used internally by frameworks, in Python dynamic language features are used more widely. Python’s dynamic language instruction set contains many features, duck typing, and eval() to name a few.

Despite these challenges, several open-source frameworks (such as WALA [29], and PyAN [20]) have become available that support static code analysis for Python. Hence, static analysis writers have the choice of several frameworks that support also different CG construction algorithms. To build meaningful client analyses for Python programs, static analysis writers need to understand the limitations of these algorithms and their implementations.

To map this landscape, this paper presents a systematic evaluation of different CG construction algorithms, with the goal to help analysis writers to select the algorithm appropriate for their needs. The evaluation covers the three frameworks, namely, CODE2FLOW [5], PyAN [20] and WALA. To allow others to evaluate also future frameworks, we make our work publicly available at https://github.com/sritejakv/eval_CG.

For our evaluation we took a similar approach as Reif et al. [23], who published study for Java CG. Our evaluation includes a qualitative comparison of the CGs to identify the sources of unsoundness (aiming to strengthen internal validity) and a quantitative comparison on real-world programs to measure overall performance in terms of soundness and running time (to strengthen external validity). The evaluation methodology includes two experiments, namely, Synthetic test, and Real-World test. Synthetic test runs the CG algorithms on a benchmark suite of Python programs, and Real-World test runs the algorithms on real-world applications. The CG Evaluation Framework in each of the experiments compares the static CG edges with dynamic CG edges and records precision, recall, as well as runtime information.

Our evaluation shows that none of the current CG construction frameworks produce a sound CG. Moreover, the static CGs contain many spurious edges. CODE2FLOW is also comparatively slow. Hence, further research is needed to support CG generation for Python programs. Among the three frameworks, PyAN and WALA perform better on individual, synthetic Python test cases. PyAN performs best in the case of real-world applications. Yet, all three algorithms virtually yield results that will be unacceptable for most use cases.

This paper makes the following original contributions:

- **eval_CG**, a CG evaluation framework for Python consisting of Synthetic test for qualitative evaluation and Real-World test for quantitative evaluation,
- a Python benchmark suite consisting of 49 programs separated into 13 categories that cover the language features, and
- adapters for the frameworks: CODE2FLOW, PyAN, and WALA and their evaluation with eval_CG.

The remainder of the paper is organised as follows. Section 2 discusses the challenges in generating a CG for Python programs. Section 3 explains the methodology of evaluation. Section 4 explains the two experiments, namely, Synthetic test and Real-World test. Section 5 presents the results. Finally, Section 6 discusses related works and Section 7 concludes the paper.

2 Challenges in static callgraph construction for Python

It is difficult to analyze Python programs statically, i.e., without executing them, due to its dynamic features that cause difficulties in the generation of a precise and sound callgraph (CG). Python is a strongly, dynamically typed language, which means the runtime objects have types and can vary over the course of the execution of the program. It is both compiled and interpreted language [17].

We explain the challenges in constructing static CG for Python through some of its dynamic capabilities shown in the example in Listing 1. The listing shows a simple calculator program which can perform sum, sin, cos, and tan operations on numbers. It contains a function check_values (line 1) and a class Calculator (line 9).

The function check_values contains another nested function decorated_wrapper (at line 2), which validates if the first two items in the arguments are numbers (at line 4). This function serves as a decorator. Decorators in Python allow to modify the behaviour of an object at runtime. When there is a function call to a decorated function, the implementation in the decorator is executed first, and then the control is shifted to the actual function. The function init_values is decorated with check_values (at line 10) to allow only numbers as the parameters.

The class Calculator consists of three user-defined functions, init_values, calculate_sum, calculate_scientific, and an overridden function, __getattr__. init_values initializes the two class variables x and y.
Figure 1. The dynamic CG and the static CGs constructed by WALA, and PYAN for Listing 1

The method calculate_sum prints the sum of the two class variables. calculate_scientific performs scientific calculations such as sin, cos, and tan by generating a function during runtime using in-built Python function exec. The string variable, embedded (line 20), contains a template that generates a function. During runtime, if self.operation class variable contains the value sin, then a function with name calculate_sin (line 22) is generated which invokes sin function of math library (line 23). PYTHON also allows runtime code generation using eval.

Reflection is the ability to modify or change the structure and behaviour of an object at runtime. PYTHON has the ability to invoke a function that is not defined in its corresponding class. During runtime, PYTHON invokes __getattr__ method of the object’s class, when it encounters access to an undefined attribute. In this example, we override __getattr__ method to allow calls to three undefined functions namely, calculate_sin, calculate_cos, and calculate_tan (at line 30).

Listing 2 shows the main function of the above program. It contains two function invocations on the Calculator’s object, namely, calculate_sum and calculate_cos.

Let us first inspect the dynamic CG used at runtime. Figure 1 presents the program’s dynamic CG as a directed graph (solid edges). The graph contains a total of eight edges and nine nodes.

Figure 1 also presents the edges generated by the frameworks PYAN and WALA. PYAN generates three of them (shown with blue-dashed line) whereas WALA generates only two of them (shown with red-dotted line) that are present in the dynamic CG. Code2Flow does not generate any edges for the above example. These missing edges may largely impact the quality of any static client analysis.

3 EVAL_CG: Callgraph Evaluation Framework

For the evaluation of PYTHON callgraphs (CGs), we present EVAL_CG, a CG evaluation framework. EVAL_CG enables a qualitative and a quantitative analysis
of the supplied CG algorithms for an in-depth comparison. Our framework accepts a program along with its tests cases as input and extracts the dynamic CG by executing the program using the test cases. The CG is then used as ground truth for comparison metrics against statically generated CG that is supplied as additional input to the framework. Figure 2 shows the component diagram of \texttt{EVAL\_CG}. The framework consists of three main components, \texttt{Dot2probe}, \texttt{Framework Adapter}, and \texttt{Call Graph Analyzer}.

\textbf{Dot2probe}. Each of the CG generation frameworks has its own CG format, which makes the comparison a tedious task. \texttt{EVAL\_CG} converts all the static and dynamic CGs to a single format. We chose to reuse Lhotak’s “probe” format [15], a data structure specifically developed for comparing CGs. It also provides look-up abilities to examine the CGs at a fine-granular level. The nodes of the probe CG represent the function names and the edges represent the function calls. Since, the CGs of \texttt{PYAN}, \texttt{CODE2FLOW}, and \texttt{PyCallGraph} are in the dot format [13], we implemented the corresponding probe-converters in this component.

\textbf{Framework Adapter}. We have implemented three framework adapters, one for each corresponding CG generation framework, namely, \texttt{PYAN}, \texttt{CODE2FLOW}, and \texttt{WALA}. These adapters are housed in the component, \texttt{Call Graph Analyzer}. Each framework adapter is responsible for converting the source CG to probe CG and provide them to the \texttt{Call Graph Analyzer} for comparison. \texttt{PYAN} and \texttt{CODE2FLOW} adapters use the converters from \texttt{dot2probe}. Since the CG format of \texttt{WALA} is different from \texttt{PYAN}, and \texttt{CODE2FLOW}, we have implemented the conversion of \texttt{WALA} static CG to probe CG in its framework adapter.

\textbf{Call Graph Analyzer} This is the central component of \texttt{EVAL\_CG} that houses other components. It initiates the evaluation process by generating static and dynamic CGs for the given input and providing them to framework adapters to get corresponding probe CGs. It then uses the class, \texttt{ExtractMetrics}, to compare the CGs. \texttt{ExtractMetrics} checks each edge in the static probe CG against each edge in the dynamic probe CG to record the metrics. Though \texttt{Probe} provides comparison and inspecting tools for CGs, they are mostly focused on finding the differences. Our approach also focuses on identifying the sources of unsoundness and the quantitative comparison. So, we only adapt the CG format from \texttt{Probe} and implement the comparison suiting our needs.

The dynamic CG is generated using \texttt{PyCallGraph} [21]. This component runs \texttt{PyCallGraph} on the input \texttt{PYTHON} program or alongside the test cases in the case of input \texttt{PYTHON} application to generate the dynamic CG. \texttt{PyCallGraph} uses \texttt{sys.set_trace()} function, which is called by \texttt{PYTHON} every time when the execution flow enters or exits a function to record the CG [21].

We use the established metrics precision, and recall for the evaluation of the quality of CGs. All metrics are based on the definition of true positives (TPs), false positives (FPs), and false negatives (FNs). We consider the dynamic CG as ground truth. The definitions of these terms is then as follows.

A \textit{TP} is a CG edge that is present in the static and the dynamic CG. A \textit{FP} is an edge that is only present in the static CG. A \textit{FN} is an edge that is only present in the dynamic CG.

Using the above terms, the \textit{precision} of a CG algorithm is defined as the ratio of the total number of static CG edges present in the dynamic CG (as shown in Equation 1). The \textit{recall} is defined as the ratio of the total number of dynamic CG edges present in the static CG (as shown in Equation 2).
4 Experimental Setup

In this section, we discuss our experiments for the qualitative and quantitative analysis [32] of static callgraphs (CG) for Python programs. For both analyses, we used eval_CG introduced in the previous section with different set of input programs. Three CG generation frameworks are considered, namely, Code2flow [5], Pyan [20], and WALA [29]. Table 1 shows the different CG algorithms of each framework.

Table 1. CG algorithms supported by Code2flow, Pyan, and WALA.

<table>
<thead>
<tr>
<th>Framework</th>
<th>CG algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code2flow</td>
<td>C2F_CG</td>
</tr>
<tr>
<td>Pyan</td>
<td>Pyan_CG</td>
</tr>
<tr>
<td>WALA</td>
<td>nCFA, VanillaZeroOneCFA, ZeroCFA, ZeroContainerCFA, ZeroOneCFA</td>
</tr>
</tbody>
</table>

4.1 Qualitative analysis

For this analysis we performed an experiment named Synthetic test in which the input to eval_CG is a benchmark suite as shown in the Figure 3. The benchmark suite consists of Python programs where each one showcases different Python language construct. For each program from the suite we create the static CG and compare it against the dynamic CG. The dynamic CG is generated using the tool, PyCallGraph [21]. It runs each file in the benchmark suite and records the dynamic CG when the program is being executed. This dynamic CG is considered as the ground truth in the comparison. Next we discuss our benchmark suite.

4.2 Benchmark suite

Our benchmark suite consists of 49 Python programs segregated into 13 different categories, which we created based on the literature [1, 11] and the Python specification [22]. Table 2 shows the different categories in the benchmark suite along with the number of programs in each category.

Table 2. Benchmark categories with the number of programs

<table>
<thead>
<tr>
<th>Category</th>
<th>No. of programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime code generation</td>
<td>3</td>
</tr>
<tr>
<td>Decorators</td>
<td>5</td>
</tr>
<tr>
<td>Object changes</td>
<td>4</td>
</tr>
<tr>
<td>Static functions</td>
<td>4</td>
</tr>
<tr>
<td>Library loading</td>
<td>2</td>
</tr>
<tr>
<td>Reflection</td>
<td>3</td>
</tr>
<tr>
<td>Duck typing</td>
<td>1</td>
</tr>
<tr>
<td>Branching</td>
<td>6</td>
</tr>
<tr>
<td>Direct functions</td>
<td>9</td>
</tr>
<tr>
<td>Lambda functions</td>
<td>2</td>
</tr>
<tr>
<td>Nested code</td>
<td>2</td>
</tr>
<tr>
<td>Polymorphic functions</td>
<td>5</td>
</tr>
<tr>
<td>Recursion</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>49</strong></td>
</tr>
</tbody>
</table>

In the following, we discuss some of the categories that are unique to the Python programming language other than the ones presented in the motivation (Section 2).

Duck typing. This gives the ability to treat an object as if it is of the requested type without the knowledge of the existence of object properties. Duck typing gives more importance to the method an object defines than the type of the object.

Listing 3 shows an example of duck typing. It contains three classes Duck (at line 38), Mallard (at line 42), Person (at line 46), and a function shoot (at line 50). Duck typing is demonstrated in the function shoot (at line 51).
38 class Duck:
39     def quack(self):
40         print("Quack!")
41
42 class Mallard:
43     def quack(self):
44         print("Quack quack!")
45
46 class Person:
47     def quack(self):
48         print("Help!")
49
50 def shoot(bird):
51     bird.quack()
52
53 if __name__ == '__main__':
54     for b in [Duck(), Mallard(), Person()]:
55         shoot(b)

Listing 3. Example of a PYTHON class in the duck typing category.

It assumes that the parameter it receives will always contain a method quack associated to it and invokes it. One of the practical examples of this pattern is that any object can become iterable in PYTHON if __iter__ method is implemented.

Object changes. This is the ability of a variable to mute its type at runtime. A variable in PYTHON can be a number at a given execution point and can be an object at another point during execution. Even more, new attributes can be added to an object that the variable is pointing to at some other point during execution. Such implementations cause difficulties in identifying the type of the variables and contribute to false positive and negative edges in the static CG.

Static functions. In PYTHON, the declaration of static functions is peculiar. A static function can be defined using a decorator, @staticmethod (similar to the decorator declaration at line 10) or using a function, staticmethod. Listing 4 shows an example using a function call by passing the function name as the parameter (as shown in line 60). As shown in line 63, the static function is invoked without creating the object.

Other categories. Library loading includes programs which contain calls to library functions. PYTHON programs belonging to this category makes use of __import__ to fetch the libraries or classes dynamically.

56 class One:
57     def static_function():
58         return "hello"
59
60 static_function = staticmethod(static_function)
61
62 if __name__ == '__main__':
63     One.static_function()

Listing 4. Example of a PYTHON static function call using decorator.

Branching includes PYTHON classes and modules that demonstrate function calls inside the control-flow constructs such as for, while, and if-else. Direct functions contain programs that include arbitrary calls to functions in the same class or in the same module. Lambda functions contain PYTHON programs whose implementation contain the usage of lambda functions. A lambda function is defined without a name and accepts any number of function arguments. The body of the lambda function is a single statement which can also contain function calls. Nested code contains PYTHON programs whose implementation uses inner classes, inner methods, and contain the respective function calls. Polymorphic functions embed programs that use inheritance and function overloading. Recursion has programs featuring recursive function calls in PYTHON.

4.3 Quantitative analysis

In this experiment, called Real-World test, the input to eval_CG is a real-world application. Figure 4 shows different components of the experiment.

PYTHON projects are real-world applications from GitHub.² We used three criteria to select the projects. First, we considered the active maintenance, activities, and support from the developer community, which is depicted by the number of forks (at least 1,000) and number of stars (at least 10,000) in the respective repositories. The second selection criterion is that the projects should have a minimum of 85% branch coverage. The branch coverage for most of the projects is identified from coverage tools such as Coveralls³, and Codecov⁴, which are integrated in to the corresponding GitHub repositories. The last criterion is the coverage of the language constructs. For this, we manually inspected

²https://github.com/
³https://coveralls.io
⁴https://codecov.io
the corresponding implementations and marked the benchmark-suite categories covered by each real-world application. Table 3 shows the list of real-world applications with the corresponding branch coverage and the presence of benchmark suite categories.

![Real-World Test Diagram](image)

**Figure 4.** Process diagram of Real-World test.

Our Real-World test consists of five applications. Python robotics is a collection of robotics algorithms implemented in Python programming language focusing mainly on autonomous navigation [27]. mitmproxy is a free and open-source HTTP proxy tool implemented in Python [16]. Cookiecutter helps in creating Python projects from the project templates [6]. YouCompleteMe (ycm) is a code-completion engine for vim editor [5] [30]. The Fuck is a command-line utility tool that corrects the errors in the previous console commands [9].

Branch coverage forms an important criterion to dynamic CG generation. In this experiment, PyCallGraph is integrated into the test suite of each real-world application. When the test cases in the project are executed, PyCallGraph monitors the stack trace and generates the dynamic CG. It is configured to record edges only from the application source code. In the default configuration of the frameworks, the source code of the libraries is excluded from the analysis. Providing the source code of all the libraries will increase the CG generation time. Also, the library source code might not be publicly available. Hence, we decided to only record the edges from the application source code and to ignore the edges representing library function calls. In each of the real-world applications, we generate a dynamic CG for each test case, and in the end, combine all the dynamic CGs into one using the set union relationship, which represents the dynamic CG of the application.

The Entry-points Extractor is responsible for providing entry points to eval_CG. In Synthetic test, each file in the benchmark suite forms the entry point to each static and dynamic CG algorithms. However, in the case of a real-world application, the entry points depend on that application. Instead, we generate a CG for all the source files in each project individually and ignore the files for which any of the three CG frameworks throw an error. An error is thrown when the source code contains a Python feature that is not supported by the framework. Table 4 shows the reduction in the number of source files (entry points) for each real-world application as a result of entry-point extractor. Finally, we collect the common files for which a CG is generated successfully from the three frameworks. These extracted source files form the entry points to the static and dynamic CG frameworks. This filtration results in efficient comparison by using the same number of entry points in all the static and dynamic CG generation frameworks. As we see, though, the tool implementations fail on many input files, nCFA particularly so.

## 5 Experimental Results

We used eval_CG to evaluate the CG algorithms of Python programs in terms of soundness and precision. Specifically, we address the following four research questions:

**RQ 1** Which CG algorithm has the highest precision?
**RQ 2** Which CG algorithm has the highest recall?
**RQ 3** What are the sources of unsoundness in each CG algorithm?
**RQ 4** What is the CG algorithms’ running time?

WALA provides several algorithms. When evaluating them in terms of precision and recall, we found that nCFA performs generally better than other WALA CGs. One of the WALA authors also confirmed that nCFA was tailored for Python. Hence, in the following we only compare $C2F_{CG}$ (Code2Flow), $Pyan_{CG}$ (Pyan), and $nCFA$ (WALA).

### 5.1 RQ 1: Precision

The precision in both experiments is calculated as defined in Equation 1. Lower number of false positives, i.e., edges present only in the static CG, yields higher precision. Figure 5 shows the results from Synthetic test, Figure 6 the results from Real-World test. To
Table 3. Branch coverage and benchmark-suite category presence in the real-world applications.

<table>
<thead>
<tr>
<th>Project</th>
<th>Coverage (in %)</th>
<th>Runtime code generation</th>
<th>Decorators</th>
<th>Object changes</th>
<th>Static functions</th>
<th>Library loading</th>
<th>Reflection</th>
<th>Duck typing</th>
<th>Branching</th>
<th>Direct functions</th>
<th>Lambda functions</th>
<th>Nested code</th>
<th>Polymorphic functions</th>
<th>Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python robotics</td>
<td>90</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mitmproxy</td>
<td>88</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>cookiecutter</td>
<td>100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>YouCompleteMe</td>
<td>89</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>The Fuck</td>
<td>93</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4. Number of entry points for each real-world application.

<table>
<thead>
<tr>
<th>Application</th>
<th>Total source files</th>
<th>Successfully processed source files</th>
<th>C2F&lt;sub&gt;CG&lt;/sub&gt;</th>
<th>PyAN&lt;sub&gt;CG&lt;/sub&gt;</th>
<th>nCFA</th>
<th>Common</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python robotics</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>mitmproxy</td>
<td>207</td>
<td>136</td>
<td>136</td>
<td>105</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>cookiecutter</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>14</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>YouCompleteMe (ycm)</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>19</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>The Fuck</td>
<td>197</td>
<td>180</td>
<td>180</td>
<td>170</td>
<td>157</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>538</td>
<td>450</td>
<td>450</td>
<td>340</td>
<td>317</td>
<td></td>
</tr>
</tbody>
</table>

compute the average, we used the geometric mean, as recommended for normalized values.

In **Synthetic test**, C2F<sub>CG</sub> has better precision than PyAN<sub>CG</sub> and nCFA on average. For the categories such as duck typing, nested code, object changes, and reflection it achieves 100% precision. PyAN<sub>CG</sub> follows C2F<sub>CG</sub> with 100% precision in duck typing, library loading, and recursion categories of the benchmark suite. nCFA has by far the worst precision among the three CG algorithms: 27% on average. In **Real-World test**, C2F<sub>CG</sub> provides significantly better precision than the other two algorithms, yet precision in general is rather poor. nCFA in particular shows horrible precision: virtually no edges of its static CG actually appear at runtime.

While Code2flow and PyAn show a useful level of precision on **Synthetic test**, on **Real-World test** even the best algorithm Code2flow only shows a precision of 33%.

5.2 **RQ 2: Recall**

In both experiments, the recall is calculated as defined in Equation 2. The recall is higher the more edges of the dynamic CG the static CG covers. Figure 7 shows the results from **Synthetic test**. Figure 8 shows the results from **Real-World test**.

In **Synthetic test**, while C2F<sub>CG</sub> gives precision of 72% on average, it provides poor results in terms of recall (25% on average). A CG algorithm that has a good precision but poor recall depicts that its CG tends to under-approximate.

Similar to the precision, PyAN<sub>CG</sub> follows C2F<sub>CG</sub> in terms of recall with 40% on average. While nCFA is worst in terms of precision, it gives better recall (53% on average) than C2F<sub>CG</sub> and PyAN<sub>CG</sub>—but as we will see, only for **Synthetic test**. nCFA provides 100% recall for the duck typing, nested code, and recursion categories of the benchmark suite.
Qualitative and Quantitative Analysis of Callgraph Algorithms for Python

![Figure 5. Precision of CG algorithms from Synthetic test](image)

![Figure 6. Precision of CG algorithms from Real-World test](image)

In Real-World test, PyanCG shows better recall (33% on average) than nCFA and C2FCG. For the applications such as cookiecutter, and Python robotics, PyanCG contains more than half of the dynamic CG edges (recall > 50%). While C2FCG has the average precision of 33%, it has less recall when compared to PyanCG but more than nCFA (on average, 27%). Unfortunately, also in terms of recall, nCFA performs terrible for real-world programs (2% on average).

**Discussion.** Overall one must conclude that, while some algorithms perform okay on Synthetic test, when applied to real-world programs, the results of all three algorithms are still far from perfect, which really questions the practical utility of these current implementations. WALA’s nCFA in particular shows surprisingly bad results. In Section 5.3.2 we will investigate more deeply in particular the prime reasons for low recall.

5.3 RQ 3: Sources of Unsoundness

This section is divided into two research questions:

- **RQ 3.1** What are the sources of unsoundness in Python language constructs?
• RQ 3.2 How does the type of real-world application influence the edges in the static CG?

5.3.1 Influence of language constructs. We classify the edges of the dynamic CG of a Python program into three types, namely, initialization edges, direct edges, and cardinal edges. We evaluate the CG algorithms for these three types of edges. To understand the three edge types, we revisit the Listing 1 and its corresponding dynamic CG in Figure 1. The initialization edges refer to the edges in the CG that are generated between the classes or methods and their namespaces, such as the edge between `main` and the class `Calculator`. The direct edges are the edges in the CG generated for the function calls that are straightforward, such as the edge between `main` and `calculate_sum` method. The cardinal edges refer to the edges generated for the function calls that use special language constructs mentioned in the benchmark suite categories. In our example, the edge generated between `calculate_scientific` and `calculate_cos` methods is a cardinal edge generated due to runtime code generation. The direct and cardinal edges for the direct functions and branching categories in the benchmark suite are same.

The cardinal edges are the important edges that the CG algorithms should support. These edges are directly proportional to the level of support for the categories in the benchmark suite.

We manually analyzed the FNs of each static CG algorithm for all the programs in the benchmark suite to
categorize them and record the statistics. Table 5 shows how the FNs are distributed over the three edge types (initialization, direct, and cardinal edges) for each of the three static CG algorithms. It also shows, how missing recall percentages (Figure 7) for each benchmark suite category is distributed over the three edge types.

**Conclusions.** Amongst the three edge types, most of the FNs amount to initialization edges and cardinal edges.

All the three static CG algorithms miss very few direct edges as depicted by the total direct edges in the Table 5 (three for C2F, zero for PyAN, and two in the case of nCFA).

For the categories, that have 100% recall in the Figure 7 (nCFA for duck typing, nested code, and recursion), the table shows that there are no FNs, which validates the comparison.

C2F computes all cardinal edges only for recursion, whereas PyAN does so for recursion and nested code. In all the other categories for the three static CG algorithms at least some cardinal edges are lacking.

Though only nCFA supports runtime code generation (19% recall as shown in Figure 7), it only generates initialization edges. Yet, in the case of lambda functions nCFA shows only two missing cardinal edges, which is less than C2F (six), and PyAN (nine).

In the case of static functions, though C2F and PyAN have non-zero recall, they still miss to generate cardinal edges.

Among the three static CG algorithms, nCFA has the least number of total FN (94) and FN cardinal edges (49), and C2F has the highest number of total FNs (197) and FN cardinal edges (67). This is also depicted from the Figure 7, which shows nCFA has the highest average recall. The support to Python programming constructs is also less from the three static CG algorithms as depicted by the non-zero FN cardinal edges.

**5.3.2 Close-up on real-world applications.** To shed more light on the results on real-world applications, we inspected the number of CG edges generated by each algorithm for the five real-world applications used in the Real-World test. Figure 9 shows the statistics of CG edges using venn diagrams for each real-world application.

The left side of each venn diagram shows the number of edges in the dynamic CG (represented by green color). The right side shows the number of static CG edges (C2F is represented by orange, PyAN with blue, and nCFA with pink color). The intersection shows the number of edges that are present in both the static CG and the dynamic CG, i.e., number of TPs.

Within the intersection, for most of the applications, PyAN has more edges than C2F and nCFA. C2F follows PyAN in CG generation for real-world applications. nCFA shows almost no TPs edges at all. In fact it seems like its static CG reflect almost not at all how the applications execute. Most edges the CG does contain are synthetic edges. nCFA also filters more files than PyAN and C2F as observed from the Table 4. Hence, it is not suitable to generate a better CG for applications that include multiple source files.

For real-world programs, none of the three algorithms produces a callgraph that closely mirrors these programs’ runtime execution. For nCFA, the static callgraph has no resemblance to the dynamic callgraph at all.

**5.4 RQ 4: Runtime**

We next compare the algorithms’ running times. Figure 10 shows the runtime of each CG algorithm from Synthetic test and Real-World test. The left part of the figure shows the arithmetic mean of time taken by the static CG algorithms to generate a CG for the benchmark suite and the right part shows the runtime for real-world applications. PyAN generates CG in shorter time than C2F and nCFA. Though C2F produces a better CG in terms of precision and recall for real-world applications, it has far longer running times.

**Conclusions.** C2F’s long running times can be explained by the fact that it iterates over its CG nodes with the complexity of $O(n^2)$. Though the running time of C2F is less than nCFA in Synthetic test, it increases as the number of input files increase as shown in the case of real-world applications. nCFA takes much time in parsing the source file and walking through the source code. It also uses the Jython library to obtain an abstract syntax tree (AST) of the source file. Though nCFA takes less time than PyAN in the case of real-world applications, it has virtually no TPs.

While PyAN and nCFA generally compute callgraphs in under a second, for Code2Flow computation times tend to be much higher.

**5.5 Threats to Validity**

The results of all three algorithms are much worse than we had originally expected. Consequently, we took

[49 of 93]
Table 5. Categorisation of FNs into initialization (I), direct (D), and cardinal edges (C) from SYNTHETIC TEST.

<table>
<thead>
<tr>
<th>Category</th>
<th>C2F&lt;sub&gt;CG&lt;/sub&gt;</th>
<th>PyAN&lt;sub&gt;CG&lt;/sub&gt;</th>
<th>nCFA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#FNs</td>
<td>I</td>
<td>D</td>
</tr>
<tr>
<td>Branching</td>
<td>15</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>Runtime code generation</td>
<td>17</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Decorators</td>
<td>24</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Direct functions</td>
<td>28</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>Duck typing</td>
<td>7</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Lambda functions</td>
<td>12</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Library loading</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Nested code</td>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Object changes</td>
<td>20</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Polymorphic functions</td>
<td>24</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>Recursion</td>
<td>7</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Reflection</td>
<td>16</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Static functions</td>
<td>11</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>197</strong></td>
<td><strong>127</strong></td>
<td><strong>3</strong></td>
</tr>
</tbody>
</table>

Figure 9. CG edge statistics for real-world applications

great care to validate our experimental setup. This setup is also openly available as a curated artifact.

 Nonetheless, we identified three remaining threats to validity in our approach to evaluating the CG algorithms for Python.
6 Related Work

In the following, we discuss the related studies with empirical nature that evaluate static callgraph (CG) algorithms for different programming languages.

Yu evaluated three tools for Python CG construction [31], namely, PyAn [20], Code2flow [5], and Understand [28]. These were compared to a dynamic CG created by PyCallGraph [21]. The CGs are compared based on metrics for centrality, connectivity, and the PageRank algorithm. Additionally, the author manually identified impactful nodes from the experimental data to evaluate which tool creates the most accurate CG. In this study, the commercial tool, Understand showed the best results. This study was performed on selected Python applications and did not consider the language constructs. Moreover, no framework as eval_CG is designed, nor available.

Static analyses for the Java language are one of the most mature, and hence, there are few studies that evaluate the CG algorithms for Java. Rountev et al. published one of the first evaluations of static CGs based on dynamically collected traces of the method calls [25]. They evaluated the rapid type analysis (RTA) implementation in Soot and identified the reasons for imprecision. Reif et al. published Judge [23], a CG evaluation framework for Java which compared multiple algorithms implemented in Soot, Wala and Opal, identifying reasons for unsoundness. This study also showed that the implementation of an algorithm among the frameworks leads to different results. Grove et al. presented an evaluation framework [10] for the algorithms implemented in Wala in which they evaluated Java and Cecil programs. This study shows insights on the relationship between the runtime and the precision of the algorithms. Jász et al. published comparative study
on six Java CG construction tools [12] based on a self-created application with different language features and two real-world applications.

Lhotak proposed an algorithm for comparing CGs [14] that additionally identifies important call edges with high impact on the CG accuracy. He used a case study with static and dynamic CG on one benchmark Java application to demonstrate the tool which also includes a simple interactive component.

Murphy et al. explored nine tools that generated CGs for C code [19] and is one of the first empirical study in the area to stress out the importance of the choice of CG algorithm with respect to the application.

Antal et al. explored the CGs of JavaScript programs [3]. In this comparative study they used five tools. The results showed very high precision for most of the tools. However, the number of missing edges is relatively high, which is caused by the dynamic nature of the language that makes static CG construction hard.

Finally, Ali et al. have published an extensive study on multiple algorithms and programming languages [2]. In the case of Python, the authors converted the Python code to Java-bytecode and used the algorithms of Soot and WALA for the evaluation. They showed that generating a CG using this method often leads to high imprecision.

7 Conclusion

In this paper, we stressed out the importance of the call-graph (CG) as a fundamental data structure for inter-procedural static analyses and the need for empirical evidence on the quality of the state-of-the-art CG algorithms for Python. We presented a reusable CG Evaluation Framework and used it to fully automatically evaluate three existing CG algorithms for Python. We performed two experiments, evaluating the CG algorithms both in terms of precision and recall, using both a benchmark suite of synthetic, small Python programs, and real-world applications with high test coverage from GitHub.

The results surprised us because they clearly show that, while the algorithms perform okay on the synthetic micro-benchmarks, they provide both poor precision and recall for real-world programs. While WALA’s nCFA algorithm and PyANCG give the best recall when executed on single, synthetic Python files, nCFA is not suitable for CGs generation of real-world applications: both its precision and recall are close to zero. The CG from PyAN framework (PyANCG) and CODE2FLOW (C2FCG) give more sound results than nCFA when executed on real-world applications but CODE2FLOW has a very long running time.

In conclusion, on real-world programs none of the CG algorithms show a precision and recall that one would deem acceptable in practice. In the case of CODE2FLOW, also performance optimizations seem advisable. Consequently, there is much room in the development of CG algorithms for the Python programming language.

References

[7] Cookiecutter. [n.d.]. A command-line utility that creates projects from cookiecutters (project templates), e.g. Python package projects, VueJS projects. https://github.com/cookiecutter/cookiecutter


