PyAnalyzer: An Effective and Practical Approach for Dependency Extraction from Python Code

Wuxia Jin*†
Xi'an Jiaotong University
Xi'an, China
jinwuxia@mail.xjtu.edu.cn

Shuo Xu*†
Xi'an Jiaotong University
Xi'an, China
spoon1116@stu.xjtu.edu.cn

Dawei Chen*†
Xi'an Jiaotong University
Xi'an, China
thisrabbit@stu.xjtu.edu.cn

Jiajun He*†
Xi'an Jiaotong University
Xi'an, China
znzz_hjj@stu.xjtu.edu.cn

Dinghong Zhong*†
Xi'an Jiaotong University
Xi'an, China
ahong_934@163.com

Ming Fan†
Xi'an Jiaotong University
Xi'an, China
mingfan@mail.xjtu.edu.cn

Hongxu Chen
Huawei Technologies Co., Ltd
Shenzhen, China
chenhongxu5@huawei.com

Huijia Zhang
Huawei Technologies Co., Ltd
Shenzhen, China
zhanghuijia1@huawei.com

Ting Liu†
Xi'an Jiaotong University
Xi'an, China
tingliu@mail.xjtu.edu.cn

ABSTRACT

Dependency extraction based on static analysis lays the groundwork for a wide range of applications. However, dynamic language features in Python make code behaviors obscure and nondeterministic; consequently, it poses huge challenges for static analyses to resolve symbol-level dependencies. Although prosperous techniques and tools are adequately available, they still lack sufficient capabilities to handle object changes, first-class citizens, varying call sites, and library dependencies. To address the fundamental difficulty for dynamic languages, this work proposes an effective and practical method namely PyAnalyzer for dependency extraction. PyAnalyzer uniformly models functions, classes, and modules into first-class heap objects, propagating the dynamic changes of these objects and class inheritance. This manner better simulates dynamic features like duck typing, object changes, and first-class citizens, resulting in high recall results without compromising precision. Moreover, PyAnalyzer leverages optional type annotations as a *shortcut* to express varying call sites and resolve library dependencies on demand. We collected two micro-benchmarks (278 small programs), two macro-benchmarks (59 real-world applications), and 191 real-world projects (10MSLOC) for comprehensive comparisons with 7 advanced techniques (i.e., Understand, Sourcetrail, Depends, ENRE19, PySonar2, PyCG, and Type4Py). The results demonstrated that PyAnalyzer achieves a high recall and hence improves the F_1 by 24.7% on average, at least 1.4x faster without an obvious compromise of memory efficiency. Our work will benefit diverse client applications.

KEYWORDS

Dependency Extraction, Python, Dynamic Features

ACM Reference Format:
Wuxia Jin, Shuo Xu, Dawei Chen, Jiajun He, Dinghong Zhong, Ming Fan, Hongxu Chen, Huijia Zhang, and Ting Liu. 2024. PyAnalyzer: An Effective and Practical Approach for Dependency Extraction from Python Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3640325

1 INTRODUCTION

Dependency extraction [22] lays the groundwork for a wide range of static analysis applications such as program comprehension [56], bug localization [6, 8, 39], vulnerability analysis [52], quality measurement [4, 44], dependency smell and architectural anti-pattern detection [13, 31, 54]. It aims at resolving the dependency relations between entities from the source code. Dependencies from entity e_i to e_j indicate that e_i would be updated accordingly if e_j is altered [5]. Entities can be high-level structural elements such as *modules*, *packages*, and *classes*, and fine-grained symbols like *functions* and *variables*. Dependencies include *import*, *inherit*, *call*, *define*, *use*, etc. They can be categorized into *deterministic dependencies* (i.e., explicit dependencies) that are explicitly manifested in the source code and *nondeterministic dependencies* (i.e., possible dependencies) that are invisible and only can be determined at runtime due to dynamic features [21, 22, 24]. For dynamic languages, a complete and sound static analysis comes with a heavy performance cost given the need for modeling and approximating runtime behaviors [53]. Therefore, effective and practical static techniques for dependency extraction are always favored by their downstream applications in parsing real-world projects.

*The School of Software Engineering, Xi'an Jiaotong University.
†Ministry of Education Key Lab for Intelligent Networks and Network Security, Xi'an Jiaotong University.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3640325

Both industries and researchers have invested great efforts in designing and building tools for dependency extraction. Compared to dynamic analysis which requires planning sufficient test cases to expose program behaviors [58], static analysis is less expensive since it over-approximates code behaviors without code executions [55]. A plethora of static-based dependency extraction tools have been invented. A mature proprietary tool named SciTools Understand [63], open-source Sourcetrail [57], Depends [38], and ENRE (labeled as ENRE19) [22] support statically analyzing multiple languages like Java and C++. Recently, the analysis of Python has received tremendous attention due to the language's popularity. Google employs Pysonar1.0 (an early version of PySonar2 [64]) for code search and analysis services. Pyan [61], Code2Graph [59], and PyCG [53] generate call dependencies. PyCG is the most recent work that computes an assignment graph to reason about dependencies.

Despite prosperous methods and tools, they still suffer limitations due to the dynamic features of "everything is an object" and "dynamic typing" in Python. These features make code behaviors implicit and nondeterministic, leading to false negatives and false positives in static analysis. We summarize four major issues that dominantly challenge existing solutions for Python (see Section 2). ❶ The *object change*, which allows arbitrarily altering attributes of class objects, still has not been well-resolved [1]. ❷ Functions, classes, and modules are *first-class citizens*, which can be passed as arguments, returned as values, and assigned on demand [43]. Prior methods always consider first-class functions while ignoring first-class classes and first-class modules. ❸ The signatures of one dependent entity might be different in *varying call sites*, while few methods distinguish between them. ❹ For dependencies across the project and dependent libraries, existing methods can identify module-level dependencies but drop symbol-level ones when library source code is unavailable.

To overcome those limitations, we propose PyAnalyzer, an effective and practical approach for Dependency Extraction from Python Code. Similar to prior methods like PyCG, PyAnalyzer also conducts an inter-procedural points-to analysis to reason about dependencies. The difference is that the core of PyAnalyzer is to uniformly model functions, classes, and modules as first-class heap objects to reveal more intricate dynamic features without compromising precision. To address the *issue* ❶ and *issue* ❷, PyAnalyzer conducts modular summarization to model Python objects based on a tree structure which expresses the hierarchical namespace. Compared to PyCG which addresses duck typing and first-class functions, our modular summarization additionally models first-class classes and first-class modules into heap objects since they can also be dynamically created, passed, modified, and returned. Our points-to propagation additionally considers the dynamic changes of both the objects and class inheritance. To address the *issue* ❸ and *issue* ❹, PyAnalyzer utilizes static type annotations (Python Enhancement Proposals, PEP [46]) to resolve library dependency on demand and express the same dependent entity in different call sites. Existing techniques require analyzing library source code, which comes at a heavy cost due to a large number of released modules [62]. PyAnalyzer processes type annotation files of libraries that only declare type information without the need for the whole source code.

Our experiments collected four comprehensive benchmarks and 10MSLOC real-world projects. Based on this dataset, we compared PyAnalyzer with 7 SOTA baselines, including one proprietary tool (Understand [63]), three open-source tools supporting multiple languages (Sourcetrail [57], Depends [38], ENRE19 [22]), two techniques dedicated to Python (PySonar2 [64], PyCG [53]), as well as a deep learning model (Type4Py [35]). Experiments show that 1) on a micro-benchmark covering diverse deterministic dependencies, PyAnalyzer improves F_1 score by 9.8% $\sim$ 48.9% than baselines; 2) on a micro-benchmark of nondeterministic dependencies, F_1 score of PyAnalyzer is 8.7% $\sim$ 35.1% higher than baselines; 3) using two sets of macro-benchmarks collected from real-world projects, PyAnalyzer achieves the highest F_1 score, surpassing baselines by 24.3% $\sim$ 32.7% in recall results; and 4) averaged on 191 projects with 10MSLOC, PyAnalyzer performs 1.4x $\sim$ 5.6x faster than baselines in analyses of a medium-scale project ($SLOC \leq 10K$). When analyzing a large project ($10KSLOC \sim 750KSLOC$), PyAnalyzer spends less time than baselines except for Understand.

In summary, this work makes the contributions as follows.

- We propose PyAnalyzer to resolve symbol-level dependencies for Python code via enhanced points-to relation propagation and the use of type annotation features. The method tackles the four challenges attributed to object changes, first-class citizens, varying call sites, and libraries.
- Our experiments on four sets of benchmarks and 10MSLOC real-world projects demonstrate the effectiveness and efficiency of PyAnalyzer when compared to 7 baseline techniques.
- We contribute a dataset[1] including benchmarks and analysis results on real-world projects. To the best of our knowledge, the size of our benchmarks significantly surpasses that of benchmarks used by related work.

2 MOTIVATION

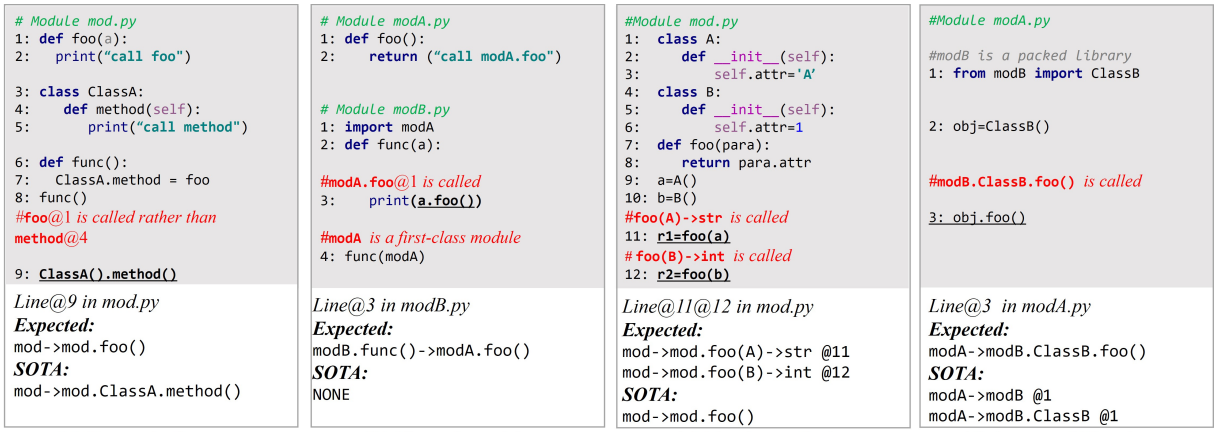
Figure 1 illustrates the four challenges faced by prior methods through examples.

Issue ❶:**Insufficient coverage of dynamic features.** Python presents diverse dynamic features like duck typing and object change [16]. The duck typing feature determines the type of an object by checking the presence of given attributes and behaviors. The object change allows updating, adding, and removing attributes of an object arbitrarily. Consequently, runtime behaviors of objects would differ from those statically declared ones, making dependency extraction via static analysis challenging.

Figure 1(a) shows a code snippet that dynamically changes the attributes of class objects. In Line@4, `ClassA` defines an attribute named `method`, which is then re-assigned with the function `foo` inside `func` in Line@7. Thus, the behavior of `ClassA.method` would be changed if `func` is invoked at runtime: `ClassA().method()` (Line@9) will call `foo()` (Line@1) rather than the original `method` (Line@4). Therefore, one dependency from `mod` to `mod.foo` is expected from Line@9. However, tools like PySonar2 and PyCG wrongly reported `mod` $\rightarrow$ `mod.ClassA.method()`.

Issue ❷:**Limited modeling of first-class citizens.** Functions, classes, and modules in Python are first-class objects that can be

[1]All datasets are available on https://github.com/xjtu-enre/ICSE2024_PyAnalyzer



(a) Dynamic features (b) First-class citizens (c) Varying call sites (d) Imported libraries

Figure 1: Motivation examples

passed as function parameters, returned as values, and modified as required. This flexibility makes it difficult to identify the heap creation of high-order objects, hindering the dependency accuracy. Although first-class functions have been considered by PyCG [53], first-class classes and modules receive little attention.

Figure 1(b) utilizes modules as first-class citizens. The module `modA` defines a method `foo`; module `modB` imports `modA` and defines `func`; `modB` invokes `func` in Line@4, passing `modA` as a parameter. Therefore, the dependency from `modB.func()` to `modA.foo()` is expected at `print(a.foo())`. Nevertheless, PyCG missed this result. The parsing of first-class classes also faces similar issues.

Issue ❸:**A lack of distinguishing between varying call sites.** A method is often invoked in different calling contexts. Due to the duck typing in Python, the type of an object that is received or returned by a function depends on context. The signature of a callee should be customized in varying call sites to reflect this difference, while existing tools failed to handle it.

In Figure 1(c), `foo()` is called and accepts an instance of class `A` in Line@11; thus the callee `foo` signature will be `foo(para:A)->str`. Similarly, `foo` signature will be `foo(para:B)->int` in Line@12. Therefore, two dependencies are expected: the one from module `mod` to `foo(para: A)->str` in Line@11; and the one from `mod` to `foo(para:B)->int` in Line@12. That is, the same callee `foo` is expressed with different type annotations. However, prior methods ignored distinguishing this point.

Issue ❹:**Inability of revealing dependencies about libraries.** Modern code heavily relies on built-in or third-party libraries. The dependency resolution between the project code and libraries is a well-recognized difficulty–"You enter the library, you stay in the library" [62]. Due to the unavailable source code of the imported libraries, existing methods always identify dependencies at the module level, dropping symbol-level information.

In Figure 1(d), module `modA` imports `modB` that is a library without source code. Line@3 indicates `modA->modB.ClassB.foo`. However, given that the code of `modB` is unavailable, Methods like PyCG only generate `modA->modB` or `modA->modB.ClassB` at a course-grained level. It is also impossible to scan the whole libraries from the PyPI [14] to obtain symbol-level dependencies in a reasonable amount of time due to millions of released packages. A *shortcut* is required to resolve such dependencies at fine-grained levels.

Summary. It is challenging for static code analysis to resolve symbol-level dependencies due to the nontrivial and dynamic features in dynamic languages, especially object changes, first-class citizens, varying call sites, and libraries. Such dependencies account for a substantial amount, imposing a significant impact on a wide range of software analysis [21, 24]. This motivates us to solve the fundamental difficulty for Python.

3 METHOD

We propose PyAnalyzer to resolve entities and dependencies from Python code based on inter-procedural points-to analysis. PyAnalyzer makes four improvements to overcome the limitations aforementioned in Section 2.

The general idea of PyAnalyzer is that it uniformly models functions, classes, and modules as first-class objects to simulate the Python feature of "everything is an object". First, unlike PyCG which maintains an assignment graph between identifiers, PyAnalyzer creates an intermediate representation (i.e., a tree structure) that expresses the hierarchical namespace and non-deterministic symbol lookup order. Second, PyAnalyzer conducts modular summarization based on the tree structure to abstract Python objects. Besides modeling duck typing and first-class functions well-addressed by prior methods, our modular summarization additionally emphasizes first-class classes/modules. Third, PyAnalyzer extends points-to propagation based on modular summarization. Our propagation additionally considers the dynamic creation of first-class functions/classes/modules and runtime changes of class inheritance. Fourth, instead of processing heavy-weight source code of libraries, PyAnalyzer leverages the feature of Python type annotations as *shortcut* to resolve library dependencies.

3.1 Method Overview

We follow the work of [42, 53] to define the syntax of a simple imperative and object-oriented language, where PyAnalyzer works.

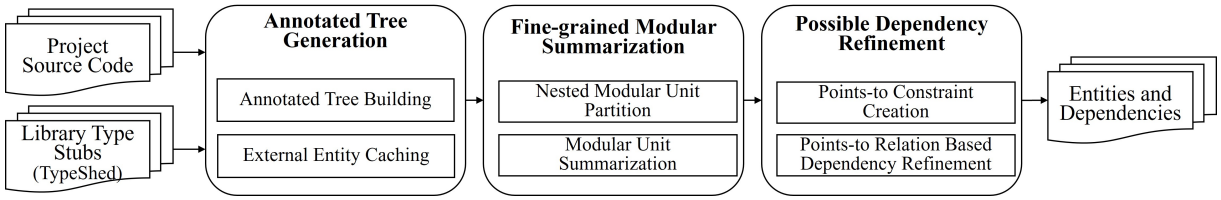


Figure 2: Method overview of PyAnalyzer

In the syntax representation as follows, e and s denote an expression and a statement; $x = e$ is an assignment statement; `class` $e(e...) : e$ defines a class; `function` $e(e...) : e$ defines a function; $e(e, ..., e)$ denotes a function call; `import` x `from` m `as` y imports a module; `lambda` $x : e$ expresses an anonymous function; $[e, ..., e]$ and $\{e : e, ..., e : e\}$ correspond to a list and dictionary; e op e denotes *boolean*, *numeric*, *bitwise*, and *compare* operations.

$$
\begin{aligned}
e \in Expr ::= & x \mid c \mid x : e \mid e \text{ op } e \mid e[e : e : e] \mid e(e, ..., e) \\
& \mid e.x \mid [e, ..., e] \mid \{e, ..., e\} \mid \{e : e, ..., e : e\} \\
& \mid [e \texttt{ for } e \texttt{ in } e] \mid \{e \texttt{ for } e \texttt{ in } e\} \\
& \mid \{e : e \texttt{ for } e,\ e \texttt{ in } e\} \mid (e \texttt{ for } e \texttt{ in } e) \\
& \mid \texttt{if } e : e \mid \texttt{elif } e : e \mid \texttt{else } : e \\
s \in Stmt ::= & \texttt{function } e(e...) : \ e \mid \texttt{class } e(e...) : e \mid \texttt{return } e \\
& \mid \texttt{import } x \texttt{ from } m \texttt{ as } y \mid \texttt{lambda } x : e \\
& \mid \texttt{new } x(y = e...) \mid \texttt{iter } x \\
x,\ y \in Identifier ::= & \ is\ the\ set\ of\ identifiers \\
c \in Constants ::= & \ is\ the\ set\ of\ literals \\
m \in Module ::= & \ is\ the\ set\ of\ modules
\end{aligned} \tag{1}
$$

We illustrate concepts and symbols for method formulation. An **entity** is denoted as e and has several properties. *e.name* labels the qualified name of e; *e.startline*,*e.endline*, *e.startcolumn*, and *e.endcolumn* denote the location where e is defined; *e.type* denotes the entity kind like *class*, *method*, and *variable*. A **dependency** is $e_i \xrightarrow{d} e_j$, where e_i is the source entity; e_j is the target entity depended on by e_i; d denotes the dependency kind such as *import*, *inherit* and *call*. Dependencies are categorized into **deterministic dependencies** (i.e., explicit dependencies) that are explicitly manifested in the source code and **nondeterministic dependencies** (i.e., possible dependencies) that are invisible due to dynamic features [21, 24]. The targets of possible dependencies may be bound into more than one entity, and such dependencies can, in general, only be determined at runtime.

Figure 2 depicts the overview of PyAnalyzer. The input includes the *Project Source Code* that will be analyzed, and *Library Type Stubs* like the Typeshed project that stores type annotations for Python libraries and packages. Typeshed [47], officially maintained by Python, provides type stubs for built-in, standard, and third-party libraries. PyAnalyzer finally outputs *Entities and Dependencies* which are expected from the analyzed project.

3.2 Annotated Tree Generation

PyAnalyzer translates the target source code into a set of annotated trees. This representation expresses the hierarchical namespace and non-deterministic symbol lookup order, connecting entities with deterministic dependencies. In general, this module scans *Project Source code* and *Library Type Stubs* to generate entities E_b, deterministic dependencies D_b, unresolved non-deterministic dependencies D_{im}, and annotated trees T_a. It also caches external entities E_{ex} that are imported by the project from libraries.

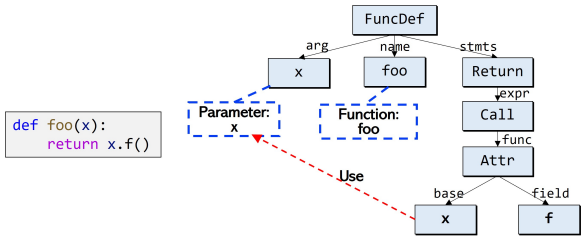
3.2.1 Annotated Tree Initialization. This part indexes code facts from the source code to produce annotated trees. Similar to the work by [22, 24], it first employs Python built-in `ast` module [48] to create ASTs T from the analyzed project. By traversing ASTs, we record basic entities E_b and dependencies D_b based on symbol tables. The dependencies directly obtained from AST are *deterministic dependencies* and they are only a subset of the whole dependencies that would be expected [21]; the unresolved ones (D_{im}) are *nondeterministic dependencies* related to implicit code behaviors, which will be the focus of PyAnalyzer.

It then creates a set of annotated trees, i.e., T_a based on T, E_b, and D_b. An annotated tree is an AST that is augmented with extra nodes and dependencies indicated by $e \in E_b$ and $d \in D_b$. Figure 3 illustrates an annotated tree. In Figure 3(b), the dotted blue rectangles denote two entities resolved from the AST nodes, (i.e., `Parameter x` and `Function foo`), and the dotted red line is added due to an *use* dependency in D_b, i.e., `x.f` $\xrightarrow{use}$ `Parameter:x`.

3.2.2 External Entity Caching. Existing techniques like PyCG and Understand require the availability of source code of libraries to handle symbol-level dependencies between projects and libraries. It comes with a heavy cost due to a large number of library modules [62]. Python 3.5 introduces type annotation (see PEP 484), which allows only declaring type information in separate stub files (i.e., .pyi files). Our PyAnalyzer leverages this type annotation feature to resolve libraries in *shortcut*, as well as deals with dependency cycles introduced by this feature.

Given that *Typeshed* [47] provides type stubs (`*.pyi` files) for Python libraries and packages, we collect external entities (E_{ex}) by retrieving modules by names or finding AST nodes that define particular names in stubs. Since a stub usually contains a large number of entities, PyAnalyzer only caches a portion on demand: it stores the external entities that are imported and reused by the analyzed project, dropping other entities in stub files.

Dependency cycles in type stub files make it complex to resolve E_{ex}. For example, `builtins.pyi` and `typing.pyi` constructs a cyclic dependency: `builtins.pyi` is the stub file of built-in module; `typing.pyi` corresponds to the Python module which defines types; `builtins.pyi` imports type aliases defined by `typing.pyi`



(a) Code snippet (b) An annotated tree with basic dependencies

Figure 3: An annotated tree produced by PyAnalyzer

like `Dict` and `Optional`; and `typing.pyi` imports the entities defined in `builtins.pyi`. PyAnalyzer adopts an iterative procedure to resolve cycles. For each cycle, PyAnalyzer initially scans one module m_1 by ignoring its `import` statements. It labels the entities that are imported by m_1 with `UNKNOWN`. Other stub modules m_i in this cycle are continuously processed. In the next iteration, it resolves the `UNKNOWN` entities in m_i by analyzing `import` statements and searching other scanned modules m_j. Once one `UNKNOWN` entity is resolved, its tag will be cleared. After several iterations, `UNKNOWN` entities are resolved. Finally, the external entities and cross-module dependencies are cached on demand, further updating the pre-built annotated trees namely T_a.

3.3 Fine-Grained Modular Summarization

Based on the intermediate representation (i.e., T_a), PyAnalyzer generates a set of modular summaries (S). A modular summary describes each modular unit (e.g., *module*, *class*, and *function*) as a sequence of abstract operations. Different from existing work [53], PyAnalyzer uniformly models functions, classes, and modules into first-class objects and customizes their modular summarization. This manner helps cover diverse kinds of Python advanced features, preserving more code behavior than prior work.

3.3.1 Nested Modular Unit Partition. According to the Python LEGB scope rules [50], PyAnalyzer naturally divides T_a into a nested structure, where each element represents a modular unit (i.e., $mu \in MU$) that will be analyzed. PyAnalyzer divides the analyzed code into four types of modular units, namely *package*, *module*, *class*, and *function*. A modular unit of *package* embeds a sequence of *module*, a *module* embeds a sequence of *class* and *function* units, and a *class* (or *function*) can embed *class* and *function* units.

3.3.2 Modular Unit Summarization. This step analyzes each modular unit (*mu*) and outputs a sequence of abstract operations, i.e., *Summary(mu)*=$[op_1, op_2, ..., op_n]$. The first 4 summary generation rules express how PyAnalyzer dynamically creates functions, classes, and modules as first-class objects.

Abstract Operations. Abstract operations model Python code behaviors. Each statement can be expressed in a sequence of operations. *Move*, *Load*, *Store*, *Return*, *Invoke*, and *Assign* are basic operations commonly covered by existing work [53, 55, 62]. We used *CreateClass*, *CreateFunction*, *CreateModule*, *AddBase*, *InstCreation*, *ListCreation*, *DictCreation*, *ReadIndex*, and *WriteIndex* to express code syntax related to the function, class, module, class inheritance, data structures, and field operations.

$$Move(x, y)\ IF\ x = y;\ Load(x, y, f)\ IF\ x = y.f;$$
$$Store(x, f, y)\ IF\ x.f = y;\ Invoke(t, f, a1, ...)\ IF\ t = f(a1, ...);$$
$$Return(x)\ IF\ return\ x;\ CreateClass(c, cls)\ IF\ Class\ cls(b, ...) : ...;$$
$$CreateFunction(f, func)\ IF\ def\ f(x_1, ..., x_n) : ...;$$
$$CreateModule(m, mod)\ IF\ a\ new\ file;\ Assign(x, y)\ IF\ x = y;$$
$$AddBase(c, b1, ...)\ IF\ Class\ c\ (b1, ...) : ...;\ InstCreation(x)\ IF\ x = cls(...);$$
$$ListCreation(x)\ IF\ x = list(), x = [y\ for\ y\ in\ lst];$$
$$DictCreation(x)\ IF\ x = dict(), x = \{y : dct[y]\ for\ y\ in\ dct\};$$
$$ReadIndex(x, y)\ IF\ x = y[*],\ for\ x\ in\ y;\ WriteIndex(x, y)\ IF\ x[*] = y;$$

These operations exclude arithmetic semantics like unary and binary operations due to their irrelevance in dependency resolution.

Summary Generation Rules. The summary generation rules (①-⑪) for diverse syntax statements are as follows. Prior work [53] defined rules (⑤-⑪) for object-oriented programming languages to generate summaries, covering function calls, assignment, return, property access, and property assignment. PyAnalyzer adds ①-④ for the creation of classes, functions, and modules. The extension helps model advanced features like first-class classes/modules, object change, and duck typing.

Generation Rules:

$$① e_0(e_1, ..., e_n) \Longrightarrow \frac{r{=}NewName(),\ Summary(e_i,t_i)}{Invoke(r,t_0,t_1,...,t_n),\ Summary(e_0(e_1,...,e_n),r)}$$
$$②class\ cls(e_1, ..., e_n) \Longrightarrow \frac{Summary(e_i,t_i)}{CreateClass(cls),\ AddBase(cls,t_1,...,t_n)}$$
$$③def\ f(x_1, ..., x_n) \Longrightarrow CreateFunction(f, x_1, ..., x_n)$$
$$④x \Longrightarrow \frac{Use(x,v)}{Summary(x,v)}\quad ⑤x = e \Longrightarrow \frac{Summary(e,t)}{Assign(x,t)}$$
$$⑥return\ e \Longrightarrow \frac{Summary(e,t)}{Return(t)}$$
$$⑦\ e.f \Longrightarrow \frac{r{=}newName(),\ Summary(e,t)}{Load(r,t_2,f),\ Summary(e.f,r)}$$
$$⑧\ e_1.f = e_2 \Longrightarrow \frac{Summary(e_i,t_i)}{Store(t_1,f,t_2)}$$
$$⑨\ e_1[e_2] \Longrightarrow \frac{r{=}NewName(),\ Summary(e_i,t_i)}{ReadIndex(r,t_1),\ Summary(e_1[e_2],r)}$$
$$⑩\ e_1[e_2] = e_3 \Longrightarrow \frac{Summary(e_i,t_i)}{WriteIndex(t_1,t_3)}$$
$$⑪\ a\ new\ mod \Longrightarrow CreateModule(mod)$$

where $Summary(e, t)$ indicates that the evaluation result of variable t is equal to that of an expression e; t is a variable defined in statements or an intermediate variable; $Use(x, v)$ is a *use* dependency obtained in Section 3.2; $Use(x, v)$ indicates that the reachable definition of a name expression x is variable v; *NewName()* creates a unique intermediate name.

*Rule*① corresponds to the function call expression $e_0(e_1, ..., e_n)$. It first generates a summary for each sub-expression in e_i, storing evaluation results into t_i, i.e., $Summary(e_i, t_i)$. It then creates a new variable r and adds an invocation behavior $Invoke(r, t_0, t_1, ..., t_n)$, producing the summary result for the analyzed modular unit, i.e., $Summary(e_0(e_1, ..., e_n), r)$.

*Rule*② corresponds to the class declaration like *class* $cls(e_1, ..., e_n)$. Unlike static languages, Python treats *class* as a first-class citizen. This rule creates an object of the class for class declaration syntax, i.e., $CreateClass(cls)$, and then $AddBase(cls, t_1, ..., t_n)$ if the parent object is specified. $CreateClass(cls)$ indicates a class object creation named *cls* at runtime, and $AddBase(cls, t_1, ..., t_n)$ indicates the base class of *cls* including evaluation results of $e_1, ..., e_n$.

*Rule*③ corresponds to function definition $def\ f(x_1, ..., x_n)$. Similar to the classes in Rule②, functions are also first-class citizens in Python. $CreateFunction(f, x_1, ..., x_n)$ indicates that the analyzed unit will create a function object f with a parameter list of $x_1, ..., x_n$.

*Rule*④ corresponds to x, a name expression syntax. It finds all possible reachable definitions v for x through *use* dependencies obtained in Section 3.2. Then it adds $Summary(x, v)$ to the summary results, meaning that the evaluation result of the expression x would be all reachable definitions named v. Different from existing methods, this rule employs basic dependency results (D_b) to characterize non-deterministic name lookups for Python.

The remaining *Rule*⑤-⑪ generate modular summary results for *assignment*, *return*, *index*, *access*, and *modify* syntax, which are similar to existing work [53].

For example, Figure 3(b) shows a modular unit named the function `foo`. Its modular summary consists of a sequence of abstract operations, `[Load(_`t_0, x, f`), Invoke(_`t_1`, _`t_0`), Return(_`t_1`)]`.

3.4 Possible Dependency Refinement

Similar to PyCG [53], PyAnalyzer also conducts points-to analysis to create candidate relations, which are further used to refine ambiguous dependencies. Points-to analysis computes abstractions of the possible values of pointer variables, i.e., variables, functions, lists, class instances, etc[26]. The difference is that PyAnalyzer extends existing points-to constraints based on the modular summaries we have improved (S). More importantly, our extension mainly considers dynamic object creation of first-class functions/classes/modules and run-time changes of class inheritance relationship.

3.4.1 Points-to Constraint Creation. The following formulation lists the constraints for points-to analysis. *Rules*①-⑤ correspond to heap objects, handling the high-level language characteristics. *Rule*⑥-⑫ are the general rules to propagate points-to relations [32, 53, 60]. We will explain the first five rules that contribute to the analysis of Python's advanced features.

Points-to Constraints:

$$\textcircled{1}\ i : Invoke(r, f, x_1, ..., x_n) \Longrightarrow \frac{o_i \in pt(f),\ ClassObject(o_i, cls),\ o_j = Heap(cls, i)}{o_j \in pt(r)}$$

$$\textcircled{2}\ CreateClass(c, cls) \Longrightarrow \frac{o_i = Heap(cls)}{ClassObject(o_i, cls),\ o_i \in pt(c)}$$

$$\textcircled{3}\ CreateFunction(f, func) \Longrightarrow \frac{o_i = Heap(func)}{FunctionObject(o_i, f), o_i \in pt(f)}$$

$$\textcircled{4}\ CreateModule(m, mod) \Longrightarrow \frac{o_i = Heap(i)}{ModuleObject(o_i, mod),\ o_i \in pt(m)}$$

$$\textcircled{5}\ AddBase(cls, x_1, ..., x_n) \Longrightarrow \frac{o_u \in pt(x_j),\ ClassObject(o_u, cls_j),\ 1 \le j \le n}{Inherit(cls_j, cls)}$$

$$\textcircled{6}\ Assign(x, y) \Longrightarrow \frac{o_i \in pt(y)}{o_i \in pt(x)}$$

$$\textcircled{7}\ Load(x, y, f) \Longrightarrow \frac{o_i \in pt(x),\ o_j \in pt(o_i, f)}{o_j \in pt(y)}$$

$$\textcircled{8}\ Store(x, f, y) \Longrightarrow \frac{o_i \in pt(x), o_j \in pt(y)}{o_j \in pt(o_i, f)}$$

$$\textcircled{9}\ ReadIndex(x, y) \Longrightarrow \frac{o_i \in pt(x), o_j \in pt_{index}(o_i)}{o_j \in pt(y)}$$

$$\textcircled{10}\ WriteIndex(x, y) \Longrightarrow \frac{o_i \in pt(x), o_j \in pt(y)}{o_j \in pt_{index}(o_i, f)}$$

$$\textcircled{11}\ i : Invoke(r, f, x_1, ..., x_n) \Longrightarrow \frac{o_i \in pt(f),\ Function(o_i, m), o_u \in pt(x_j), 1 \le j \le n, o_v \in pt(m_{ret})}{o_u \in pt(m_{pj}), o_j \in pt(r)}$$

$$\textcircled{12}\ Return(x) \Longrightarrow \frac{o_i \in pt(x)}{o_i \in pt(m_{ret})}$$

A call in Python may imply both the creation of heap objects and the general method invocation behavior. Therefore, our method identifies the heap object creation point in function call sites. Concretely, in Rule①: if the collection of objects pointed to by the call target f contains a class object o_i (i.e., $o_i \in pt(f)$), this function call will be recognized as a heap object creation point. Then an object o_j identified by the class object and the call point will be created; o_j is added to the pointer collection that accepts the return value of the function.

Rule② enables the points-to analysis to handle first-class classes. It creates an object o_i identified by a class creation statement ($o_i = Heap(func)$), treating o_i as an object associated with the class creation statement cls, i.e., $ClassObject(o_i, cls)$. It then adds o_i to the pointer collection that accepts the class object, i.e., $o_i \in pt(c)$.

Similar to Rule②, Rule③ models a function as a first-class citizen, which can be passed as a parameter, a return value, etc. This rule allows a better simulation of the polymorphism of functions in dynamic languages. It first creates an object o_i identified by a function creation statement, i.e., $o_i = Heap(func)$. It then identifies o_i as a function object associated with f through $FunctionObjec(o_i, f)$. At last, it adds o_i to the pointer collection that accepts this function object via $o_i \in pt(f)$.

Rule ④ handles the module as a first-class citizen, similar to first-class classes and functions. The explanation is similar to the Rule② and ③. Briefly, it first creates an object via $o_i = Heap(mod)$, then identifies o_i as a module object, i.e., $ModuleObject(o_i, mod)$. Finally, it updates $o_i \in pt(m)$.

The class hierarchy in static languages is determined during compilation while it is built at runtime in dynamic languages. Rule⑤ tracks the inheritance system of objects and changes at runtime. If a class object o_u inside $pt(x_j)$ is associated with cls_j, cls_j would be modeled as a base class of cls and further added to the inheritance system for cls, forming a new class inheritance between cls and cls_j (i.e., $Inherit(cls_j, cls)$). It better simulates the dynamic inheritance system in dynamic languages.

Rule⑥-⑫ present how the points-to relations pt propagate through $Assign$, $Load$, $Store$, $ReadIndex$, $WriteIndex$, $Invoke$, and $Return$ operations. The points-to propagation is similar to that in static languages [29]. A set of points-to relationships named $\{pt\}$ is initialized based on the modular summarization results. $\{pt\}$ will be iteratively updated through the points-to constraints until any pt keeps unchanged, reaching a fixed point [12]. Up to now, a final $PT = \{pt\}$ that satisfies the points-to constraints is produced, indicating the objects to which a name x would point.

3.4.2 Points-to Relation Based Dependency Refinement. PyAnalyzer employes the points-to relation results (PT) to refine the unresolved dependencies D_{im} in Section 3.2. For $e_i \rightarrow e_j \in D_{im}$, PyAnalyzer updates the reference of e_j into the object set that e_j points to.

We use Figure 4 to illustrate this step. In Figure 4(a), O_i denotes the heap object created at a program point i. At STEP1, the *Rule*② indicates three object creations, i.e., O_1, O_3, and O_6. The points-to relations pt of cls_x, cls_y, $mixed\_cls$, and obj are initialized as empty. When handling the modular summarization $Invoke(mixin, A, B)$ in Line 8 at STEP2, $pt(cls\_x)$ and $pt(cls\_y)$ are updated to point to O_1 and O_2, due to function invocation of `mixin()` (see *Rule*⑪). At STEP3, two *Inherit()* operations are created inside `mixin()`, i.e., $Inherit(Mixed, A)$ and $Inherit(Mixed, B)$; it also updates $pt(mixed\_cls) = O_6$ after `mixin()` returns. Since $pt(mixed\_cls)$ points to O_6 which is a class creation point of `Mixed`, STEP4 will create an instance obj of Class `Mixed`, pointing to O_9. The final points-to relations PT contains $pt(cls\_x) = O_1, pt(cls\_y) =$

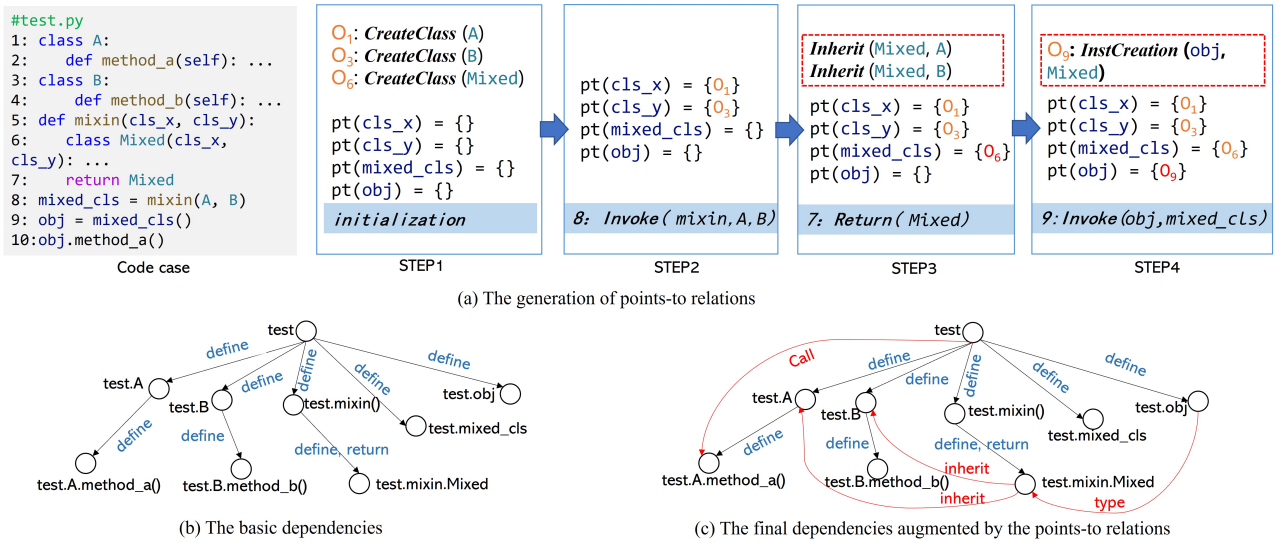


Figure 4: An example to illustrate the *Possible Dependency Refinement* of PyAnalyzer

Table 1: Baseline tools and methods

Study	Baselines
RQ1	Understand, Sourcetrail, Depends, ENRE19 , PySonar2
RQ2	PyCG, Type4Py
RQ3	Understand (from RQ1), PyCG (from RQ2)
RQ4	All baselines except Type4Py

O_3, $pt(mixed\_cls) = O_6$, and $pt(obj) = O_9$; two inherit operations, i.e., $Inherit(Mixed, A)$ and $Inherit(Mixed, B)$, are created.

Figure 4(b) and (c) correspond to the dependencies before and after using *PT* results. Figure 4(b) depicts deterministic dependencies D_b: each node denotes an entity labeled with a qualified name; each edge shows a dependency between entities like *define* and *return*. Augmented with *PT* in Figure 4(a), we further resolve four nondeterministic dependencies, including (1) *inherit*: `test.mixin.Mixed` → `test.B`, (2) *inherit*: `test.mixin.Mixed` → `test.A`, (3) *type*: `test.obj` → `test.mixin.Mixed`, and (4) *call*: `test` → `test.A.method_a`. The four dependencies are rendered by red edges in Figure 4(c), which cannot be identified by PyCG.

4 EVALUATION SETUP

We investigate the four research questions.

RQ1. What is the accuracy of PyAnalyzer to identify deterministic dependencies? The answer shows the method's effectiveness in handling common language features.

RQ2. What is the accuracy of PyAnalyzer to identify nondeterministic dependencies? This study shows the method's effectiveness in handling the four challenges as aforementioned.

RQ3. What is the general accuracy of PyAnalyzer to analyze real-world projects? This RQ observes the effectiveness of PyAnalyzer in practice.

RQ4. What is the time and memory consumption of PyAnalyzer to analyze real-world projects? This RQ tests the efficiency of PyAnalyzer in practice.

4.1 Baseline Methods and Tools

To evaluate the effectiveness of our PyAnalyzer, we collected seven state-of-the-art tools as baselines for comparison (see Table 1). The baselines include one commercial tool named SciTools Understand [63], four open-source community tools that are Sourcetrail [57], Depends [38], ENRE19 [22] and PySonar2 [64], two advanced methods that are PyCG [53] and Type4py [35]. Understand has been extensively employed in both industry and academia [17, 36, 37]. Sourcetrail dumps dependencies into SQLite3. ENRE (labeled as ENRE19), is our early work [22] to analyze dependencies based on Antlr parser generator [41], and it has been used for type annotation studies[25] and architecture analysis [21, 23, 24]. Depends has been considered as baselines for research [53]. PySonar2 [64] is dedicated to analyzing Python projects. Google and SourceGraph use an early version of PySonar2 to index code for code search. Similar to the work [7], we selected the advanced version PySonar2. PyCG generates call dependencies, outperforming Pyan [61] and Code2graph [59]. Type4Py [35] is a deep learning-based model to infer types for Python functions. It performs better than Typilus [3] and TypeWriter [45]. Call dependencies can be dumped from the source code retrofitted with the inferred type annotations.

As listed in Table 1, RQ1 uses Understand, Sourcetrail, Depends, ENRE19, and PySonar2 as baseline techniques against the benchmark of deterministic dependencies. The reason is that all these tools support analyzing *deterministic* dependencies with multiple kinds like *import*, *call*, and *set*. RQ1 excludes PyCG and Type4Py since they can only generate function *call* dependencies. RQ2 uses PyCG and Type4Py against the benchmark of nondeterministic dependencies since only the two tools support inferring nondeterministic dependencies, while the others fail to handle them. RQ3 compares PyAnalyzer with the two best baselines, i.e., Understand and PyCG (see RQ1 and RQ2 results in Section 5) for a general

Table 2: *Micro-BenchmarkA* for RQ1 study

Entity		Dependency		Total	
#Test	#Item	#Test	#Item	#Test	#Item
18	62	27	160	45	222

Entities cover *module*, *variable*, *function*, *class*, etc. Dependencies cover *define*, *use*, *import*, *call*, *inherit*, etc. #Item is the number of entities or dependencies.

Table 3: *Micro-BenchmarkB* for RQ2 study

Category	Description	#Test
args_call	call the parameter objects	14
assignments	call the assigned objects	15
builtins	call built-in objects	10
classes	call first-class classes and members	40
dicts	call the object aggregated in a dict	22
direct_calls	call the objects returned by functions	10
dynamic	call dynamic code through eval function	2
functions	function calls	4
generators	call objects produced by generators	13
high_order_class	call and return first-class classes	8 (0+8)
imports	call objects imported in complex cases	14
kwargs	call with keyward parameter passing	10
lambdas	call anonymous functions	14
lists	call the object aggregated in a list	22 (16+6)
mro	call functions in multiple inheritance	13 (10+3)
returns	return functions in complicated cases	12
unsure_lookup	look up unsure names in branches/loops	10 (0+10)
SUMMARY		**233 (206+27)**

accuracy evaluation against the benchmark collected from real-world projects. RQ4 tests the efficiency of all baseline tools except a pre-trained model namely Type4Py.

4.2 Collection of Four Benchmarks

4.2.1 Micro-BenchmarkA for RQ1. We collected deterministic dependencies along with entities for method verification. The collection process is as follows. We first referred to Python language specification [15] and divided it into excerpts, each one focusing on an individual feature. We then manually programmed code snippets to exploit language feature based on the specification excerpt. Finally, for every code snippet, we appended an assertion block to record ground-truth entities and dependencies. As listed in Table 2, the collected *Micro-BenchmarkA* contains 45 small programs (i.e., tests), covering 62 entities and 160 dependencies.

4.2.2 Micro-BenchmarkB for RQ2. We reused the benchmark collected by the work of PyCG [53] which provides nondeterministic dependencies at function levels. The original benchmark consists of a set of unique tests, each including source code, the corresponding *call* dependencies, and a short description. We extended this suite into 233 tests to cover more implicit code behaviors. Table 3 summarizes this *Micro-BenchmarkB*. This micro-benchmark covers 233 small programs (i.e., tests), where 206 tests were introduced from the work of [53] and 27 tests were supplemented by our work.

4.2.3 Macro-BenchmarkC and Macro-BenchmarkD for RQ3. We collected two macro-benchmarks from real-world projects. *Macro-BenchmarkC* reuses the macro-benchmarks provided by PyCG, containing function-level dependencies from 5 real-world projects.

Table 4: Two sets of macro-benchmarks for RQ3 study

BenchmarkC	#SLOC	BenchmarkD	#SLOC
5 projects	10,031	54 projects	245,167

Table 5: Project Collection for RQ4 study (SLOC distribution)

Projects	SUM	MIN	25th	50th	75th	MAX
191	10,616,422	19	3,334	11,736	51,233	746,786

The work of PyCG manually created this benchmark. To construct a larger size of benchmarks, we automatically built *Macro-BenchmarkD from execution traces*. Existing work by [21, 24, 58] also employed the dependencies from software executions as the ground truth. First, we used the 191 projects collected in Section 4.3. We then selected $191 \times 38.7\% = 74$ projects since PyCG (our baseline) failed to scan the others and reported errors like `MEMORY_ERROR` and `TIME_OUT`. Next, driven by test cases, we utilized MonkeyType [19] to collect runtime information from execution traces, retrofitting run-time types to the corresponding code. After that, we employed Pyre [34], maintained by Facebook, to directly export ground-truth call dependencies from type-decorated code. We further removed the outlier dependencies (introduced by Pyre) where their source entities are beyond the project code. To reduce possible bias, we filtered out 20 projects where the collected ground-truth dependencies were less than 10 items, keeping the other 54 projects as benchmarks.

Table 4 summarizes collection results. The two macro-benchmarks consist of dependencies from 59 real-world projects with 255KSLOC.

4.3 Project Collection

We collected the top-starred projects from GitHub to assess multiple tools' efficiency for RQ4. Concretely, we programmatically accessed the GitHub search API endpoint for Python projects that are sorted based on star count descending. We initially accessed the top 200 projects. We further excluded toy and demo-like projects and tutorials. We finally collected 191 projects for method efficiency testing. Table 5 lists the distribution of project size (SLOC). The 191 projects have 10M SLOC, with diverse sizes of $19 \sim 700K$ SLOC.

5 EVALUATION RESULTS

5.1 RQ1 Results

5.1.1 Effectiveness Measures. We calculated precision (P), recall (R), and F_1 score to verify deterministic dependencies that were produced by different methods, against *Micro-BenchmarkA*.

$$Precision = \frac{|E|}{|E'|},\ Recall = \frac{|E|}{|B|},\ F_1 = \frac{2 * Precision * Recall}{(Precision + Recall)} \quad (2)$$

where $|E|$ counts the entities that are correctly identified within the benchmark, $|E'|$ counts the entities that are identified, and $|B|$ counts all entities in the benchmark. The measures for dependencies are similar to those for entities. One entity e_i is considered correct if its name is equal to that of the corresponding entity (e_b) within the benchmark; one dependency is considered correct if the name of its source entity and the name of its target entity are both correct.

Table 6: Precision, Recall, and F_1 measurements in RQ1

Tool	Entity (%)			Dependency (%)		
	P	*R*	F_1	*P*	*R*	F_1
Understand	98.3	95.2	96.7	79.9	81.9	80.9
Sourcetrail	86.4	61.3	71.7	27.8	27.5	27.7
Depends	**100**	58.1	73.5	41.9	32.5	36.6
ENRE19	90.2	59.7	71.8	39.8	33.1	36.2
PySonar2	**100**	77.4	87.3	67.0	48.1	56.0
PyAnalyzer	96.9	**100**	**98.4**	**99.4**	**98.1**	**98.7**

5.1.2 Results. Table 6 shows that PyAnalyzer achieved the best performance of F_1 score, followed by Understand. Averaged on F_1 scores of entities and dependencies, PyAnalyzer improved the effectiveness by 9.8% than the second best tool, and by 48.9% than the worst one.

We examined false positives and false negatives reported by PyAnalyzer and baseline tools. Depends and PySonar2 presented 100% precision for entities due to their conservative analysis. PyAnalyzer indicated false positives because it produced incorrect *set* dependencies from methods to variables inside *conditional branch* blocks. Sourcetrail performed the worst majorly due to its failure to capture dependency occurrence sites. PyAnalyzer reported false negatives when resolving but not reporting "*b alias a*" in a syntax like "`import a from mod as b`". A major reason for the false negatives of Understand, Sourcetrail, ENRE19, and PySonar2 is their insufficient ability to track advanced syntax features. Moreover, Depends did not identify call dependencies from the module namespace, resulting in low recall. A common observation is that all baselines performed better when retrieving module- and class-level dependencies than those at function and variable levels.

Summary of RQ1. When resolving diverse deterministic dependencies as well as the involved entities, PyAnalyzer outperforms baselines, improving the F_1 score by 9.8% ∼ 48.9%.

5.2 RQ2 Results

5.2.1 Effectiveness Measures. Similar to RQ1, we calculated precision, recall, F_1 score on the *Micro-BenchmarkB*.

5.2.2 Results. The *TOTAL* row in Table 7 indicates a better F_1 performance of PyAnalyzer than PyCG and Type4Py, improving F_1 score by 8.7% and 35.1%. The precision of PyAnalyzer is 92.9%, 4.9% slightly smaller than that of PyCG while both methods achieve 90%+; PyAnalyzer's precision is 10.5% larger than that of Type4Py. Considering recall, PyAnalyzer presents a substantial improvement of 19.7% and 49.7%. Besides, recall of Type4Py suggests that more than half of the expected dependencies are missed by Type4Py.

We inspected the measurements of each category to check false positives and false negatives. The false positives by PyAnalyzer are due to two major reasons: PyAnalyzer ignores modeling the behavior related to *dynamic* category that embeds Python code in `string` value; PyAnalyzer is field-insensitive when handling collection datatypes like *lists*. Most false negatives reported by PyAnalyzer are in the *builtins* category since our library dependency resolution depends on the completeness of type stubs maintained by *typeshed*. PyCG is insufficient to model language semantics of first-class classes, lists, and dicts, leading to a recall value smaller than 50%. PyCG chose conservative analysis, making its precision value in 13/15 categories achieve 100%. Type4Py failed in most of Python's complicated features.

Table 7: Precision, Recall, F_1 measurement results in RQ2

Category	PyCG (%)			Type4Py (%)			PyAnalyzer (%)		
	P	*R*	F_1	*P*	*R*	F_1	*P*	*R*	F_1
args_call	100	100	**100**	100	35.7	30.0	100	100	**100**
assignments	100	86.7	92.9	87.5	93.3	45.2	100	100	**100**
builtins	100	30.0	46.2	85.7	60.0	35.3	75.0	60.0	**66.7**
classes	100	100	**100**	100	15.0	13.0	97.6	100	98.8
dicts	100	22.7	37.0	84.6	100	45.8	91.7	100	**95.7**
direct_calls	100	100	**100**	100	30.0	23.1	100	100	**100**
dynamic	50.0	50.0	**50.0**	0	0	0	0	0	0
functions	100	100	**100**	0	0	0	100	100	**100**
generators	100	100	**100**	65.0	100	39.4	100	92.3	96.0
high_order_class	100	37.5	54.5	0	0	0	100	100	**100**
imports	83.3	71.4	76.9	100	7.1	6.7	100	92.9	**96.3**
kwargs	100	100	**100**	100	30.0	23.1	100	90.0	94.7
lambdas	100	100	**100**	100	35.7	26.3	100	100	**100**
lists	100	45.5	62.5	66.7	36.4	23.5	71.0	100	**83.0**
mro	100	100	**100**	100	41.2	58.3	86.7	100	92.9
returns	100	100	**100**	57.1	33.3	21.1	100	100	**100**
unsure_lookup	100	50.0	66.7	100	30.0	23.1	100	100	**100**
TOTAL	**97.8**	**76.4**	**85.8**	**82.4**	**46.4**	**59.4**	**92.9**	**96.1**	**94.5**

PyAnalyzer performs well in *high_order_class* and *unsure_lookup* cases, with F_1 equal to 100%; on the contrary, PyCG and Type4Py only achieved $\frac{54.5+66.7}{2}\% = 60.6\%$ and $\frac{0+23.1}{2}\% = 11.6\%$ on average.

Summary of RQ2. On the nondeterministic dependency benchmark, F_1 scores of PyAnalyzer are 8.7% and 35.1% bigger than those of PyCG and Type4Py. Since PyAnalyzer models more implicit code behaviors, it substantially improves the recall by 19.7% and 49.7%.

5.3 RQ3 Results

5.3.1 Effectiveness Measures. Reusing the complete *BenchmarkC* provided by the work of PyCG, we computed precision, recall, and F_1 score on this benchmark. Since the *BenchmarkD* was automatically collected from execution traces, its completeness is limited by the coverage of built-in tests. Therefore, similar to the work of [21, 24, 58], we also only computed the recall on *BenchmarkD* for method verification.

5.3.2 Results. Since Understand is the best baseline in RQ1 results (Section 5.1) and PyCG is the best baseline in RQ2 results (Section 5.2), we focus on comparing PyAnalyzer with them. Table 8 lists the precision, recall, F_1 results on *BenchmarkC*. Table 9 illustrates the recall result distribution on *BenchmarkD*.

Table 8 shows that PyAnalyzer performs the best on the average, followed by PyCG. One exception is the *sublist3r* project. The reason is that PyAnalyzer does not reason about *conditional branch* (leading to false positives), which was also observed in RQ1. In summary, PyAnalyzer improves the precision, recall, and F_1 score by 5.6% ∼ 9.9%, 13.8% ∼ 51.6%, 9.6% ∼ 36.3%. The recall values again demonstrate the superiority of PyAnalyzer in modeling more language features.

Table 9 indicates that PyAnalyzer surpasses Understand and PyCG by an improvement of 27.2% and 21.4% on *BenchmarkD*. The median value implies that PyAnalyzer achieves 90%+ recall in half of the studied projects. Moreover, Understand, PyCG, and PyAnalyzer completely captured all ground-truth dependencies (i.e., $recall = 100\%$) in 3.7%, 20.4%, and 20.4% of the studied projects. In a

Table 8: RQ3 results against the Macro-*BenchmarkC*

Project	Understand			PyCG			PyAnalyzer		
	P	*R*	F_1	*P*	*R*	F_1	*P*	*R*	F_1
asciinema	89.8	49.8	64.0	**99.3**	68.1	80.8	95.5	**80.3**	**87.2**
autojump	62.4	47.1	53.7	70.2	77.5	73.7	**76.4**	**82.1**	**79.1**
fabric	47.8	16.3	24.3	67.0	89.4	76.6	**73.5**	**93.6**	**82.3**
face_classification	91.9	43.3	58.9	91.6	46.7	61.8	**98.6**	**99.0**	**98.8**
sublist3r	90.1	55.2	68.4	**100**	**81.9**	**90.0**	86.4	**81.9**	84.1
TOTAL	72.5	37.1	49.1	76.8	74.9	75.8	82.4	**88.7**	**85.4**

Table 9: RQ3 results against the Macro-*BenchmarkD*

R	MIN	25th	50th	75th	MAX	AVERAGE
Understand	0%	21.3%	59.6%	87.4%	100%	**54.2%**
PyCG	0%	22.6%	70.1%	98.9%	100%	**60.0%**
PyAnalyzer	13.9%	69.6%	91.0%	99.2%	100%	**81.4%**

project (*you-get*) that takes the largest number of ground-truth dependencies, PyAnalyzer successfully identified 95.6% dependencies, while Understand and PyCG failed to resolve a portion of 50%.

Summary of RQ3. Measured on the *Macro-BenchmarkC* (collected from 5 real-world projects) provided by PyCG, PyAnalyzer improves the precision, recall, and F_1 score by $\frac{5.6+9.9}{2}\% = 7.8\%$, $\frac{13.8+51.6}{2}\% = 32.7\%$, and $\frac{9.6+36.3}{2}\% = 23.0\%$ than PyCG and Understand. On a larger *Macro-BenchmarkD* automatically collected from 54 real-world projects' executions, the recall of PyAnalyzer surpasses Understand and PyCG by $\frac{27.2+21.4}{2}\% = 24.3\%$.

5.4 RQ4 Results

5.4.1 Efficiency Measures. We assessed the *Completion Time* (sec) and *Peak Memory Usage* (GB) to observe method efficiency. The testing environment is a Windows PC with an i7-12700K CPU and 16GB memory. We employed `time.time()`, a Python built-in API, to record the start and end timestamps for the execution of a method on a project. The range between two timestamps is the time consumption. We employed `psutil` [18] to retrieve the peak memory usage of an execution, similar to prior work [20].

5.4.2 Results. Figure 5 visualizes the measurements with project's SLoC as heat color. The medium size in all 191 projects is 10KSLOC (See Table 5), and we observed the two groups: 89 projects each with $SLOC \leq 10K$ (the left part in Figure 5) and the other 102 projects (the right part in Figure 5). Each boxplot is labeled with the medium value. PyCG only successfully analyzed 38.7% projects, lacking most values due to its reporting runtime errors like `TIMEOUT`.

Figure 5 shows that PyAnalyzer is more time-efficient and memory-efficient when analyzing one project with $SLOC \leq 10K$. PyAnalyzer is 1.4x ∼ 5.6x faster than baseline tools. PyAnalyzer consumes less memory, 3x ∼ 11.7x better than baselines. Analyzing projects with sizes between $10K$ and $750K$, PyAnalyzer spends less time than baselines except for Understand. PyAnalyzer requires a little more memory amount than Understand and Depends.

Summary of RQ4. Testing on 191 projects, PyAnalyzer is 1.4x ∼ 5.6x faster and consumes less memory amount than baselines when handling a project within $10KSLOC$; when analyzing $10KSLOC$+, PyAnalyzer is also time-efficient, spending less time than baselines except for Understand.

6 DISCUSSION

Approach Limitations. First, our PyAnalyzer is insufficient to reason about the *conditional branch*. As reported in RQ1 and RQ3, this deficiency leads to false positives. Second, PyAnalyzer is a field-insensitive approach, reporting false positives related to *container* datatypes (as aforementioned in RQ2). Third, PyAnalyzer ignores modeling the code embedded in strings, producing false negatives (see the *dynamic* category in RQ2). Moreover, PyAnalyzer requires type stubs as inputs to resolve library dependencies. These limitations, on the contrary, can promote a performance balance between effectiveness and efficiency in analyzing real-world projects. As indicated by evaluation results, PyAnalyzer has achieved a substantial improvement in recall without obviously compromising precision and efficiency. Our future work will overcome these limitations by combining static analysis with dynamic analysis.

Threats to Validity. First, comparing PyAnalyzer with different baseline tools might produce inconsistent observations. We chose 7 representative SOTA baselines from comprehensive aspects to mitigate this threat. They include 1) a proprietary tool namely Understand, 2) three open-source multiple-language tools such as ENRE19, Depends, and Sourcetrail, and 3) Python-specific extraction tools, i.e., PySonar2, PyCG, and a deep learning-based Type4Py model. All tools except for Type4Py are static analyses. Understand, Depends, ENRE19, PySonar2, and Sourcetrail have been widely employed in existing work for comparison [37, 53] and downstream tasks [21, 23–25]. Compared to PyCG which employed 2 baselines, our experiments evaluated PyAnalyzer against the collected 7 baselines. Another threat is that the benchmarks for experiments might introduce bias into RQ results. To produce solid and convincing results, we collected 4 comprehensive benchmarks, and two of them reused the benchmark provided by the work of PyCG. We also collected 191 projects with sizes ranging from 19SLOC to 700KSLOC for efficiency testing. To the best of our knowledge, the size of our benchmarks and projects for evaluation is the largest among all those used in related work.

7 RELATED WORK

Dependency Extraction based on Static Analysis and Dynamic Analysis. Many static analysis tools are available. SciTools Understand [63], Sourcetrail [57], and ENRE [21, 22] support analyzing multiple programming languages. PySonar [64] has been integrated by Google to index millions of lines of Python code. Code2Graph [59], Pyan [61], and PyCG [53] generate call graphs for Python. A recent work by Salis et al. [53] showed that PyCG outperforms Pyan and Code2Graph. Dynamic approaches for Python also exist. DynaPyt [11] is a heavy-weight dynamic analysis framework, allowing developers to customize analyses. Pycallgraph [9] traces call stacks to produce call graphs. The dependencies via dynamic techniques are precise while requiring code executions. Our work focuses on static analysis and is most similar to PyCG.

Deep Learning Models for Type Inference. Treating source code as a special type of structured natural language [27], DL models predict types for code. Such type annotations can be employed to generate call dependencies. Many models [33, 49, 51, 65] exist for JavaScript. Typewriter [45], Typilus [3], and Type4Py [35] infer types for Python. The work of [35] indicated that Type4Py surpasses

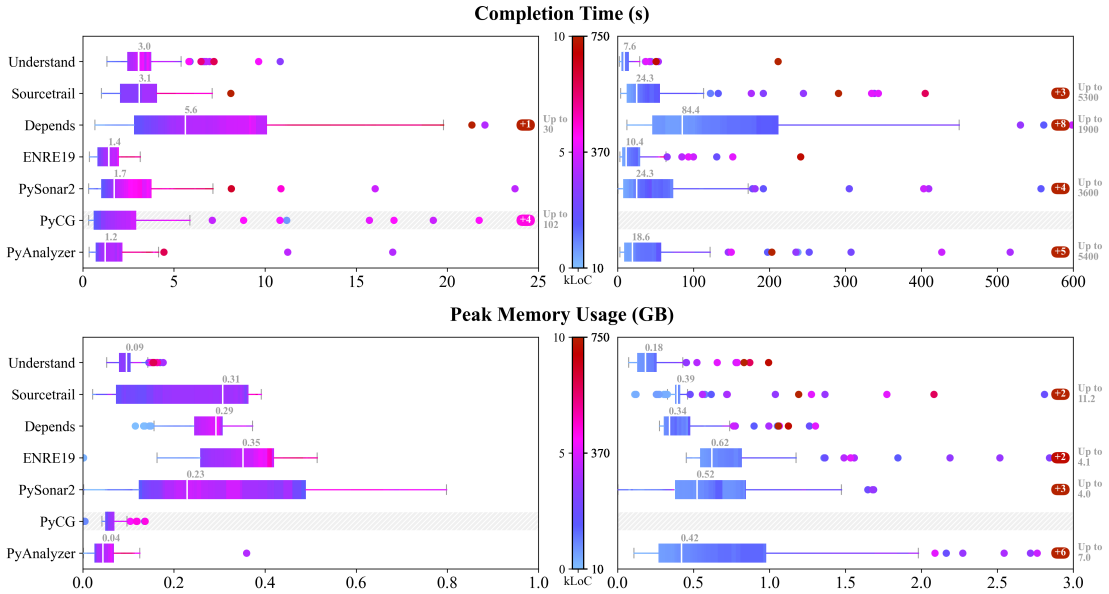


Figure 5: Time consumption and memory consumption of methods in RQ4

Typilus and TypeWriter. Our work focuses on static code analysis instead of models, hence making comparisons with multiple static methods and the advanced Type4Py.

Studies of Dynamic Features. Prior work has summarized dynamic features like duck typing [10], introspection [30], object changes [40], and code generation [28]. Åkerblom et al. [2] inspected execution traces to measure dynamic calls. The work of [25] revealed type annotation practices in Python. Jin et al. [21, 24] explored the impact of nondeterministic dependencies on software analysis. Those studies highlight the need to reveal code dependencies precisely, which motivates our work.

8 CONCLUSION

We have proposed a dependency extraction approach namely PyAnalyzer to address the challenges of object changes, first-class citizens, varying call sites, and library dependencies in Python. PyAnalyzer improves the dependency resolution through enhanced points-to relation propagation and the use of type annotation features. Our evaluation has demonstrated a substantial improvement in effectiveness and time efficiency over the state-of-the-art techniques. Our future work will conduct downstream tasks like architecture analysis and vulnerability detection to further demonstrate the superiority of PyAnalyzer.

ACKNOWLEDGMENTS

This work was supported by National Key R&D Program of China (2022YFB2703500), National Natural Science Foundation of China (62232014, 62272377, 62293501, 62293502, 62032010,62372368, 62372367, 62272387), CCF-AFSG Research Fund and Young Talent Fund of Association for Science and Technology in Shaanxi, China.

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, New York, NY, USA, 292–295.

[2] Beatrice Åkerblom and Tobias Wrigstad. 2015. Measuring polymorphism in Python programs. In *ACM SIGPLAN Notices*, Vol. 51. ACM, ACM, New York, NY, USA, 114–128.

[3] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*. ACM, New York, NY, USA, 91–105.

[4] Erik Arisholm, Lionel C Briand, and Audun Foyen. 2004. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on software engineering* 30, 8 (2004), 491–506.

[5] Carliss Young Baldwin and Kim B Clark. 2000. Design rules: The power of modularity. *Industrial and Corporate Change* 1, 1 (2000), 1–10.

[6] Benjamin Cosman, Madeline Endres, Georgios Sakkas, Leon Medvinsky, Yao-Yuan Yang, Ranjit Jhala, Kamalika Chaudhuri, and Westley Weimer. 2020. Pablo: Helping novices debug python code through data-driven fault localization. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 1047–1053.

[7] Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. 2021. PYInfer: Deep Learning Semantic Type Inference for Python Variables. *CoRR* abs/2106.14316 (2021). arXiv:2106.14316 https://arxiv.org/abs/2106.14316

[8] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2019. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, IEEE, Montreal, Quebec, Canada, 46–57.

[9] daneads. 2007. Python Call Graph. Retrieved 2023-07-31 from https://github.com/daneads/pycallgraph2

[10] Python docs. 2001-2022. https://docs.python.org/3/glossary.html#term-duck-typing.

[11] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 760–771. https://doi.org/10.1145/3540250.3549126

[12] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices* 29, 6 (1994), 242–256.

[13] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. 2016. Automatic detection of instability architectural smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, IEEE

Computer Society, Raleigh, North Carolina, USA, 433–437.
[14] Python Software Foundation. 2003-2023. PyPI · The Python Package Index. https://pypi.org
[15] Python Software Foundation. 2023. https://docs.python.org/3/reference/.
[16] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. 2018. Static value analysis of Python programs by abstract interpretation. In *NASA Formal Methods: 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings 10*. Springer, Springer, Cham, 185–202.
[17] Ritu Garg and Rakesh Kumar Singh. 2022. SBCSim: Classification and Prioritization of Similarities Between Versions. *International Journal of Software Innovation (IJSI)* 10, 1 (2022), 1–18.
[18] giampaolo. 2014-2022. https://github.com/giampaolo/psutil.
[19] Instagram. 2017-2022. https://github.com/instagram/MonkeyType.
[20] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. 2020. An Empirical Study on ARM Disassembly Tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. ACM, New York, NY, USA, 401–414. https://doi.org/10.1145/3395363.3397377
[21] Wuxia Jin, Yuanfang Cai, Rick Kazman, Gang Zhang, Qinghua Zheng, and Ting Liu. 2020. Exploring the Architectural Impact of Possible Dependencies in Python Software. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, ACM, New York, NY, USA, 1–13.
[22] Wuxia Jin, Yuanfang Cai, Rick Kazman, Qinghua Zheng, Di Cui, and Ting Liu. 2019. ENRE: a tool framework for extensible eNtity relation extraction. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, IEEE, Montréal, QC, Canada, 67–70.
[23] Wuxia Jin, Yitong Dai, Jianguo Zheng, Yu Qu, Ming Fan, Zhenyu Huang, Dezhi Huang, and Ting Liu. 2023. Dependency Facade: The Coupling and Conflicts between Android Framework and Its Customization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, IEEE, Melbourne, Australia, 1674–1686.
[24] Wuxia Jin, Dinghong Zhong, Yuanfang Cai, Rick Kazman, and Ting Liu. 2023. Evaluating the impact of possible dependencies on architecture-level maintainability. *IEEE Transactions on Software Engineering* 49, 3 (2023), 1064–1085.
[25] Wuxia Jin, Dinghong Zhong, Zifan Ding, Ming Fan, and Ting Liu. 2021. Where to Start: Studying Type Annotation Practices in Python. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, IEEE, Luxembourg , Luxembourg, 529–541.
[26] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices* 48, 6 (2013), 423–434.
[27] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3, Article 62 (jun 2020), 38 pages. https://doi.org/10.1145/3383458
[28] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA, USA, 2613–2630.
[29] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using S park. In *Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 12*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 153–169.
[30] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 2 (2019), 1–50.
[31] Jingwen Liu, Wuxia Jin, Qiong Feng, Xinyu Zhang, and Yitong Dai. 2021. one step further: investigating problematic files of architecture anti-patterns. In *2021 IEEE 32st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, IEEE, Wuhan, China, 1–12.
[32] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 539–564.
[33] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, Montréal, QC, Canada, 304–315.
[34] Meta. 2018-2023. https://pyre-check.org.
[35] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical deep similarity learning-based type inference for Python. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, New York, NY, USA, 2241–2252.
[36] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2019. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering* 47, 5 (2019), 1008–1028.
[37] Ran Mo, Will Snipes, Yuanfang Cai, Srini Ramaswamy, Rick Kazman, and Martin Naedele. 2018. Experiences applying automated architecture analysis tool suites. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, ACM, New York, NY, USA, 779–789.
[38] Multilang-depends. 2018-2022. https://github.com/multilang-depends/depends.
[39] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. ACM, Shanghai, China, 452–461.
[40] J Palsberg. 1991. Object-oriented type inference. In *Proc. OOPSLA'91*. ACM, New York, NY, USA, 146–161.
[41] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.
[42] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, New York, NY, USA, 2019–2030.
[43] Yun Peng, Yu Zhang, and Mingzhe Hu. 2021. An Empirical Study for Common Language Features Used in Python Projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 24–35.
[44] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. 2009. Using information retrieval based coupling measures for impact analysis. *Empirical software engineering* 14, 1 (2009), 5–32.
[45] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 209–220.
[46] Python. 2001-2022. https://www.python.org/dev/peps/pep-0484/.
[47] python. 2015-2023. typeshed. https://github.com/python/typeshed
[48] Python Software Foundation. 2021. *Python AST (Abstract Syntax Trees)* (3.10.0 ed.). Python Software Foundation, Wilmington, DE. https://docs.python.org/3/library/ast.html.
[49] Jonathan Raiman. 2022. Deeptype 2: Superhuman entity linking, all you need is type interactions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. AAAI, Virtual, 8028–8035.
[50] Leodanis Pozo Ramos. [n. d.]. Python Scope & the LEGB Rule: Resolving Names in Your Code. https://realpython.com/python-scope-legb-rule/
[51] Veselin Raychev, Martin Vechev, and Andreas Krause. 2019. Predicting program properties from'big code'. *Commun. ACM* 62, 3 (2019), 99–107.
[52] Jukka Ruohonen, Kalle Hjerppe, and Kalle Rindell. 2021. A large-scale security-oriented static analysis of python packages in PyPI. In *2021 18th International Conference on Privacy, Security and Trust (PST)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 1–10.
[53] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, IEEE, Madrid, Spain, 1646–1657.
[54] Darius Sas, Paris Avgeriou, Ronald Kruizinga, and Ruben Scheedler. 2021. Exploring the relation between co-changes and architectural smells. *SN Computer Science* 2, 1 (2021), 1–15.
[55] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (apr 2015), 1–69. https://doi.org/10.1561/2500000014
[56] Ioana Şora. 2016. Helping program comprehension of large software systems by identifying their most important classes. In *Evaluation of Novel Approaches to Software Engineering: 10th International Conference, ENASE 2015, Barcelona, Spain, April 29–30, 2015, Revised Selected Papers 10*. Springer, Springer, Cham, 122–140.
[57] Sourcetrail. 2014-2022. https://www.sourcetrail.com/.
[58] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1049–1060.
[59] Symbolk. 2020-2022. https://github.com/Symbolk/Code2Graph.
[60] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.
[61] Technologicat. 2009-2021. pyan. https://github.com/Technologicat/pyan
[62] Manas Thakur. 2020. How (not) to write java pointer analyses after 2020. In *Proceedings of the 2020 acm sigplan international symposium on new ideas, new paradigms, and reflections on programming and software*. ACM, New York, NY, USA, 134–145.
[63] SciTools Understand. 1996-2023. https://scitools.com/.
[64] Yin Wang. 2022. pysonar2. https://github.com/yinwang0/pysonar2/tree/f47662443310200755cbfa9a3bc020efc1a442de
[65] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020, April 26-30, 2020*. OpenReview.net, Addis Ababa, Ethiopia. https://openreview.net/forum?id=Hkx6hANtwH