PyCG: Practical Call Graph Generation in Python

Vitalis Salis,[§†] Thodoris Sotiropoulos,[§] Panos Louridas,[§] Diomidis Spinellis[§] and Dimitris Mitropoulos[§‡]
[§]*Athens University of Economics and Business*
[†]*National Technical University of Athens*
[‡]*National Infrastructures for Research and Technology - GRNET*
vitsalis@gmail.com, {theosotr, louridas, dds, dimitro}@aueb.gr

***Abstract*—Call graphs play an important role in different contexts, such as profiling and vulnerability propagation analysis. Generating call graphs in an efficient manner can be a challenging task when it comes to high-level languages that are modular and incorporate dynamic features and higher-order functions.**

Despite the language's popularity, there have been very few tools aiming to generate call graphs for Python programs. Worse, these tools suffer from several effectiveness issues that limit their practicality in realistic programs. We propose a pragmatic, static approach for call graph generation in Python. We compute all assignment relations between program identifiers of functions, variables, classes, and modules through an inter-procedural analysis. Based on these assignment relations, we produce the resulting call graph by resolving all calls to potentially invoked functions. Notably, the underlying analysis is designed to be efficient and scalable, handling several Python features, such as modules, generators, function closures, and multiple inheritance.

We have evaluated our prototype implementation, which we call PyCG, using two benchmarks: a micro-benchmark suite containing small Python programs and a set of macro-benchmarks with several popular real-world Python packages. Our results indicate that PyCG can efficiently handle thousands of lines of code in less than a second (0.34 seconds for 1k LoC on average). Further, it outperforms the state-of-the-art for Python in both precision and recall: PyCG achieves high rates of precision ∼99%, and adequate recall ∼69.3%. Finally, we demonstrate how PyCG can aid dependency impact analysis by showcasing a potential enhancement to GitHub's "security advisory" notification service using a real-world example.

***Index Terms*—Call Graph, Program Analysis, Inter-procedural Analysis, Vulnerability Propagation**

I. Introduction

A call graph depicts calling relationships between subroutines in a computer program. Call graphs can be employed to perform a variety of tasks, such as profiling [1], vulnerability propagation [2], and tool-supported refactoring [3].

Generating call graphs in an efficient way can be a complex endeavor especially when it comes to high-level, dynamic programming languages. Indeed, to create precise call graphs for programs written in languages such as Python and JavaScript, one must deal with several challenges including higher-order functions, dynamic and metaprogramming features (e.g., `eval`), and modules. Addressing such challenges can play a significant role in the improvement of dependency impact analysis [4], [5], [6], especially in the context of package managers such as *npm* [7] and *pip* [8].

To support call graph generation in dynamic languages, researchers have proposed different methods relying on static analysis. The primary aim for many implementations is completeness, i.e., facts deduced by the system are indeed true [9], [10], [11]. However, for dynamic languages, completeness comes with a performance cost. Hence, such approaches are rarely employed in practice due to scalability issues [12]. This has led to the emergence of *practical* approaches focusing on incomplete static analysis for achieving better performance [13], [14]. Sacrificing completeness is the key enabler for adopting these approaches in applications that interact with complex libraries [13], or Integrated Development Environments (IDEs) [14]. Prior work primarily targets JavaScript programs and—among other things—attempts to address challenges related to events and the language's asynchronous nature [15], [16].

Despite Python's popularity [17], there have been surprisingly few tools aiming to generate call graphs for programs written in the language. *Pyan* [18] parses the program's Abstract Syntax Tree (AST) to extract its call graph. Nevertheless, it has drawbacks in the way it handles the inter-procedural flow of values and module imports. *code2graph* [19], [20] visualizes Pyan-constructed call graphs, so it has the same limitations. *Depends* [21] infers syntactical relations among source code entities to generate call graphs. However, functions assigned to variables or passed to other functions are not handled by *Depends*, thus it does not perform well in the context of a language supporting higher-order programming. We will expand on the shortcomings of the existing tools in the remainder of this work. That said, developing an effective and efficient call graph generator for a dynamically typed language like Python is no minor task.

We introduce a practical approach for generating call graphs for Python programs and implement a corresponding prototype that we call PyCG. Our approach works in two steps. In the first step we compute the *assignment graph*, a structure that shows the assignment relations among program identifiers. To do so, we design a context-insensitive inter-procedural analysis operating on a simple intermediate representation targeted for Python. Contrary to the existing static analyzers, our analysis is capable of handling intricate Python features, such as higher-order functions, modules, function closures, and multiple inheritance. In the next step, we build the call graph of the original program using the assignment graph. Specifically, we utilize the graph to resolve all functions that can be potentially pointed to by callee variables. Such a programming pattern is particularly common in higher-order programming.

Similar to previous work [14], our analysis does not deal with non-functional values, and follows a conservative approach, meaning that the analysis does not reason about loops and conditionals. To make our analysis more precise, especially when dealing with features like inheritance, modules or programming patterns such as duck typing [22], we distinguish attribute accesses (i.e, $e.x$) based on the namespace where the attribute (x) is defined. Prior work uses a *field-based* approach that correlates attributes of the same name with a single global location without taking into account their namespace [14]. This leads to false positives. Our design choices make our approach achieve high rates of precision, while remaining efficient and applicable to large-scale Python programs.

We evaluate the effectiveness of our method through a micro- and a macro-benchmarking suite. Also, we compare it against Pyan and *Depends*. Our results indicate that our method achieves high levels of precision (∼99%) and adequate recall (∼69.3%) on average, while the other analyzers demonstrate lower rates in both measures. Our method is able to handle medium-sized projects in less than one second (0.34 seconds for 1k LoC on average). Finally, we show how our method can accommodate the fine-grained tracking of vulnerable dependencies through a real-world case study.

Contributions. Our work makes the following contributions.

- We propose a static approach for pragmatic call graph generation in Python. Our method performs inter-procedural analysis on an intermediate language that records the assignment relations between program identifiers, i.e., functions, variables, classes and modules. Then it examines the documented associations to extract the call graph (Section III).
- We develop a micro-benchmark suite that can be used as a standard to evaluate call graph generation methods in Python. Our suite is modular, easily extendable, and covers a large fraction of Python's functionality related to classes, generators, dictionaries, and more (Section V-A1).
- We evaluate the effectiveness of our approach through our micro-benchmark and a set of macro-benchmarks including several medium-sized Python projects. In all cases our method achieves high rates of precision and recall, outperforming the other available analyzers (Sections V-B, V-C).
- We demonstrate how our approach can aid dependency impact analysis through a potential enhancement of GitHub's "security advisory" notification service (Section V-E).

II. Background

Generating precise call graphs for Python programs involves several challenges. Existing static approaches fail to address these challenges leaving opportunities for improvement.

A. Challenges

- *Higher-order Functions*: In a high-level language such as Python, functions can be assigned to variables, passed as parameters to other functions, or serve as return values.
- *Nested Definitions*: Function definitions can be nested, meaning that a function can be defined and invoked within the context of another function.

```python
import cryptops

class Crypto:
    def __init__(self, key):
        self.key = key

    def apply(self, msg, func):
        return func(self.key, msg)

crp = Crypto("secretkey")
encrypted = crp.apply("hello world",
  ↪   cryptops.encrypt)
decrypted = crp.apply(encrypted,
  ↪   cryptops.decrypt)
```

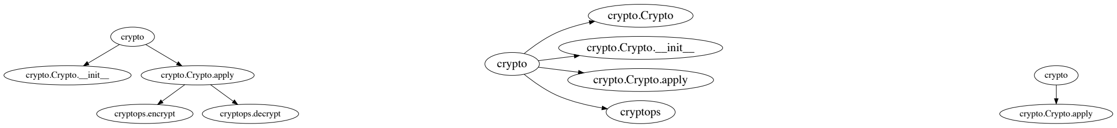
Fig. 1: The `crypto` module. Existing tools fail to generate a corresponding call graph effectively.

- *Classes*: As an object-oriented language, Python allows for the creation of classes that inherit attributes and methods from other classes. The resolution of inherited methods from parent classes requires the computation of the Method Resolution Order (MRO) of each class.
- *Modules*: Python is highly extensible, allowing applications to import different modules. Keeping track of the different modules that are imported in an application, as well as the resolution order of those imports, can be a challenging task.
- *Dynamic Features*: Python is dynamically typed, allowing variables to take values of different types during execution. Also, it allows for classes to be dynamically modified during runtime. Furthermore, the `eval` function allows for a dynamically constructed string to be executed as code.
- *Duck Typing*: Duck typing is a programming pattern that is particularly common in dynamic languages such as Python [22]. Through duck typing, the suitability of an object is determined by the presence of specific methods and properties, rather than the type of the object itself. In this context, given a method defined by two (or more) classes, it is not trivial to identify its origins when it is invoked.

B. Limitations of Existing Static Approaches

We focus on two *open-source* static analyzers: Pyan [18] and *Depends* [21]. We do not examine code2graph [19], [20] separately, as it is based on Pyan to generate call graphs. We discuss the limitations of the two existing analyzers in terms of efficiency and practicality. To do so, we introduce a small Python module named `crypto` (see Figure 1), which is used to encrypt and decrypt a "hello world" message. First, it imports an external Python module named `cryptops`, which defines two functions, namely: `encrypt(key, msg)` and `decrypt(key, msg)`. Then, the `Crypto` class is defined. To use it, we instantiate it with an encryption key and we can encrypt or decrypt messages by calling `apply(self, msg, func)`, where `func` is one of `encrypt(key, msg)` and `decrypt(key, msg)`. Figure 2a shows the call graph of the module.

Pyan [18] produces the imprecise call graph shown in Figure 2b. This graph does not contain all function calls, because the tool does not track the inter-procedural flow of values. Therefore, it is unable to infer which functions are



(a) Precise call graph.

(b) Pyan-generated call graph.

(c) *Depends*-generated call graph.

Fig. 2: Call graphs for the `crypto` module.

passed as arguments to `apply(self, msg, func)`. In addition, there are several features that lead to the addition of unrealized call edges. Specifically, when Pyan detects object initialization, it creates call edges to both the class name and the `__init__()` method of the class.[1] Beyond that, in the case of a module import, Pyan generates a call edge from the importing namespace to the module name.

Depends produces the call graph presented in Figure 2c. As *Depends* does not track function calls originating from the module's namespace (e.g., `crypto.apply()`). This in turn, led to an empty call graph. Therefore, to get a result, we wrapped those function calls within a new function. The resulting graph does not contain most of the calls included in the source program. This is because depends does not capture the call to the `__init__()` function of the `Crypto` class. Furthermore, (like Pyan) *Depends* does not track the inter-procedural flow of functions leading to missing edges to the parameter functions. Compared to Pyan, depends follows a more conservative approach. That is, it only includes a call edge when it has all the necessary information it needs to anticipate that the call will be realized. Contrary to Pyan, this can lead to a call graph without false positives.

III. Practical Call Graph Generation

Our approach for generating call graphs employs a context-insensitive inter-procedural analysis operating on an intermediate representation of the input Python program. The analysis uses a fixed-point iteration algorithm, and gradually builds the *assignment graph*, which is a structure that shows the assignment relations between program identifiers (Section III-A). In a language supporting higher-order programming, the assignment graph is an essential component that we use for resolving functions pointed to by variables. Function resolution takes place at the final step where we build the call graph for the given program by exploiting the assignment graph stemming from the analysis step (Section III-B).

A. The Core Analysis

The starting point of our approach is to compute the assignment graph using an inter-procedural analysis working on an intermediate representation targeted for Python programs.

One of the key elements of our analysis is that it examines attribute accesses based on the namespace where each attribute is defined. For example, consider the following code snippet:

$$\begin{aligned}
e \in \mathit{Expr} &::= o \mid x \mid x := e \mid \textbf{function}\ x\ (y\ldots)\ e \mid \textbf{return}\ e \mid \\
&\ e(x{=}e\ldots) \mid \textbf{class}\ x\ (y\ldots)\ e \mid e.x \mid e.x := e \mid \\
&\ \textbf{new}\ x\ (y = e\ldots) \mid \textbf{import}\ x\ \textbf{from}\ m\ \textbf{as}\ y \mid \\
&\ \textbf{iter}\ x \mid e;e \\
o \in \mathit{Obj} &::= n, v \\
v \in \mathit{Definition} &::= x, \tau \\
\tau \in \mathit{IdentType} &::= \textbf{func} \mid \textbf{var} \mid \textbf{cls} \mid \textbf{mod} \\
n \in \mathit{Namespace} &::= (v)^* \\
x, y \in \mathit{Identifier} &::= \textit{is the set of program identifiers} \\
m \in \mathit{Modules} &::= \textit{is the set of modules} \\
E &::= [] \mid x := E \mid \textbf{return}\ E \mid E(x = e\ldots) \mid \\
&\ o(x = E\ldots) \mid \textbf{new}\ x(y{=}E) \mid E.x \mid E.x := e \mid \\
&\ o.x := E \mid \textbf{iter}\ o \mid E;e \mid o;E
\end{aligned}$$

Fig. 3: The syntax for representing the input Python programs along with the evaluation contexts.

```
1  class A:
2    def func():
3      pass
4
5  class B:
6    def func():
7      pass
8
9  a = A()
10 b = B()
11 a.func()
12 b.func()
```

Our analysis is able to distinguish the two functions defined at lines 2 and 6, because they are members of two different classes, i.e., class `A` and `B` respectively. Note that field-based approaches focused on JavaScript [14] will fail to treat the two invocations as different, causing imprecision. That is because a field-based approach will match all accesses of identical attribute names (e.g., `func()`) with a single object.

1) Syntax: The intermediate representation, where our analysis works on, follows the syntax of a simple imperative and object-oriented language, which is shown in Figure 3. The last rule in this figure also shows the evaluation contexts [23] for this language, which we will explain shortly.

An important element of this model language is identifiers. Every identifier can be one of the following four types: (1) **func** corresponding to the name of a function (2) **var** indicating the name of a variable, (3) **cls** for class names, and (4) **mod** when the identifier is a module name. Every pair $(x, \tau) \in \mathit{Identifier} \times \mathit{IdentType}$ forms a definition. We represent every definition and its namespace as an object (see the Obj rule). A namespace is a sequence of definitions, and it is essential for distinguishing objects sharing the same identifier from each other. For example, consider the following

[1] In Python, `__init__()` is the name of a special function called during object construction.

$$\begin{aligned}
&\pi \in \mathit{AssignG} = (\mathit{Namespace} \times \mathit{Identifier}) \mapsto \\
&\quad \mathcal{P}(\mathit{Namespace} \times \mathit{Identifier}) \\
&s \in \mathit{Scope} = \mathit{Definition} \hookrightarrow \mathcal{P}(\mathit{Definition}) \\
&h \in \mathit{ClassHier} = \mathit{Obj} \hookrightarrow \mathit{Obj}^{*} \\
&\sigma \in \mathit{State} = \mathit{AssignG} \times \mathit{Scope} \times \mathit{Namespace} \times \mathit{ClassHier}
\end{aligned}$$

Fig. 4: Domains of the analysis.

Python code fragment located in a module named main.

```
1 var = 10
2 class A:
3   var = 10
```

The analysis distinguishes the objects created at lines 1 and 3, as the first one resides in the namespace $[(\texttt{main}, \textbf{mod})]$, while the second one lives in the namespace $[(\texttt{main}, \textbf{mod}), (\texttt{A}, \textbf{cls})]$.

Our approach treats every object as the *value* given from the evaluation of the expressions supported by the language. In particular, our representation contains expressions that capture the inter-procedural flow, assignment statements, class and function definitions, module imports, and iterators / generators (see the *Expr* rule). Note that the language is able to abstract different features, including lambda expressions, keyword arguments, constructors, multiple inheritance, and more.

As with prior work focusing on JavaScript [16], [24], [15], we use evaluation contexts [23] that describe the order in which sub-expressions are evaluated. For example, in an attribute assignment $E.x := e$, the E symbol denotes that we are currently evaluating the receiver of the attribute x, while $o.x := E$ indicates that the receiver has been already evaluated to an object $o \in \mathit{Obj}$ (recall that evaluating expressions results in objects), and the evaluation now proceeds to the right-hand side of the assignment.

Remarks. When calling Python functions that produce a generator (i.e., they contain a yield statement instead of return), these calls take place only when the generator is actually used. To model this effect, when encountering such lazy calls (e.g., gen = lazy_call(x)), we create a thunk (e.g., gen = lambda: lazy_call(x)) that is evaluated only when we iterate the generator (through the **iter** construct). Furthermore, dictionaries and lists are treated as regular objects. For example, we model a dictionary lookup x["key"], as an attribute access $x.key$.

2) State: After converting the original Python program to our intermediate representation, our analysis starts evaluating each expression, and gradually constructs the assignment graph. To do so, the analysis maintains a state consisting of four domains as shown in Figure 4, namely, *scope*, *class hierarchy*, *assignment graph*, and *current namespace*.

A scope is a map of definitions to a set of definitions. Conceptually, a scope is a tree where each node corresponds to a definition (e.g., a function), and each edge shows the parent/child relations between definitions, i.e., the target node is defined inside the definition of the source node. The domain of scopes is useful for correctly resolving the definitions that are visible inside a specific namespace. Figure 5a illustrates the scope tree of the program depicted in Figure 1, and shows all program definitions and their inter-relations. Orange nodes correspond to module definitions, red nodes are class definitions, black nodes indicate functions, while blue nodes denote variables. Based on this scope tree, we infer that the function apply is defined inside the class Crypto, which is in turn defined inside the module crypto, i.e., notice the path **crypto** → **Crypto** → **apply** → **func**. This domain enables us to properly deal with Python features such as function closures and nested definitions.

A class hierarchy is a tree representing the inheritance relations among classes. An edge from node u to node v indicates that the class u is a child of the class v. The analysis uses this domain for resolving class attributes (either methods or fields) defined in the base classes of the receiver object. Through this domain we are able to handle the object-oriented nature of Python, addressing features such as multiple inheritance, and the method resolution order.

The assignment graph is defined as a map of objects to an element of the power set of objects $\mathcal{P}(\mathit{Obj})$. This graph holds the assignment relations between objects, capturing the assignments and the inter-procedural flow of the program. Figure 5b illustrates the assignment graph corresponding to the program of Figure 1. Each node in the graph (e.g., {crypto.Crypto.apply, func}) represents an object. The first component of the node label (e.g., crypt.Crypto.apply) indicates the namespace where each identifier (e.g., func) is defined. Colors reveal the type of the identifier as explained in a previous paragraph (e.g., the blue color implies variable definitions). An edge shows the possible values that a variable may hold. For example, the variable func defined in the crypt.Crypto.apply namespace may point to the functions decrypt and encrypt, both defined in the cryptops namespace. As another example, notice the edge originating from the node ({crypto.Crypto.apply, msg} and leading to {crypto, encrypted}. This edge shows that the parameter msg of the function crypto.Crypto.apply points to the variable encrypted when the function is invoked on line 12. The assignment graph domain enables us to address the challenge regarding higher-order programming in Python.

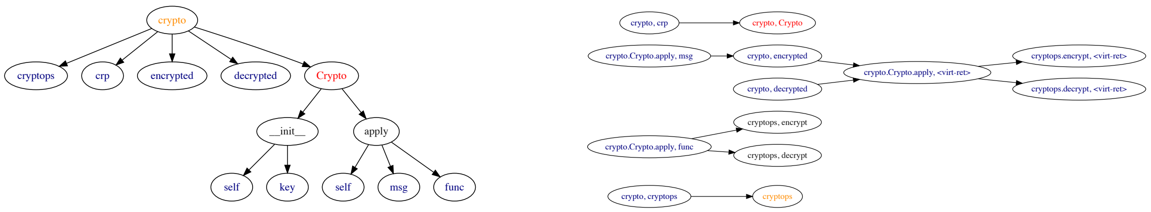
Finally, we use the current namespace to track the location where new variables, classes, modules, and functions are defined. This domain is important for establishing a more precise analysis than field-based analysis employed by prior work. Through namespaces, objects and attribute accesses are distinguished based on their namespace, addressing challenges such as duck typing.

3) Analysis Rules: The analysis examines every expression found in the intermediate representation of the initial program, and transitions the analysis state according to the semantics of each expression. The algorithm repeats this procedure until the state converges, and the assignment graph is given by the final state of the analysis.

Figure 6 demonstrates the state transition rules of our analysis. The rules follow the form:

$$\langle \pi, s, n, h, E[e] \rangle \rightarrow \langle \pi', s', n', h', E[e'] \rangle$$

In the following, we describe each rule in detail.



(a) The scope tree of the `crypto` module.

(b) The assignment graph of the `crypto` module.

Fig. 5: Analyzing the `crypto` module.

E-CTX

$$\frac{\langle \pi, s, n, h, e\rangle \hookrightarrow \langle \pi', s', n', h', e'\rangle}{\langle \pi, s, n, h, E[e]\rangle \rightarrow \langle \pi', s', n', h', E[e']\rangle}$$

COMPOUND

$$\frac{}{\langle \pi, s, n, h, E[o_1; o_2]\rangle \rightarrow \langle \pi, s, n, h, E[o_2]\rangle}$$

IDENT

$$\frac{o = \texttt{getObject}(s, n, x)}{\langle \pi, s, n, h, E[x]\rangle \rightarrow \langle \pi, s, n, h, E[o]\rangle}$$

ASSIGN

$$\frac{\begin{array}{c} s' = \texttt{addScope}(s, n, x, \textbf{var}) \\ o' = (n, (x, \textbf{var})) \qquad \pi' = \pi[o' \rightarrow \pi(o') \cup \{o\}] \end{array}}{\langle \pi, s, n, h, E[x := o]\rangle \rightarrow \langle \pi', s', n, h, E[o']\rangle}$$

FUNC

$$\frac{\begin{array}{c} s' = \texttt{addScope}(s, n, x, \textbf{func}) \\ n' = n \cdot (x, \textbf{func}) \qquad s'' = \texttt{addScope}(s', n', \texttt{ret}, \textbf{var}) \\ s^{(3)} = \texttt{addScope}(s'', n', y, \textbf{var}) \end{array}}{\langle \pi, s, n, h, E[\textbf{function}\ x\ (y \ldots)\ e]\rangle \rightarrow \langle \pi, s^{(3)}, n', h, E[e]\rangle}$$

RETURN

$$\frac{o' = (n \cdot x, (\texttt{ret}, \textbf{var})) \qquad \pi' = \pi[o' \rightarrow \pi(o') \cup \{o\}]}{\langle \pi, s, n \cdot x, h, E[\textbf{return}\ o]\rangle \rightarrow \langle \pi', s, n, h, E[o']\rangle}$$

CALL

$$\frac{\begin{array}{c} o_1 = (n', (f, \textbf{func})) \\ o_2' = (n' \cdot f, (y, \textbf{var})) \qquad \pi' = \pi[o_2' \rightarrow \pi(o_2') \cup \{o_2\}] \end{array}}{\langle \pi, s, n, h, E[o_1(y = o_2, \ldots)]\rangle \rightarrow \langle \pi', s, n, h, (n' \cdot f, (\texttt{ret}, \textbf{var}))\rangle}$$

CLASS

$$\frac{\begin{array}{c} s' = \texttt{addScope}(s, n, x, \textbf{cls}) \qquad t = \langle \texttt{getObject}(\texttt{s}, \texttt{n}, \texttt{b}) \mid b \in (y \ldots)\rangle \\ h' = h[(n, (x, \textbf{cls})) \rightarrow t] \qquad n' = n \cdot (x, \textbf{cls}) \end{array}}{\langle \pi, s, n, h, E[\textbf{class}\ x\ (y \ldots)\ e]\rangle \rightarrow \langle \pi, s', n', h', E[e]\rangle}$$

ATTR

$$\frac{o' = \texttt{getClassAttrObject}(o, x, h)}{\langle \pi, s, n, h, E[o.x]\rangle \rightarrow \langle \pi, s, c, h, E[o']\rangle}$$

NEW

$$\frac{\begin{array}{c} o_3 = \texttt{getObject}(s, n, x) \\ o_2 = \texttt{getClassAttrObject}(o_3, \texttt{\_\_init\_\_}, h) \end{array}}{\langle \pi, s, n, h, E[\textbf{new}\ x(y = o_1 \ldots)]\rangle \rightarrow \langle \pi, s, n, h, E[o_2(y = o_1 \ldots)]; o_3]\rangle}$$

ATTR-ASSIGN

$$\frac{\begin{array}{c} o_3 = \texttt{getClassAttrObject}(o_1, x, h) \\ \pi' = \pi[o_3 \rightarrow \pi(o_3) \cup \{o_2\}] \end{array}}{\langle \pi, s, n, h, E[o_1.x := o_2]\rangle \rightarrow \langle \pi', s, n, h, E[o_3]\rangle}$$

IMPORT

$$\frac{\begin{array}{c} o_2 = \texttt{getObject}(\texttt{s}, \texttt{m}, \texttt{x}) \qquad s' = \texttt{addScope}(s, n, y, \textbf{var}) \\ o_1 = (n, (y, \textbf{var})) \qquad \pi' = \pi[o_1 \rightarrow \pi(o_1) \cup \{o_2\}] \end{array}}{\langle \pi, s, n, h, E[\textbf{import}\ x\ \textbf{from}\ m\ \textbf{as}\ y]\rangle \rightarrow \langle \pi', s', n, h, E[o_1]\rangle}$$

ITER-ITERABLE

$$\frac{o' = \texttt{getClassAttrObject}(o, \texttt{\_\_next\_\_}, h)}{\langle \pi, s, n, h, E[\textbf{iter}\ o]\rangle \rightarrow \langle \pi, s, n, h, E[o'()]\rangle}$$

ITER-GENERATOR

$$\frac{\texttt{getClassAttrObject}(o, \texttt{\_\_next\_\_}, h) = \text{undefined}}{\langle \pi, s, n, h, E[\textbf{iter}\ o]\rangle \rightarrow \langle \pi, s, n, h, E[o()]\rangle}$$

Fig. 6: Rules of the analysis.

According to the [E-CTX] rule, when we have an expression e in the evaluation context E, an assignment graph π, a scope s, a namespace n, a class hierarchy h, we can get an expression e' in the evaluation context E, if the initial expression e evaluates to e'. For what follows, the binary operation $x \cdot y$ stands for appending the element y to the list x.

The [COMPOUND] rule states that when we have a compound expression consisting of two objects o_1, o_2, we return the last object o_2 as the result of the evaluation. Observe that the evaluation of the compound expression requires each subterm to have been evaluated to an object according to the evaluation contexts shown in Figure 3. The rest of the rules also follow this behavior.

The [IDENT] rule describes the scenario when the initial expression is an identifier x. In this case, the analysis retrieves the object o corresponding to the identifier x, in the namespace n, based on the scope tree s. To do so, the analysis uses the function `getObject`(s, n, x), which iterates every element y of the namespace n in the reverse order. Then, by examining the scope tree s, it checks whether the element node y has any child matching the identifier x. In case of a mismatch, the function `getObject` proceeds to the next element of the namespace. Notice that the [IDENT] rule does not have any side-effect on the analysis state.

The [ASSIGN] rule assigns the object o to the identifier x. First, the analysis adds the identifier x in the current namespace n of the scope tree s, using the function `addScope`(s, n, x, τ). This function adds an edge from the node accessed by the path n to the target node given by the definition (x, τ). Second, this rule updates the assignment graph by adding an edge from the object corresponding to the left-hand side of the assignment (i.e., o') to that of the right-hand side (i.e., o). This update says that the variable x defined in the namespace n can point to the object o.

[FUNC] updates the scope tree. In particular, it adds the function x to the current namespace n, leading to a new scope s'. Then, it creates a new namespace n' by adding the function definition $(x, \textbf{func})$ to the top of the current namespace. It adds all function parameters, and a virtual variable named `ret`—which stands for the variable holding the return value of the function—to the newly-created namespace n'. This results in a new scope tree $s^{(3)}$. Finally, the analysis proceeds to the evaluation of the body of the function x in the fresh namespace n', i.e., observe that the rule evaluates to $E[e]$. The new namespace n' correctly captures that any variable defined in

e, is actually defined in the body of the function.

[RETURN] assigns the object o to the virtual variable `ret`, which is used for storing the return value of a function (recall the [FUNC] rule). To do so, the analysis updates the assignment graph by adding a new edge from the object o' corresponding to the return variable `ret` to the object o which is the operand of **return**. Finally, this rule evaluates to the object o' related to the return virtual variable `ret`.

The inter-procedural flow is captured by the [CALL] rule. Specifically, when we encounter a call expression $o_1(x = o_2 \ldots)$, we examine the callee object o_1 associated with a function f defined in a namespace n'. Then, the rule connects every parameter of f with the appropriate argument passed during function invocation (e.g., the counterpart object of the parameter y at call-site is o_2), leading to a new assignment graph π'. As an example, consider again the graph of Figure 5b. The outgoing edges of the {`crypto.Crypto.apply, func`} node are created by this rule. These edges imply that the parameter `func` of the `crypto.Crypto.apply` function may hold the functions `cryptops.encrypt` and `cryptops.decrypt` passed when calling `crypto.Crypto.apply` (Figure 1).

The [CLASS] rule handles class definitions. The rule first adds the class x to the scope tree through the function `addScope()`, and then gets every object related to the base classes of x (i.e., $y \ldots$). To achieve this, the rule consults the scope tree in the namespace n, and gets a sequence of objects t that respects the order in which base classes are passed during class definition. We later explain why keeping the registration order of base classes is important. The rule then updates the class hierarchy so that the freshly-defined class x is a child of the base classes pointed to by the identifiers $(y \ldots)$. After this, the analysis works on the body of the class e in a new namespace n'. The new namespace contains the class definition to the top of the current namespace (i.e., $n \cdot (x, \textbf{cls})$). Then, the analysis starts examining the body of the class using the new namespace.

The [ATTR] rule is similar to [IDENT]. However, this time, in order to correctly retrieve the object corresponding to the attribute x of the receiver object o, the analysis examines the hierarchy of classes h through the function `getClassAttrObject(`o, x, h`)`. This is the point where our analysis is able to distinguish attributes according to the location (i.e., o) where they are defined.

To deal with multiple inheritance, the function `getClassAttrObject()` respects the method resolution order implemented in Python. For example, consider the following code snippet.

```
class A:
  def func():
    pass

class B:
  def func():
     pass

class C(B, A):
    pass

c = C()
c.func()
```

In the example above, the method resolution order is $C \rightarrow B \rightarrow A$, because the class `B` is the fist parent class of `C`, while `A` is the second one. As a result, `c.func()` leads to the invocation of function `func` defined in class `B`, as it is the first matching function whose name is `func` in the method resolution order. Correctly resolving class members explains why the domain of the class hierarchy maps every object to a *sequence* of objects rather than a set—we need to track the order in which the parents of a class are registered.

For object initialization, we introduce the [NEW] rule. This rule gets the object o_3 associated with the definition of the class x. Using the `getClassAttrObject()` function, the rule inspects the method resolution order of the object o_3 to find the first object o_2 matching the function `__init__`. Recall that this function is called whenever a new object is created. Observe how the **new** evaluates; it reduces to $o_2(y = o_1); o_3$. That is, we first call the constructor of the class with the same arguments passed as in the initial expression (i.e., $o_2(y = o_1)$), and then we return the object o_3 corresponding to class definition, which is eventually the result of the **new** expression.

The rule for attribute assignment $o_1.x := o_2$ describes the case when the attribute x is defined somewhere in the class hierarchy of the receiver object o_1. In this case, `getClassAttrObject()` returns the object o_3 associated with this attribute, and the rule updates the assignment graph so that o_3 points to the object o_2 from the right-hand side of the assignment. If the attribute is not defined in the class hierachy, (i.e., `getClassAttrObject()` returns $\perp$) the attribute assignment is similar to [ASSIGN], i.e., we first add the attribute x to the current scope through `addScope()`, and then update the graph. This case is omitted for brevity.

When we encounter an **import** x **from** m **as** y expression, we retrieve the object o_2 corresponding to the imported identifier x, which is defined in the module m. Then, we create an alias y for x. To do so, we add y to the scope tree of the current namespace, and update the assignment graph by adding an edge from the object of y to that of x. Through this rule, we are able to deal with Python's module system.

Consuming iterables and generators is supported through the **iter** x expression. When the identifier x points to an iterable, (i.e., the object pointed to by x has an attribute named `__next__`), we get the object o' related to `__next__`. Then, **iter** evaluates to a call of $o'()$ (see the [ITER-ITERABLE] rule). If this is not the case, we treat x as a generator ([ITER-GENERATOR]). In this case, **iter** reduces to a call of $x()$. Recall from Section III-A1 that we model generators as thunks, therefore this scenario describes the evaluation of these thunks (generators) when they are actually used (iterated).

Remark about analysis termination. The analysis traverses expressions, and transitions the analysis state based on the rules of Figure 6, until the state converges. The analysis is guaranteed to terminate, because the domains are

Algorithm 1: Call Graph Construction

```
Input : $p \in Program$
        $\sigma \in State$
Output: $cg \in CallGraph$
1  foreach $e$ in $Program$ do
2      while $e \notin Obj$ do
3          $\langle \sigma, E[e] \rangle \to \langle \sigma', E[e'] \rangle$
4          if $e' = o_1(y = o_2 \dots)$ then // Call Expression
5              $(\pi, s, n \cdot f, h) \leftarrow \sigma'$
6              $c \leftarrow$ getReachableFuns($\pi$, $o_1$)
7              $o_3 \leftarrow$ getObject($s$, $n$, $f$)
8              $cg \leftarrow cg[f \to cg(f) \cup c]$ // Add Call Edges
9          end
10         $e \leftarrow e'$
11     end
12 end
13 return $cg$
```

finite. Even in the presence of the domain of class hierarchy $h \in ClassHier$ (Figure 4), which is theoretically infinite, the analysis eventually terminates, because a Python program cannot have an unbounded number of classes.

B. Call Graph Construction

After the termination of the analysis, we build the call graph by performing a final pass on the intermediate representation of the given Python program. Algorithm 1 describes the details of this pass. The algorithm takes two elements as input: (1) a program $p \in \mathit{Program}$ of the model language whose syntax is shown in Figure 3, and (2) the final state $\sigma \in \mathit{State}$ stemming from the analysis step. The algorithm produces a call graph:

$$cg \in \mathit{CallGraph} = \mathit{Obj} \hookrightarrow \mathcal{P}(\mathit{Obj})$$

The graph contains only objects associated with functions. An element $o \in \mathit{Obj}$ mapped to a set of objects $t \in \mathcal{P}(\mathit{Obj})$ means that the function o *may* call any function included in t.

The algorithm inspects every expression e found in the program (line 1), and it evaluates e based on the state transition rules described in Figure 6. The algorithm repeats the state transition rules, until e eventually reduces to an object (lines 2, 3). Every time when e reduces to a call expression of the form $o_1(y = o_2 \dots)$ (line 4), the algorithm gets the namespace where this invocation happens and retrieves the top element of that namespace (see $n \cdot f$, line 5). After that, the algorithm gets all functions that the callee object o_1 may point to. To do so, it consults the assignment graph through the function `getReachableFuns`(π, o_1), which implements a simple Depth-First Search (DFS) algorithm and gets the set of functions c that are reachable from the source node o_1. In turn, the algorithm updates the call graph cg by adding all edges from the top element of the current namespace to the set of the callee functions c (lines 7, 8). In other words, the object o_3 (line 7) representing the top element of the namespace, where the call occurs, is actually the caller of the functions pointed to by the object o_1.

C. Discussion & Limitations

One of our major design decisions is to ignore conditionals and loops. For instance, when we come across an `if` statement, our analysis over-approximates the program's behavior and considers both branches. This design choice enables efficiency without highly compromising the analysis precision (as we will discuss in Section V). Other static analyzers [10], [11], [9] choose to follow a more heavyweight approach and reason about conditionals. These static analyzers, though, do not solely focus on call-graph construction, but rather they attempt to compute the set of all reachable states based on an initial one. However, for call-graph generation, providing such an initial state that exercises all feasible paths (which is required in order to compute a complete call graph), especially when analyzing libraries, is not straightforward.

In Python where object-oriented features (e.g., dynamic method dispatch), duck typing [22], and modules are extensively used, it is important to separate attribute accesses based on the namespace where each attribute is defined. This design choice boosts—contrary to prior work [14]—the precision of our analysis without sacrificing its scalability.

Our analysis does not fully support all of Python's features. First, we ignore code generation schemes, such as calls to the `eval` built-ins. In general, such dynamic constructs hinder the effectiveness of any static analysis, and dynamic approaches are often employed as a countermeasure [25], [26]. Second, PyCG does not store information about variables' built-in types. Therefore, attribute calls that depend on a specific built-in type (e.g., `list.append()`) are not resolved. Third, PyCG can only analyze modules for which their source code has been provided. When a function—for which its code definition is not available—is called, PyCG will add an edge to the function, but no edges stemming from that function will ever be added, and its return value will be ignored.

IV. Implementation

We have developed PyCG, a prototype of our approach in Python 3, which we plan to make publicly available. For each input module, our tool creates its scope tree and its intermediate representation by employing the *symtable* [27] and *ast* [28] modules respectively.

Our prototype discovers the file locations of the different imported modules to further analyze them by using Python's *importlib* module. This is the module that Python uses internally to resolve import statements. We perform two steps. First, the file location of the imported module is identified, and then a *loader* is used to import the module's code. In Python one can define custom loaders for import statements, which allowed us to use a loader that logs the file locations discovered and then exit without loading the code. Then, in the second step, our tool takes over and uses the discovered file's contents to iterate its intermediate representation in a recursive manner. This allows us to resolve imports in an efficient way. Currently, we only analyze discovered modules that are contained in the package's namespace, and ignore third party imports.

V. Evaluation

We evaluate our approach based on three research questions:

TABLE I: Micro-benchmark suite categories.

Category	#tests	Description
parameters	6	Positional arguments that are functions
assignments	4	Assignment of functions to variables
built-ins	3	Calls to built in functions and data types
classes	22	Class construction, attributes, methods
decorators	7	Function decorators
dicts	12	Hashmap with values that are functions
direct calls	4	Direct call of a returned function (func()())
exceptions	3	Exceptions
functions	4	Vanilla function calls
generators	6	Generators
imports	14	Imported modules, functions classes
kwargs	3	Keyword arguments that are functions
lambdas	5	Lambdas
lists	8	Lists with values that are functions
mro	7	Method Resolution Order (MRO)
returns	4	Returns that are functions

RQ1 Is the proposed approach effective in constructing call graphs for Python programs? (Sections V-B and V-C)

RQ2 How does the proposed approach stand in comparison with existing open-source, static-based approaches for Python? (Sections V-B and V-C)

RQ3 What is the performance of our approach? (Section V-D)

Further, we show a potential application through the enhancement of GitHub's "security advisory" notification service.

A. Experimental Setup

We use two distinct benchmarks: (1) a micro-benchmark suite containing 112 minimal Python programs, and (2) a macro-benchmark suite of five popular real-world Python packages. We ran our experiments on a Debian 9 host with 16 CPUs and 16 GBs of RAM.

1) Micro-benchmark Suite: We propose a test suite for benchmarking call graph generation in Python. Based on this suite, researchers can evaluate and compare their approaches against a common standard. Reif et al. [29] have provided a similar suite for Java, containing unique call graph test cases, grouped into different categories.

Our suite consists of 112 unique and minimal micro-benchmarks that cover a wide range of the language's features. We organize our micro-benchmarks into 16 distinct categories, ranging from simple function calls to more complex features such as twisted inheritance schemes. Each category contains a number of tests. Every test includes (1) the source code, (2) the corresponding call graph (in JSON format), and (3) a short description. Categorizing and adding a new test is relatively easy. The source code of each test implements only a single execution path (i.e., no conditionals and loops) so there is a straightforward correspondence to its call graph. Table I lists the categories along with the number of benchmarks they incorporate and a corresponding description.

Addressing Validity Threats: The internal validity of the micro-benchmark suite depends on the range of Python features that it covers. To address this threat, we presented the suite to two researchers, who have professionally worked as Python developers (other researchers have applied similar methods to verify their work [30]). Then, we asked them to rank the suite (from 1 to 10) based on the following criteria:

1) Completeness: Does it cover all Python features?
2) Code Quality: Are the tests unique and minimal?
3) Description Quality: Does the description adequately describe the given test case?

TABLE II: Macro-benchmark suite project details.

Project	LoC	Stars	Forks	Description
fabric	3,236	12.1k	1.8k	Remote execution & deployment
autojump	2,662	10.8k	530	Directory navigation tool
asciinema	1,409	7.9k	687	Terminal session recorder
face_classification	1,455	4.7k	1.4k	Face detection & classification
Sublist3r	1,269	4.4k	1.1k	Subdomains enumeration tool

TABLE III: Micro-benchmark results for PyCGand Pyan. Depends is incomplete and sound in all cases and is omitted.

Category	PyCG		Pyan	
	Complete	Sound	Complete	Sound
assignments	4/4	3/4	4/4	4/4
built-ins	3/3	1/3	2/3	0/3
classes	22/22	22/22	6/22	10/22
decorators	6/7	5/7	3/7	3/7
dicts	12/12	10/12	6/12	6/12
direct calls	4/4	4/4	0/4	0/4
exceptions	3/3	3/3	0/3	0/3
functions	4/4	4/4	4/4	3/4
generators	6/6	6/6	0/6	0/6
imports	14/14	14/14	10/14	4/14
kwargs	3/3	3/3	0/3	0/3
lambdas	5/5	5/5	4/5	0/5
lists	8/8	7/8	3/8	4/8
mro	7/7	5/7	0/7	2/7
parameters	6/6	6/6	0/6	0/6
returns	4/4	4/4	0/4	0/4
Total	**111/112**	**102/112**	**42/112**	**36/112**

The first reviewer provided a 9.7 ranking in all cases. The second indicated an excellent (10) code and description quality but ranked lower (6) the completeness of the benchmarks.

Both reviewers provided corresponding feedback. In their comments, they suggested some code cleanups and asked for more comprehensive descriptions on some complex benchmarks. Regarding the completeness of the suite, they pointed out missing tests for some common features such as built-in functions, magic methods and generators. We applied the reviewers' suggestions by refactoring the affected benchmarks and improving their descriptions. Furthermore, we implemented more tests for some of the missing functionality.

2) Macro-benchmarks: We have manually generated call graphs for five popular real-world packages. The packages were chosen as follows. First, we queried the GitHub API for Python repositories sorted by their number of stars. Then, we downloaded each repository and counted the number of lines of Python code. If the repository contained less than 3.5k lines of Python code, we kept it. Table II presents the GitHub repositories we chose along with their lines of code, GitHub stars and forks, together with a short description.

Currently, there is no acceptable implementation generating Python call graphs in an effective manner, so the first author manually inspected the projects and generated their call graphs in JSON format. We opted for medium sized projects (less than 3.5k LoC), so that we could minimize human errors.

B. Micro-benchmark suite results

The benchmarks included in the micro-test suite have a limited scope and are designed to cover specific functionalities (such as decorators and lambdas). Table III lists the results of our evaluation. For each benchmark belonging to a specific category, we show if our prototype and Pyan generated complete or sound call graphs. Note that a call graph is complete when it does not contain any call edges that do not actually exist (no false positives), and sound when it contains every call edge that is realized (no false negatives).

PyCG produces a complete call graph in almost all cases (111/112). In addition, it produces sound call graphs for 102 out of 112 benchmarks. The lack of soundness is attributed to not fully covered functionalities, i.e. Python's starred assignments and type coercion in dictionaries.

Pyan produces either complete or sound call graphs at a much lower rate. However, for assignments, Pyan turns out as a more sound method because it supports them in a better manner. We performed a qualitative analysis on the call graphs generated by Pyan to check the reasons behind the performance of Pyan. We observed that Pyan produces incomplete graphs because it creates call edges to class names as well as their `__init__` methods (see also Section II-B). Also it generates imprecise results because it does not support all of Python's functionality, ($0/6$ generators and $0/3$ exceptions), ignores the inter-procedural flow of functions ($0/6$ parameters and $0/4$ returns), misses calls to imported ones ($4/14$), and fails to support classes ($10/22$).

The evaluation of *Depends* shows both its fundamental strengths and limitations. Recall that each benchmark implements a single execution path, all calls coming from the module's namespace are always included. Our results indicate that *Depends* does not identify calls from module namespaces, and therefore completeness is never achieved (0/112). In terms of soundness, *Depends* achieves a perfect score (112/112) due to its conservative nature—i.e., it adds an edge when it has high confidence that it will be realized.

C. Macro-benchmark results

By using our macro-benchmark, we have examined the three tools in terms of precision and recall. Precision measures the percentage of valid generated calls over the total number of generated calls. Recall measures the percentage of valid generated calls over the total number of calls. To do so, we manually generated the call graphs of the examined packages.

Table IV presents our results. The missing entries for Pyan indicate that the tool crashed during the execution. Our findings show that PyCG generates high precision call graphs. On all cases, more than 97% of the generated call edges are true positives, while on one case none of the generated call edges are false positives. Recall results show that on average, 69.3% of all call edges are successfully retrieved. The missing call edges are attributed to the approach's limitations (recall Section III-C), and missing support for some functionalities.

Pyan shows average precision and low recall. Pyan's average precision appears because the tool adds call edges to class names instead of just their `__init__()` functions. Also, it does not track the inter-procedural flow of functions, which is the reason why it has low recall. For instance, the implementation of the `face_classification` package mostly depends on functions declared in external packages. Pyan ignores such calls which in turn leads to a 7.5% recall.

TABLE IV: Macro-benchmark results and tool comparison.

Project	Precision (%)			Recall (%)		
	PyCG	**Pyan**	**Depends**	**PyCG**	**Pyan**	**Depends**
`autojump`	99.7	66	99.2	66	28.3	22.5
`fabric`	97.5	-	100	62	-	6.3
`asciinema`	100	-	98.1	68	-	15.5
`face_classification`	98.5	86.8	96.1	88.7	7.5	5.7
`Sublist3r`	99.6	69.8	100	61.6	25.6	21.9
Average	**99**	**74.2**	**98.7**	**69.3**	**20.5**	**14.4**

TABLE V: Time and memory comparison.

Project	Time (sec)			Memory (MB)		
	PyCG	**Pyan**	**Depends**	**PyCG**	**Pyan**	**Depends**
`autojump`	0.52	0.42	2.37	62.7	37.8	27.1
`fabric`	0.6	-	1.83	60.9	-	18.5
`asciinema`	0.68	-	2	61.6	-	19.4
`face_classification`	0.75	0.38	2.49	60.9	35.3	25.6
`Sublist3r`	0.44	0.33	2.01	60	35.8	19.4
Average	**0.6**	**0.38**	**2.14**	**61.2**	**36.3**	**22**

Finally, *Depends* shows high precision (98.7%) and low recall. The high precision of *Depends* can be attributed to its conservative nature. Furthermore, *Depends* does not track higher order functions and does not include calls coming from module namespaces. This in turn, leads to its low recall.

D. Time and Memory Performance

We use the macro-benchmark suite as a base for our time and memory evaluation. Table V presents the time and memory performance metrics of the three tools. The execution time was calculated using the UNIX *time* command, while the memory consumption was measured using the UNIX *pmap* command. The metrics presented are the average out of 20 runs.

The results show that Pyan is more time efficient, and that *Depends* is more memory efficient having faster execution times and memory usage than the other tools. PyCG and Pyan generate a call graph for the programs in the benchmark ($\leq$ 3.5k LoC) in under a second, while *Depends* requires more than two seconds on average. Furthermore, all tools use a reasonable amount of memory, with PyCG, Pyan and *Depends* using on average $\sim$61.4, $\sim$36.3 and $\sim$22MBs of memory respectively. Overall, PyCG is on average 1.5 times slower than Pyan, and uses 2.8 times the amount of memory that *Depends* uses. We attribute the differences in execution time between Pyan and PyCG to the fact that Pyan performs two passes of the AST in comparison to PyCG performing a fixpoint iteration (Section III). *Depends* is overall slower, because it spends most of its execution time parsing the source files. In terms of memory, Pyan and *Depends* store less information about the state of the analysis leading to better memory performance.

E. Case Study: A Fine-grained Tracking of Vulnerable Dependencies

GitHub sends a notification to the contributors of a repository when it identifies a dependency to a vulnerable library.

However, this notification does not indicate if the project invokes the function containing the defect. We show that PyCG can be employed to enhance the service with method-level information that may further warn the contributors.

To highlight the usefulness of our method in this context, we performed the following steps. First we accessed GitHub's "Advisory Database" [31]. Then, we searched for vulnerable Python packages sorted by the severity of the defect. In many occasions the accompanying CVE (Common Vulnerabilities and Exposures) entries did not include further details about the defects. We disregarded such instances and focused on the first two cases that provided information about the functions that contained the vulnerability: (1) PyYAML [32] (versions before 5.1), a YAML parser affected by CVE-2017-18342 [33], and (2) Paramiko [34] (multiple versions before 2.4.1), an implementation of the SSHv2 protocol affected by CVE-2018-7750 [35]. Both packages were imported by thousands of projects, 9226 for PyYAML and 1097 for Paramiko. We could not clone all dependent repositories because some were private and others did not exist any more: we managed to download 570 PyYAML and 322 Paramiko dependent projects. Then, we ran our tool on each project and generated corresponding call graphs for 106 out of the 570 PyYAML dependent projects and 76 out of the 322 Paramiko dependent projects—the projects that PyCG failed to generate call graphs were written in Python 2. Finally, we queried the generated call graphs to check if the vulnerable functions were included. We found that the vulnerable function in PyYAML (i.e., `load`) was invoked by 42/106 projects. In Paramiko we found that the problem method (`start_server`), was not utilized at all by any of the 76 projects. We also observed that 12 projects did not invoke any library coming from Paramiko. Paramiko was needlessly included in the requirement files of the dependents. That was not a false negative from our part: we manually checked that PyCG did not miss any invocation.

VI. Related Work

Call Graph Generation. Methods that generate call graphs can be either dynamic [36], or static [37]. Dynamic approaches usually produce fewer false positives, but suffer from performance issues. Also, they are able to analyze a single execution path, and their effectiveness relies on the program's input. Static approaches are more time efficient and can typically cover a wider range of execution paths, trying to capture all possible program's behaviors. Several approaches [38], [39], [40], try to combine the two so they can get improved results.

There are plenty of methods and tools targeting call graph generation for statically-typed programming languages such as Java. DOOP [41] and WALA [42] follow a context-sensitive, points-to analysis method. PADDLE [43], a similar approach, employs Binary Decision Diagrams (BDDs) [44]. Finally, OPAL [45] is a lattice-based approach written in Scala. Ali et al. [46], implement CGC, a partial call graph generator for Java, with the main focus being efficiency. They ignore calls coming from externally imported libraries, and only analyze the source code of a given package. We are currently following a similar approach, but we aim to efficiently analyze external dependencies in the future.

Moving to dynamic languages, Ali et al. [47] convert Python source code into JVM bytecode, and use the existing implementations for Java [42], [48], [49] to generate its call graph. However, they argue that generating precise call graphs using this method is infeasible, and sometimes the output has more than 96% of false positives. *pycallgraph* [50] generates Python call graphs by dynamically analyzing one execution path. Thus the analysis is not practical and one should pair it with another method (e.g., fuzzing) to retrieve meaningful results. On the JavaScript front, Feldthaus et al. [14] implement a flow-based approach for the generation of call graphs. They evaluate against call graphs generated by a dynamic approach paired with instrumentation, achieving $\geq 66\%$ precision and $\geq 85\%$ recall. Other JavaScript call graph generators include, IBM WALA [42], NPM call graph [51], Google closure compiler [52], Approximate Call Graph (ACG) [14], and Type Analyzer for JavaScript (TAJS) [9]. TAJS implements a lattice-based flow-sensitive approach using abstract interpretation. Although, such an approach yields more promising results, it comes with a performance cost.

Call Graph Benchmarking and Comparison. Reif et al. present Judge [29], a toolchain for analyzing call graph generators for Java. At its core, the toolchain contains a test suite with benchmarks for a range of Java features. The authors then proceed to compare Java call graph generators, namely Soot [48], [49], WALA [42], DOOP [41] and OPAL [45]. Sui et al. [53], also present a test suite of Java benchmarks, and they use it to evaluate and compare Soot [48], [49], WALA [42], and DOOP [41]. The above benchmark suites are very similar, leading to Judge consolidating them into one benchmark suite. Recall our very similar implementation of a micro-benchmark suite from Section V-A.

Static Analysis for Dynamic Languages. Numerous advanced frameworks aim for the static analysis of JavaScript programs. SAFE [10] provides a formally specified static analysis framework with the goal of being flexible, scalable and pluggable. JSAI [11] is a formally specified and provably sound platform using abstract interpretation.

Other JavaScript approaches target different aspects of its functionality. Madsen et al. implement RADAR [54] a tool that identifies bugs in event-driven JavaScript programs. Sotiropoulos et al. [15] propose an analysis targeting asynchronous functions. Bae et al. [55], implement $\text{SAFE}_{\text{WAPI}}$ a tool aimed at identifying possible API misuses. Park et al. [56] propose $\text{SAFE}_{\text{WApp}}$, a static analyzer for client-side JavaScript.

Fromherz et al. [57] implement a prototype that soundly identifies run-time errors by evaluating the data types of Python variables through abstract interpretation. In comparison, our approach does not infer the data types of variables and focuses on the generation of call graphs.

VII. Conclusion

We have introduced a practical static approach for generating Python call graphs. Our method performs a context-insensitive inter-procedural analysis that identifies the flow of values through the construction of a graph that stores all assignment relationships among program identifiers. We used two benchmarks to evaluate our method, namely a micro- and a macro-benchmark suite. Our prototype showed high rates of both precision and recall. Also, our micro-benchmark suite can serve as a standard for the evaluation of future methods. Finally, we applied our approach in a real-world case scenario, to highlight how it can aid dependency impact analysis.

References

[1] Valgrind, "Callgrind: a call-graph generating cache and branch prediction profiler," 2020. [Online]. Available: http://valgrind.org/docs/manual/cl-manual.html

[2] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," *ACM Comput. Surv.*, vol. 44, no. 3, Jun. 2012.

[3] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, "Tool-supported refactoring for JavaScript," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 119–138.

[4] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '18. New York, NY, USA: ACM, 2018, pp. 101–104.

[5] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. IEEE Press, 2017, pp. 102–112.

[6] (2016) The npm blog: changes to npm's unpublish policy. [Online; accessed 26-July-2020]. [Online]. Available: https://blog.npmjs.org/post/141905368000/changes-to-npms-unpublish-policy

[7] (2020) npm(1)—a JavaScript package manager. [Online; accessed 26-July-2020]. [Online]. Available: https://github.com/npm/cli

[8] (2020) pip 20.0.2: The PyPA recommended tool for installing Python packages. [Online; accessed 26-July-2020]. [Online]. Available: https://pypi.org/project/pip/

[9] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *International Static Analysis Symposium*. Springer, 2009, pp. 238–255.

[10] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript," in *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*. Citeseer, 2012, p. 96.

[11] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A static analysis platform for JavaScript," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 121–132.

[12] Y. Ko, H. Lee, J. Dolby, and S. Ryu, "Practically tunable static analysis framework for large-scale JavaScript applications," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, pp. 541–551.

[13] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of javascript applications in the presence of frameworks and libraries," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, pp. 499–509.

[14] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for JavaScript IDE services," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 752–761.

[15] T. Sotiropoulos and B. Livshits, "Static analysis for asynchronous JavaScript programs," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. F. Donaldson, Ed., vol. 134. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 8:1–8:30. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10800

[16] M. Madsen, F. Tip, and O. Lhoták, "Static analysis of event-driven node.js JavaScript applications," *SIGPLAN Not.*, vol. 50, no. 10, pp. 505–519, Oct. 2015.

[17] GitHub, "The state of the octoverse," https://octoverse.github.com/, 2019, [Online; accessed 09-January-2020].

[18] D. Fraser, E. Horner, J. Jeronen, and P. Massot, "Pyan3: Offline call graph generator for Python 3," https://github.com/davidfraser/pyan, 2018, [Online; accessed 09-January-2020].

[19] G. Gharibi, R. Tripathi, and Y. Lee, "Code2graph: Automatic generation of static call graphs for Python source code," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 880–883.

[20] G. Gharibi, R. Alanazi, and Y. Lee, "Automatic hierarchical clustering of static call graphs for program comprehension," in *IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018*. IEEE, 2018, pp. 4016–4025.

[21] G. Zhang and J. Wuxia, "Depends is a fast, comprehensive code dependency analysis tool," https://github.com/multilang-depends/depends, 2018, [Online; accessed 04-August-2020].

[22] N. Milojkovic, M. Ghafari, and O. Nierstrasz, "It's duck (typing) season!" in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 312–315.

[23] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics engineering with PLT Redex*. Mit Press, 2009.

[24] M. Madsen, O. Lhoták, and F. Tip, "A model for reasoning about JavaScript promises," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133910

[25] S. Guarnieri and B. Livshits, "GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. USA: USENIX Association, 2009, pp. 151–168.

[26] C.-A. Staicu, M. Pradel, and B. Livshits, "SYNODE: Understanding and automatically preventing injection attacks on Node. js." in *NDSS*, 2018.

[27] (2020) symtable. [Online; accessed 20-July-2020]. [Online]. Available: https://docs.python.org/3/library/symtable.html

[28] (2020) AST in Python. [Online; accessed 20-July-2020]. [Online]. Available: https://docs.python.org/3/library/ast.html

[29] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "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*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 251–261.

[30] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, pp. 164–175. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00033

[31] (2020) GitHub advisory database. [Online; accessed 20-July-2020]. [Online]. Available: https://github.com/advisories

[32] (2020) PyYAML: The next generation YAML parser and emitter for Python. [Online; accessed 20-July-2020]. [Online]. Available: https://github.com/yaml/pyyaml/

[33] (2017) CVE-2017-18342. [Online; accessed 20-July-2020]. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-18342

[34] (2020) Paramiko: The leading native Python SSHv2 protocol library. [Online; accessed 20-July-2020]. [Online]. Available: https://github.com/paramiko/paramiko/

[35] (2018) CVE-2018-7750. [Online; accessed 20-July-2020]. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-7750

[36] T. Xie and D. Notkin, "An empirical study of Java dynamic call graph extractors," *University of Washington CSE Technical Report 02-12*, vol. 3, 2002.

[37] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 158–191, 1998.

[38] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society, 2001, p. 602.

[39] N. Grech, G. Fourtounis, A. Francalanza, and Y. Smaragdakis, “Heaps don’t lie: Countering unsoundness with heap snapshots,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[40] J. Liu, Y. Li, T. Tan, and J. Xue, “Reflection analysis for Java: Uncovering more reflective targets precisely,” in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 12–23.

[41] M. Bravenboer and Y. Smaragdakis, “Strictly declarative specification of sophisticated points-to analyses,” in *ACM SIGPLAN Notices*, vol. 44, no. 10. ACM, 2009, pp. 243–262.

[42] S. Fink and J. Dolby, “WALA—the T.J. Watson libraries for analysis,” 2012.

[43] O. Lhoták and L. Hendren, “Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 1, p. 3, 2008.

[44] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, “Points-to analysis using BDDs,” *SIGPLAN Not.*, vol. 38, no. 5, pp. 103–114, May 2003.

[45] M. Eichberg, F. Kübler, D. Helm, M. Reif, G. Salvaneschi, and M. Mezini, “Lattice based modularization of static analyses,” in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ser. ISSTA ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 113–118.

[46] K. Ali and O. Lhoták, “Application-only call graph construction,” in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 688–712.

[47] K. Ali, X. Lai, Z. Luo, O. Lhotak, J. Dolby, and F. Tip, “A study of call graph construction for JVM-hosted languages,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[48] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot: A Java bytecode optimization framework,” in *CASCON First Decade High Impact Papers*, ser. CASCON ’10. USA: IBM Corp., 2010, pp. 214–224.

[49] O. Lhoták and L. Hendren, “Scaling Java points-to analysis using SPARK,” in *International Conference on Compiler Construction*. Springer, 2003, pp. 153–169.

[50] GitHub user gak, “pycallgraph is a Python module that creates call graphs for Python programs.” https://github.com/gak/pycallgraph, 2014, [Online; accessed 09-January-2020].

[51] G. Gessner, “npm call graph,” https://www.npmjs.com/package/callgraph, 2019, [Online; accessed 09-January-2020].

[52] M. Bolin, *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript*. ” O’Reilly Media, Inc.”, 2010.

[53] L. Sui, J. Dietrich, M. Emery, S. Rasheed, and A. Tahir, “On the soundness of call graph construction in the presence of dynamic language features—a benchmark and tool evaluation,” in *Asian Symposium on Programming Languages and Systems*. Springer, 2018, pp. 69–88.

[54] M. Madsen, F. Tip, and O. Lhoták, “Static analysis of event-driven Node.js JavaScript applications,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 505–519.

[55] S. Bae, H. Cho, I. Lim, and S. Ryu, “SAFEWAPI: Web API misuse detector for web applications,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 507–517.

[56] C. Park, S. Won, J. Jin, and S. Ryu, “Static analysis of JavaScript web applications in the wild via practical DOM modeling,” in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’15. IEEE Press, 2015, pp. 552–562.

[57] A. Fromherz, A. Ouadjaout, and A. Miné, “Static value analysis of Python programs by abstract interpretation,” in *NASA Formal Methods Symposium*. Springer, 2018, pp. 185–202.