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

2021 International Conference on Code Quality (ICCQ) | 978-1-7281-8476-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCQ51190.2021.9392986

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,[1] 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.

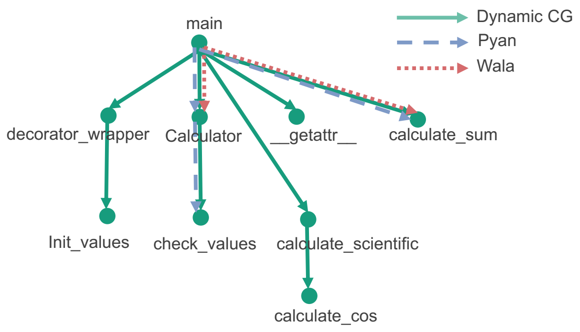


Figure 1. The dynamic CG and the static CGs constructed by WALA, and PYAN for Listing 1

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.

```
1   def check_values(function):
2       def decorator_wrapper(*args):
3           return_value = "Invalid
                parameters"
4           if isinstance(args[1], (int,
                float, complex)) and
                isinstance(args[2], (int,
                float, complex)):
5               return function(*args)
6           print(return_value)
7       return decorator_wrapper
8
9   class Calculator(object):
10      @check_values
11      def init_values(self, x, y=0):
12          self.x = x
13          self.y = y
14
15      def calculate_sum(self):
16          print(self.x + self.y)
17
18      def calculate_scientific(self):
19          x = self.x
20          embedded = '''
21              import math
22              def calculate_%s():
23                  return math.%s(x)
24              print(calculate_%s())
25              ''' % (self.operation,
                    self.operation,
                    self.operation)
26          exec(embedded, locals())
27
28      def __getattr__(self,
           function_name):
29          operation =
                function_name.split("_")[1]
30          if operation in ["sin", "cos",
                "tan"]:
31              self.operation = operation
32              return
                    self.calculate_scientific
```

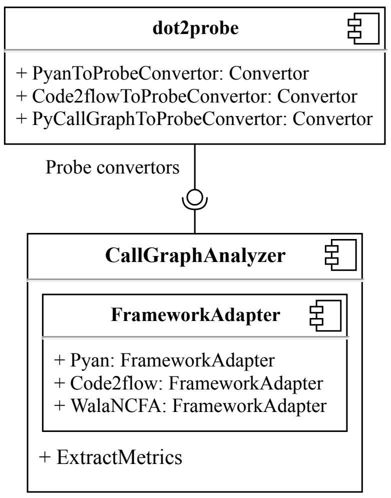
Listing 1. PYTHON program showcasing dynamic language features.

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 quantitive analysis

```
    if __name__ == '__main__':
        obj = Calculator()
        obj.init_values(5, 10)
        obj.calculate_sum() #decorated
            function call
        obj.calculate_cos() #undefined
            function call using
            reflection
```

Listing 2. Main function for the example in Listing 1



dot2probe
+ PyanToProbeConvertor: Convertor
+ Code2flowToProbeConvertor: Convertor
+ PyCallGraphToProbeConvertor: Convertor
Probe convertors
CallGraphAnalyzer
FrameworkAdapter
+ Pyan: FrameworkAdapter
+ Code2flow: FrameworkAdapter
+ WalaNCFA: FrameworkAdapter
+ ExtractMetrics

Figure 2. Component diagram of EVAL_CG

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 EVAL_CG. The framework consists of three main components, *Dot2probe*, *Framework Adapter*, and *Call Graph Analyzer*.

Dot2probe. Each of the CG generation frameworks has its own CG format, which makes the comparison a tedious task. EVAL_CG converts all the static and dynamic CGs to a single format. We chose to reuse Lhoták'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 PYAN, CODE2FLOW, and PYCALLGRAPH are in the dot format [13], we implemented the corresponding probe-converters in this component.

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

Call Graph Analyzer This is the central component of 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, *ExtractMetrics*, to compare the CGs. *ExtractMetrics* checks each edge in the static probe CG against each edge in the dynamic probe CG to record the metrics. Though 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 quantitive comparison. So, we only adapt the CG format from Probe and implement the comparison suiting our needs.

The dynamic CG is generated using PYCALLGRAPH [21]. This component runs PYCALLGRAPH on the input PYTHON program or alongside the test cases in the case of input PYTHON application to generate the dynamic CG. PYCALLGRAPH uses *sys.set_trace()* function, which is called by 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 *TP* is a CG edge that is present in the static and the dynamic CG. A *FP* is an edge that is only present in the static CG. A *FN* is an edge that is only present in the dynamic CG.

Using the above terms, the *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 *recall* is defined as the ratio of the total number of dynamic CG edges present in the static CG (as shown in Equation 2).

$$precision_{CG} = \frac{TP}{TP + FP} \quad (1)$$

$$recall_{CG} = \frac{TP}{TP + FN} \quad (2)$$

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.

Framework	CG algorithm
CODE2FLOW	$C2F_{CG}$
PYAN	$PYAN_{CG}$
WALA	*nCFA*, VanillaZeroOneCFA, ZeroCFA, ZeroContainerCFA, ZeroOneCFA, ZeroOneContainerCFA

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

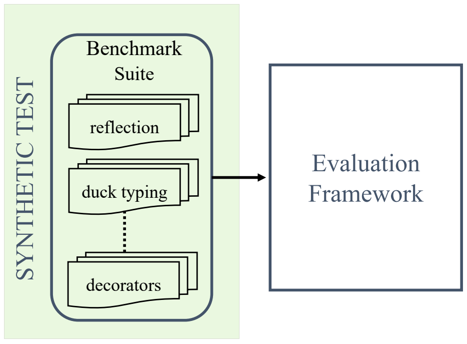


Figure 3. Process diagram of SYNTHETIC TEST.

in the benchmark suite along with the number of programs in each category.

Table 2. Benchmark categories with the number of programs

Category	No. of programs
Runtime code generation	3
Decorators	5
Object changes	4
Static functions	4
Library loading	2
Reflection	3
Duck typing	1
Branching	6
Direct functions	9
Lambda functions	2
Nested code	2
Polymorphic functions	5
Recursion	3
Total	**49**

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).

```python
class Duck:
    def quack(self):
        print("Quack!")

class Mallard:
    def quack(self):
        print("Quack quack!")

class Person:
    def quack(self):
        print("Help!")

def shoot(bird):
    bird.quack()

if __name__ == '__main__':
    for b in [Duck(), Mallard(), Person()]:
        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.

```python
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 *Coveralls*[3], and *Codecov*[4], which are integrated in to the corresponding GitHub repositories. The last criterion is the coverage of the language constructs. For this, we manually inspected

[2]https://github.com/
[3]https://coveralls.io
[4]https://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.

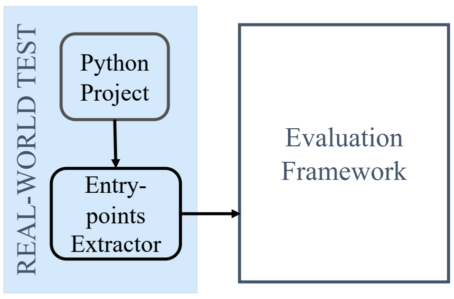


Figure 4. Process diagram of REAL-WORLD TEST.

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

[5]vim editor is the text editor most used in Linux and Unix operating systems.

Table 3. Branch coverage and benchmark-suite category presence in the real-world applications.

Project	**Coverage** (in %)	Runtime code generation	Decorators	Object changes	Static functions	Library loading	Reflection	Duck typing	Branching	Direct functions	Lambda functions	Nested code	Polymorphic functions	Recursion
Python robotics	90				✓			✓	✓	✓	✓	✓	✓	
mitmproxy	88	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
cookiecutter	100		✓						✓	✓		✓		
YouCompleteMe	89	✓	✓		✓		✓		✓	✓	✓	✓	✓	
The Fuck	93		✓				✓	✓	✓	✓	✓	✓	✓	

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

Application	**Total source files**	**Successfully processed source files**			
		C2F$_{CG}$	**Pyan$_{CG}$**	***nCFA***	**Common**
Python robotics	92	92	92	32	32
mitmproxy	207	136	136	105	95
cookiecutter	18	18	18	14	14
YouCompleteMe (ycm)	24	24	24	19	19
The Fuck	197	180	180	170	157
Sum	538	450	450	340	317

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

In Synthetic test, C2F$_{CG}$ has better precision than Pyan$_{CG}$ and *nCFA* on average. For the categories such as duck typing, nested code, object changes, and reflection it achieves 100% precision. Pyan$_{CG}$ follows C2F$_{CG}$ 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$_{CG}$ 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$_{CG}$ 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$_{CG}$ follows C2F$_{CG}$ 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$_{CG}$ and Pyan$_{CG}$—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.

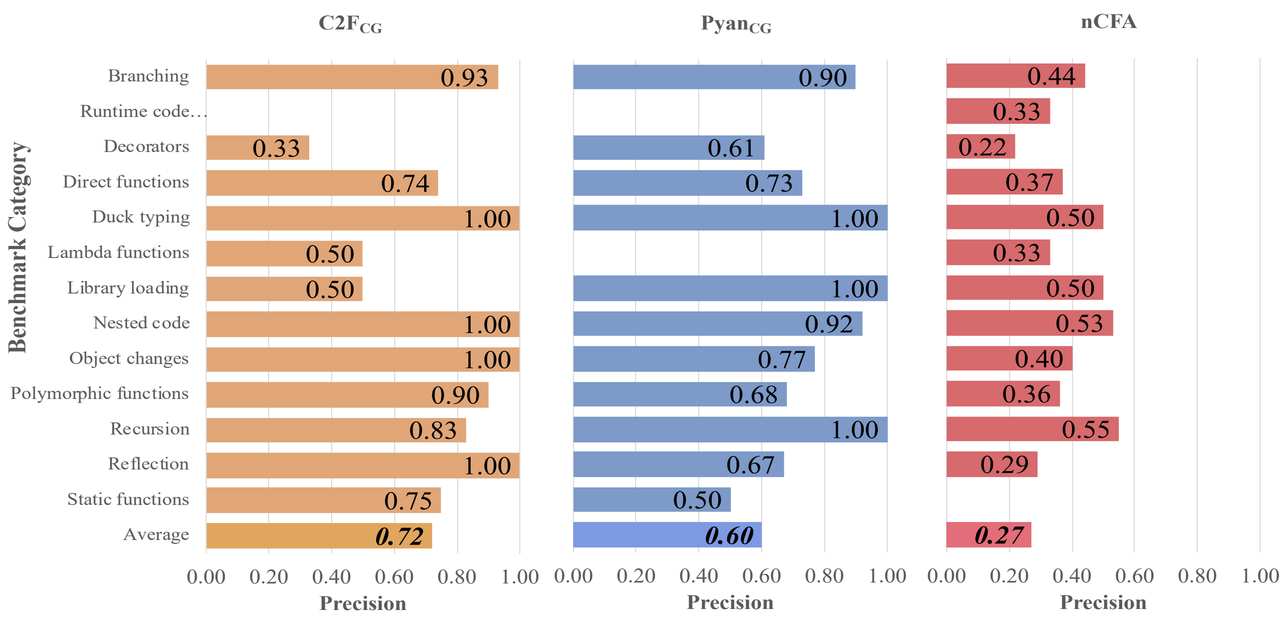


Figure 5. Precision of CG algorithms from SYNTHETIC TEST

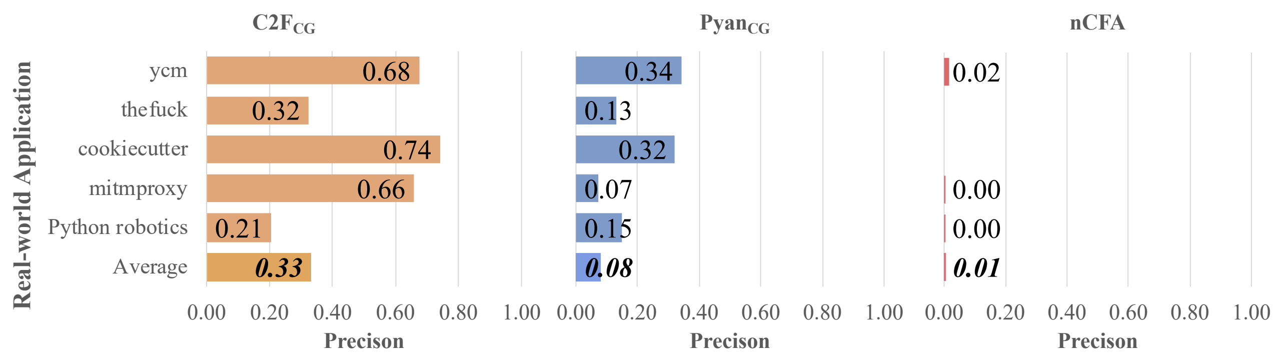


Figure 6. Precision of CG algorithms from REAL-WORLD TEST

In REAL-WORLD TEST, PYAN$_{CG}$ shows better recall (33% on average) than *nCFA* and C2F$_{CG}$. For the applications such as *cookiecutter*, and *Python robotics*, PYAN$_{CG}$ contains more than half of the dynamic CG edges (recall > 50%). While C2F$_{CG}$ has the average precision of 33%, it has less recall when compared to PYAN$_{CG}$ 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.

While *nCFA* and PYAN show at least somewhat useful level of recall on SYNTHETIC TEST, on REAL-WORLD TEST PYAN and CODE2FLOW perform best, however, also only with 33% and 27% recall, respectively.

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?

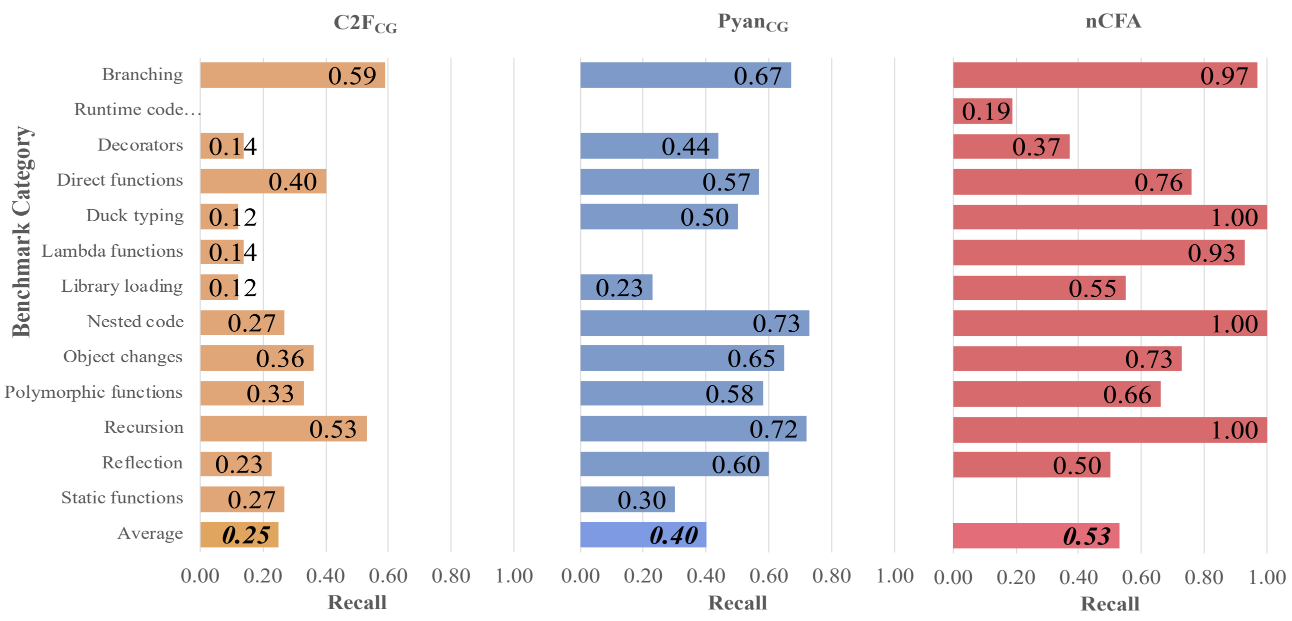


Figure 7. Recall of CG algorithms from Synthetic test

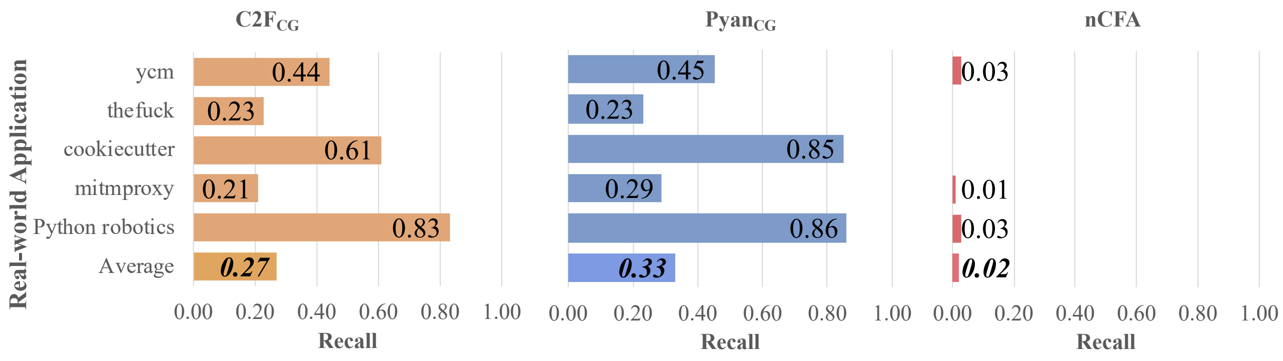


Figure 8. Recall of CG algorithms from Real-World test

- **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_{CG}, zero for PYAN_{CG}, 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_{CG} computes all cardinal edges only for recursion, whereas PYAN_{CG} 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_{CG} (six), and PYAN_{CG} (nine).

In the case of static functions, though C2F_{CG} and PYAN_{CG} have non-zero recall, they still miss to generate cardinal edges.

Among the three static CG algorithms, *nCFA* has the least number of total FNs (94) and FN cardinal edges (49), and C2F_{CG} 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_{CG} is represented by orange, PYAN_{CG} 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_{CG} has more edges than C2F_{CG}, and *nCFA*. C2F_{CG} follows PYAN_{CG} 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_{CG} and C2F_{CG} 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_{CG} generates CG in shorter time than C2F_{CG} and *nCFA*. Though C2F_{CG} produces a better CG in terms of precision and recall for real-world applications, it has far longer running times.

Conclusions. C2F_{CG}'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_{CG} 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[6] 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

[6]https://www.jython.org/

Category	C2F$_{CG}$				PYAN$_{CG}$				*nCFA*			
	#FNs	*I*	*D*	*C*	#FNs	*I*	*D*	*C*	#FNs	*I*	*D*	*C*
Branching	15	14	0	1	12	11	0	1	2	1	0	1
Runtime code generation	17	8	0	9	17	8	0	9	14	5	0	9
Decorators	24	11	0	13	16	9	0	7	17	8	0	9
Direct functions	28	18	0	10	15	11	0	4	10	4	0	6
Duck typing	7	4	0	3	4	1	0	3	0	0	0	0
Lambda functions	12	6	0	6	16	7	0	9	2	0	0	2
Library loading	8	3	1	4	7	3	0	4	4	0	0	4
Nested code	8	4	1	3	3	3	0	0	0	0	0	0
Object changes	20	16	0	4	11	8	0	3	9	5	0	4
Polymorphic functions	24	18	0	6	15	9	0	6	12	5	1	6
Recursion	7	6	1	0	3	3	0	0	0	0	0	0
Reflection	16	10	0	6	8	4	0	4	9	5	0	4
Static functions	11	9	0	2	10	8	0	2	15	10	1	4
Total	**197**	**127**	**3**	**67**	**137**	**85**	**0**	**52**	**94**	**43**	**2**	**49**

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

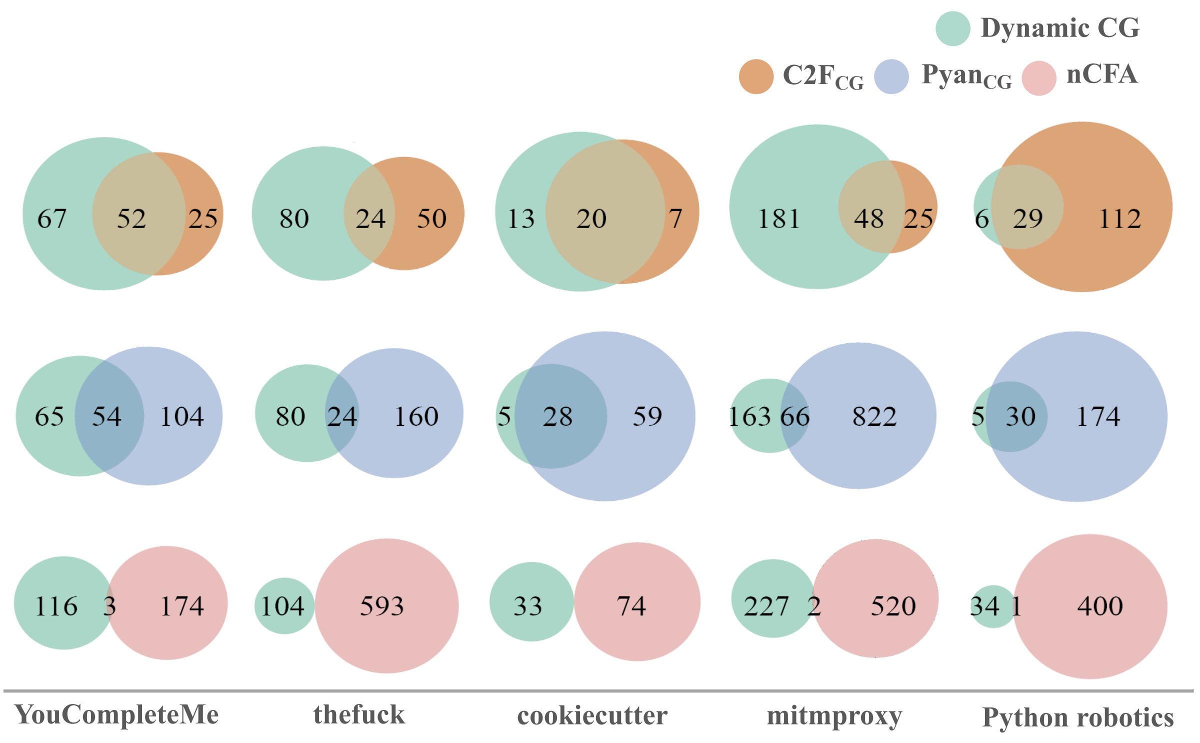


Figure 9. CG edge statistics for real-world applications

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.

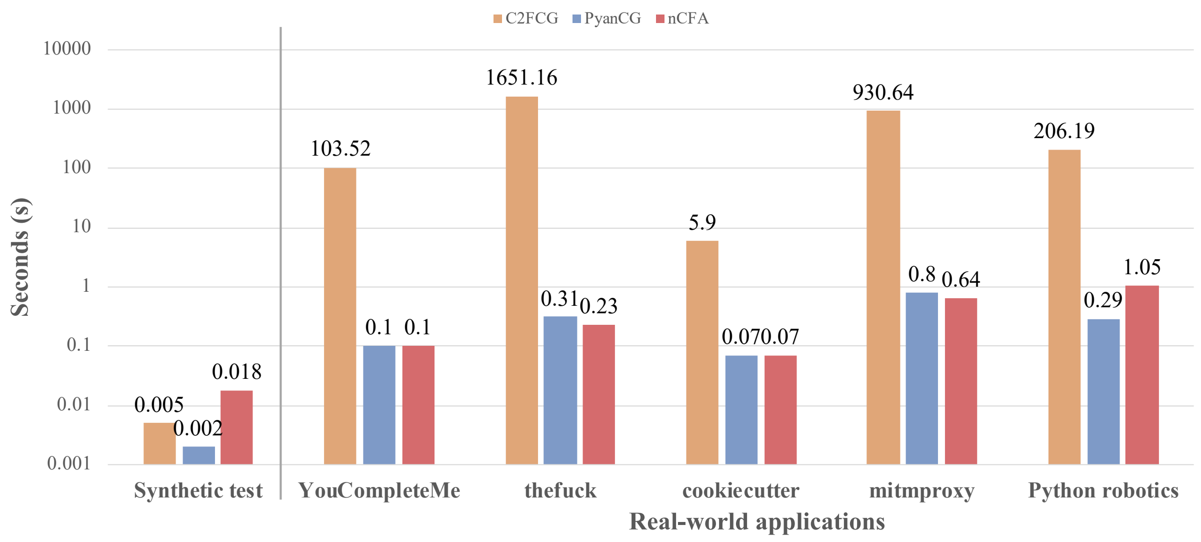


Figure 10. Runtime of CG algorithms for Synthetic test and Real-World test.

First, a possible threat is the representativeness of our benchmark suite. According to Nguyen et al. [8], four aspects should be considered while designing a benchmark: size, content, representativeness, and permanence. There is a threat to the representativeness of the benchmark suite as there can be some missing language constructs. To ensure an appropriate level of representativeness we followed the official Python specification with the focus on constructs that are relevant to function calls.

Second, in the Real-World test, we process only the source files from each real-world application that can be successfully processed by all the three CG frameworks. As a result, the total number of entry points to eval_CG reduces, which affects the metrics. If these files were to be included, the overall results would be yet much worse.

Third, the FPs are calculated as the number of static CG edges that are not present in the dynamic CG. In Real-World test, there is a possibility of labelling a static CG edge as a FP because the dynamic CG extracted from the test cases may not include this edge due to limited test coverage. Yet, exactly for this reason we chose only python projects with a high test coverage such that this threat is mitigated as much as possible.

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 PYAN_{CG} 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 (PYAN_{CG}) and CODE2FLOW (C2F_{CG}) 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

[1] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. 2014. Tracing dynamic features in python programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 292–295.

[2] Karim Ali, Xiaoni Lai, Zhaoyi Luo, Ondˇrej Lhotak, Julian Dolby, and Frank Tip. [n.d.]. A Study of Call Graph Construction for JVM-Hosted Languages. *IEEE Transactions on Software Engineering.* To Appear. (Accepted in 2019).

[3] Gabor Antal, Peter Hegedus, Zoltan Toth, Rudolf Ferenc, and Tibor Gyimothy. 2018. Static javascript call graphs: A comparative study. In *Proceedings - 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018 (Proceedings - 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018).* Institute of Electrical and Electronics Engineers Inc., 177–186. https://doi.org/10.1109/SCAM.2018.00028 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018 ; Conference date: 23-09-2018 Through 24-09-2018.

[4] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. *SIGPLAN Not.* 31, 10 (Oct. 1996), 324–341. https://doi.org/10.1145/236338.236371

[5] Code2flow. [n.d.]. *Turn your Python and Javascript code into DOT flowcharts.* https://github.com/scottrogowski/code2flow

[6] 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

[7] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, Mario Tokoro and Remo Pareschi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–101.

[8] Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. 2016. Toward an Automated Benchmark Management System. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Santa Barbara, CA, USA) *(SOAP 2016).* Association for Computing Machinery, New York, NY, USA, 13–17. https://doi.org/10.1145/2931021.2931023

[9] The Fuck. [n.d.]. *Magnificent app which corrects your previous console command.* https://github.com/nvbn/thefuck

[10] David Grove and Craig Chambers. 2001. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.* 23, 6 (Nov. 2001), 685–746. https://doi.org/10.1145/506315.506316

[11] Alex Holkner and James Harland. 2009. Evaluating the dynamic behaviour of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91.* Australian Computer Society, Inc., 19–28.

[12] Judit Jász, István Siket, Edit Pengo, Zoltán Ságodi, and Rudolf Ferenc. 2019. Systematic comparison of six open-source Java call graph construction tools. In *ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies),* Marten van Sinderen, Leszek Maciaszek, and Leszek Maciaszek (Eds.). SciTePress, 117–128. https://doi.org/10.5220/0007929201170128 14th International Conference on Software Technologies, ICSOFT 2019 ; Conference date: 26-07-2019 Through 28-07-2019.

[13] Eleftherios Koutsofios and Stephen C. North. 1996. Drawing Graphs With Dot. *CiteSeerX* (1996). https://doi.org/10.1.1.37.1582

[14] Ondřej Lhoták. 2007. Comparing Call Graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (San Diego, California, USA) *(PASTE '07).* Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/1251535.1251542

[15] Ondřej Lhoták et al. 2007. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.* ACM, 37–42.

[16] mitmproxy. [n.d.]. *An interactive TLS-capable intercepting HTTP proxy for penetration testers and software developers.* https://mitmproxy.org/

[17] Emily Morehouse. [n.d.]. *AST and me.* https://emilyemorehouse.github.io/ast-and-me/

[18] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. 1998. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.* 7, 2 (April 1998), 158–191. https://doi.org/10.1145/279310.279314

[19] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. 1998. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.* 7, 2 (April 1998), 158–191. https://doi.org/10.1145/279310.279314

[20] Pyan. [n.d.]. *Generate approximate call graphs for Python programs.* https://github.com/DavidFraser/pyan

[21] PYCALLGRAPH. [n.d.]. *Python Call Graph.* https://pycallgraph.readthedocs.io/en/master/

[22] Python. [n.d.]. *Python documentation.* https://docs.python.org/3/

[23] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, 251–261.

[24] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366.

[25] Atanas Rountev, Scott Kagan, and Michael Gibas. 2004. Static and Dynamic Analysis of Call Chains in Java. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Boston, Massachusetts, USA) *(ISSTA '04).* Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/1007512.1007514

[26] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1 (1996), 131 – 170. https://doi.org/10.1016/0304-3975(96)00072-2

[27] Atsushi Sakai, Daniel Ingram, Joseph Dinius, Karan Chawla, Antonin Raffin, and Alexis Paques. 2018. PythonRobotics: a Python code collection of robotics algorithms. *ArXiv* abs/1808.10703 (2018).

[28] Understand. [n.d.]. *Understand - Visualise your code.* https://scitools.com

[29] WALA. [n.d.]. *Watson Libraries for Analysis.* https://github.com/wala/WALA

[30] YouCompleteMe. [n.d.]. *A code-completion engine for Vim.* https://github.com/ycm-core/YouCompleteMe

[31] L. Yu. 2019. Empirical Study of Python Call Graph. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1274–1276. https://doi.org/10.1109/ASE.2019.00160

[32] Yan Zhang and Barbara M Wildemuth. 2009. Qualitative analysis of content. *Applications of social research methods to questions in information and library science* 308 (2009), 319.