AexPy: Detecting API Breaking Changes in Python Packages

Xingliang Du
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
xingliangdu@smail.nju.edu.cn

Jun Ma
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
majun@nju.edu.cn

***Abstract*—With the popularity of the Python language, community developers create and maintain a lot of third-party packages. APIs change frequently during the package evolving. Package developers need keeping backward compatibility of APIs to avoid breaking client code. Detecting breaking changes in Python packages is challenging because of Python's dynamic features and flexible designs, such as dynamic typing, API aliases, confusing public boundary and flexible argument passing. Despite the language's popularity, there have been few tools aiming to detect breaking changes, and existing approaches lack sufficient consideration of the mentioned challenges, resulting in imprecise and incomplete detection. Briefly, we propose an API-model-based systematic approach to address this problem. We design a Python-specific API model and classify API changes in different breaking levels. Based on the model, we obtain APIs with their types by a robust hybrid analysis and detect graded changes by checking constraints on API pairs. We implement a prototype, AexPy. Thanks to the more comprehensive model and hybrid analysis adopted, AexPy outperforms the state-of-the-art techniques for Python with an 86.9% recall on a dataset of 61 known breaking changes. Besides, AexPy detects 405 (manually verified) high and medium breaking changes with a precision of 93.5% on the latest versions of 45 packages. Specifically, in addition to 291 documented breaking changes, AexPy detects 114 undocumented changes. We report 63 undocumented breaking changes to active package developers, and 31 have been confirmed.**

***Index Terms*—Python, application programming interface, backward compatibility, breaking change**

I. Introduction

The convenient standard library and the large active third-party package[1] ecology of Python provide tools suited to many tasks (e.g., deep learning, automatic scripts, data processing) and help developers to achieve their ideas efficiently. However, during the evolution of a package, the package's developer may redesign data structures, change operations, or adjust specification due to fixing bugs or providing new features. Some of these changes may break client code, so called "breaking changes" [2]. One common change case is application programming interface (API) changes. APIs describe the public interfaces of a package, which are contracts that clients rely on [3], so an API change could affect clients. In the evolving of commonly used Python framework packages, more than 40% API changes are breaking [4], which is larger than static languages [5], [6]. These breaking changes break backward compatibility, decrease the evolving stability, and might cause potential bugs in clients.

Detecting breaking changes in Python packages is challenging because of Python's dynamic features and flexible designs, such as dynamic module importing, dynamic typing [7], multiple API references, and flexible argument passing [8], which altogether increase API complexity and introduce a variety of API changes. Existing approaches, such as pidiff [9] and PyCompat [4], lack sufficient consideration of these challenges, which leads to imprecise and incomplete results.

In this paper, we propose an automatic and systematic approach to detect API breaking changes in Python packages, which helps Python developers to evolve packages reliably. Our approach works in three major steps, 1) extracting APIs from different versions of Python packages, 2) comparing APIs to detect changes in different patterns, and 3) evaluating backward compatibility of changes to grade breaking levels.

To do so, we propose a model of Python package APIs, taking into consideration of primary features of Python (e.g., inheritance, argument passing, aliases, types). Specially, we model the signature types of functions to address the challenge of dynamic typing. Based on the model, we combine dynamic reflection with static analysis for API discovery and information enrichment to obtain a detailed API description, including types, parameters, aliases, and inheritance. Then we systematically classify API changes into 42 patterns. We design an entry pairing algorithm to adapt to Python's name resolution and a constraint-based checking algorithm to detect changes automatically. Finally, for practicality, we grade changes into four breaking levels according to the API scope, change pattern, and change content, indicating different severities of the changes. Specially, we use subset relationship between types to model compatibility of type changes.

We implement the proposed approach in a prototype named AexPy. We evaluate the effectiveness and efficiency of the tool by applying it to detect known/unknown breaking changes of different packages. Specifically, we compare AexPy with two existing Python API breaking change detectors, pidiff [9] and

This work was supported by Natural Science Foundation of China (Grant No. 62025202). The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Jun Ma is the corresponding author.

[1]We use the word "package" to represent "library", to keep consistence with Python's naming habit, e.g., the Python Package Index [1].

2332-6549/22/$31.00 ©2022 IEEE
DOI 10.1109/ISSRE55969.2022.00052

PyCompat [4]. AexPy benefits from the detailed model and hybrid analysis, and achieves almost 50% improvement on recall with comparable time performance. Meanwhile, in latest versions of 45 packages, AexPy detects high/medium breaking changes in 43 packages with a manually verified precision of 93.5%. So far, we have received 50 responses of 63 reported undocumented breaking changes, 31 of which are confirmed, indicating our approach is practically useful in real world.

To summarize, this work makes three major contributions:

- We systematically design a Python package API model and an API change classification. We grade changes into four breaking levels to adapt to the flexibility of Python APIs, and improve practicality of the model. (Section III)
- We propose an automatic and systematic approach for API breaking change detection in Python packages. To address challenges from Python's dynamic features and flexible designs, our approach extracts APIs through dynamic reflection and static analysis, and detects breaking changes by a constraint-based checking algorithm. To our best knowledge, we propose the first approach to detect type breaking changes in Python packages, where we check change compatibility by subset relationship between statically constructed types. (Section IV)
- We implement a prototype AexPy, to detect breaking changes in real-world packages. We evaluate AexPy on collected 61 known breaking changes and latest versions of 45 packages. Our approach achieves 86.9% recall, outperforming existing tools, and detects high and medium breaking changes in the latest versions with 93.5% precision. Of the 63 reported changes, 31 have been confirmed by package developers. (Section V)

The rest of this paper is organized as follows. Section II introduces the background and challenges on breaking change detection in Python packages. Section III depicts the design of our API model and the classification of breaking changes. Section IV details our detection method as well as our prototype AexPy for detecting breaking changes. Section V describes our evaluations, compared with existing approaches. Section VI discusses strengths, limitations, and validity. After giving an overview on related works in Section VII, we conclude this paper and discuss future works in Section VIII.

II. Background

A. API Backward Compatibility

For a software package, its APIs describe and prescribe the "expected behaviors", which contain descriptions about data structures, and operations on such data [3]. Backward compatibility is a property that the system allows interoperating with an older legacy system, e.g., dealing with inputs from the old system. Modifying a system in a way that does not follow this property is called "breaking" backward compatibility, i.e., breaking change [2]. Backward compatibility of a package means that clients can use the new version of the package in the same ways and get the same behaviors as the old version, i.e., the package have compatible APIs on syntax and semantic [2]. API breaking changes would cause compilation or runtime problems in client code. For example, signature changes in Java APIs would cause compilation errors, and function removals in shared objects (dynamic link libraries) for C/C++ could cause runtime linking errors.

B. Breaking Changes in Python Packages

Python is an interpreted, dynamically-typed, general-purpose programming language, which consistently ranks as one of the most popular programming languages [10]–[13]. Python supports multiple programming paradigms, including procedural, object-oriented and functional programming. Python Package Index (PyPI) [1], the official repository for third-party Python packages, contains over 329,000 packages by April 2022. These third-party packages cover a wide range of functionality, such as automation, machine learning, networking, scientific computing, web frameworks, and so on.

Breaking changes in Python packages happen frequently and have extensive and expensive impacts. Zhang *et al.* [4] investigated six Python frameworks and their clients, finding that breaking changes occurred more frequently than Java, and the changes affected more than half of the clients. They also investigated 409 compatibility issues on GitHub, and showed that 405 of them caused crashes at runtime, which showed the extensive and expensive effects on breaking changes.

Detecting breaking changes in Python packages is different and challenging compared to traditional programming languages. For example, Python developers often define instance attributes by assignment statements in constructors instead of the class body in C++/Java. Zhang *et al.* [4] found that the Python APIs have different evolution patterns on parameters. Peng *et al.* [14] studied language feature usage in 35 popular Python projects and found that inheritance, decorators, positional-or-keyword parameters are frequently used.

CPython [15] is the standard Python implementation. We learn about the Python language with CPython's runtime features and summarize following four challenges on API breaking change detection in Python packages.

1) Dynamic Language Features: It is difficult to collect API metadata statically because of Python's dynamic language features. Dynamic programming languages are a class of high-level programming languages, which at runtime execute many common programming behaviors that static programming languages perform during compilation (e.g., extending objects and definitions). Python provides language-level mechanisms such as decorators, metaclasses [8], class hooks to modify classes dynamically. These dynamic features provide flexibility to programmers while increasing difficulty of static analysis.

Besides, the dynamic type system in Python brings challenges to type compatibility checking. Firstly, CPython only checks type compatibility at runtime, so type-related errors occur during executing but cannot be easily detected during programming, especially between two versions. Secondly, Python has a complex type system and no enforcing type declarations, introducing complex and various type changes. Python allows duck-typing, i.e., an object is of a given type if it has all methods required by that type [8], and provides abstract base classes (ABCs), which allow the classes, that do not inherit from a class but are recognized as its subclasses [8]. Python introduces optional annotations [8], which are used by static analyzers [16] for type checking in code, and are ignored at runtime. This helps to obtain types, but it is still difficult to compare types between versions for type change detection.

2) Complex API References: It is difficult to locate and distinguish APIs because Python allows referencing the same API by different names. Python is an object-oriented programming language with multiple inheritance and allows flexible object definition, e.g., member adding or removing at runtime. Most programming elements such as modules, classes, functions, are objects, and can be stored in a variable or a member of another object. One API object can be accessed by different names and a member name can target to an API from another module, even from another package (i.e., external APIs). The feature is widely-used in popular packages, such as NumPy [17], Tensorflow [18], to create short names for convenience, while leading to a complex crossing API structure.

3) Fake Private Members: The boundary of public APIs is fuzzy in Python. Python has no access modifiers, which makes most objects accessible, leading to confusion on distinguishing the scope of APIs. Python considers class members whose names start with double underscores as private, and mangles their names to prevent accessing such members [19]. But there is no such mechanism for module members. Python developers often have their own private-naming conventions, e.g., members are private if their names start with an underscore or if they belong to a specific module. Because Python has no enforcing constraints on these conventions, clients can still access such members just like normal members, which makes it complex to cover these changes. Firstly, API aliases make it complex to identify these members by conventions. Existing approaches ignore API aliases, and may miss APIs with public aliases. Secondly, these members usually have few uses in clients indicated by the conventions, but their changes still may break, while existing approaches ignore them.

4) Flexible Argument Passing: Python has a flexible argument passing design. Firstly, a default value can be specified to make the parameter optional. Secondly, when calling a function, arguments can be passed as either position or keyword arguments [8]. Thirdly, functions can accept unbound arguments. Python collects unbound positional arguments as a list, like variadic functions in C++ [20] or Java [21]. Specially, Python collects unbound keyword arguments as a dictionary. Table I summarizes passing behaviors for parameter kinds.

TABLE I
PARAMETER KINDS [8]

Kind	Behavior
Positional	Pass by position
Keyword	Pass by name
PositionalOrKeyword	Pass by name if given otherwise by position
VarPositional	List of all unbound positional arguments
VarKeyword	Dictionary of all unbound named arguments

C. Limitations of Existing Approaches

To our best knowledge, there are two existing tools, pidiff [9] and PyCompat [4], for breaking change detection in Python. Pidiff [9] is a tool to help to enforce the usage of semantic versioning (semver) [22] on Python packages. It compares APIs provided by two versions of a pip-installable [23] package, produces a report of detected changes, and warns developers if these changes violate the semver. Pidiff extracts APIs mainly by reflection and obtains instance attributes from source code. Then it classifies and detects three categories of 23 change patterns, including module, object, and signature changes. PyCompat [4] is a semi-automatic API compatibility issue detector designed for clients of six popular Python packages, based on its own API change detector. PyCompat extracts APIs fully by dynamic reflection, and then classifies and detects three categories of 14 change patterns, including class, method, and attribute changes.

According to the challenges mentioned in Section II-B, the existing approaches lack consideration of type changes, class inheritance, API aliases, and fake private members. Besides, pidiff ignores parameter default value changes, while PyCompat ignores parameter kinds. PyCompat also ignores nested modules, which leads to the lack of deep APIs and their changes. In addition, PyCompat cannot be used directly for the packages other than those six packages. These limitations lead to imprecise and(or) incomplete detection results and low availability. To address this problem, we need a more comprehensive, systematic classification for breaking changes, and a more automatic, robust detection tool.

III. Backward Compatibility Modeling

This paper focuses on the backward compatibility of API syntax, i.e., the way to use the package does not change. We propose a model of Python package APIs, suited with Python features mentioned in Section II-B. Then we classify API changes and grade them into different breaking levels to form a comprehensive and practical breaking change model.

A. Python API Model

According to the Python modular structure, we consider the set E of APIs, i.e., the union of four sets M, C, F, A, representing modules, classes, functions, and attributes respectively. For example, the API set corresponding to the code in Figure 1 is $E = \{D, f, A, x, t, init, g, B, h, i\}$.

1) Module: A module $m \in M$ serves as an organizational unit of code, with a namespace containing arbitrary objects [8], such as classes, functions, attributes, or submodules, e.g., the module D. We model the membership by $\mathbf{members}(m)$, which contains all members of m and their corresponding member names, such as definitions of classes and functions, references to external objects. In formal, $\mathbf{members}(m)$ is a mapper in the form $\mathbf{string} \rightarrow (E \cup \{\perp\})$, from the member name to the target API ($\perp$ for external APIs). The module D in the example contains one function, two classes, and a reference to an external class in `typing`, i.e., $\mathbf{members}(D) = \{\text{“opt”} \rightarrow \perp, \text{“f”} \rightarrow f, \text{“A”} \rightarrow A, \text{“B”} \rightarrow B\}$.

```python
# File: D.py
from typing import Optional as opt
def f(a, /, b = []): pass
class A:
  x: int = 0
  t = f
  def __init__(self): self.y: "int" = 0
  @classmethod
  def g(cls, *, c: opt['abc']=None): pass
class B(list, A):
  def h(self)->'opt[A]': return self
  @staticmethod
  def i(*ar, **kw)->str: return str(kw['v'])
```

Fig. 1. API Example

2) Class: A class $c \in C$ is a template for creating user-defined objects, which contains method definitions operating on instances of the class [8], e.g., A, B. We describe its membership by $\textbf{members}(c)$ which ignores base classes' members, e.g., $\textbf{members}(B) = \{\text{“h”} \rightarrow h, \text{“i”} \rightarrow i\}$. We model inheritance in three properties.

Base Class: the set of classes which c explicitly inherits from, e.g., $\textbf{bases}(B) = \{list, A\}$.

Abstract Base Class: the set of widely-used builtin ABCs [24], which c can be recognized as, e.g., $\textbf{abcs}(B) = \{Iterable, Sequence\}$ because $list$ supports these ABCs.

Method Resolution Order: an ordered class tuple for searching a method during lookup [8], affecting APIs indirectly, e.g., $\textbf{mro}(B) = (B, list, A, object)$.

3) Function: A function $f \in F$ is the main part of package interfaces, e.g., f, g, h, i. We focus on its signature including the parameters and return type, e.g., $\textbf{paramaters}(f) = \{a, b\}, \textbf{return}(i) = \textbf{str}$. We consider three scopes, i.e., where the function is bound, e.g., $\textbf{scope}(f) = \textbf{scope}(i) =$ Static, $\textbf{scope}(g) =$ Class, and $\textbf{scope}(h) =$ Instance. Specifically, we treats the functions defined inside a class, i.e., methods, whose first parameter is `self`, like h, as instance scope, following the widely-adopted convention [8].

A parameter is a named entity in a function definition that specifies an argument that the function can accept [8], e.g., the parameters $a, b, self, c, ar, kw$. For a parameter p, we focus on its name, position, optionality, literal default value, type, and especially, passing kind to model argument passing, e.g., $\textbf{name}(a) =$ “a”, $\textbf{position}(b) = 2$, $\textbf{optional}(c) =$ True, $\textbf{default}(c) =$ None, $\textbf{kind}(a) =$ Positional, and $\textbf{kind}(kw) =$ VarKeyword. The type of parameter c is a little complex, so we explain it in Section III-A5. We find that many developers access `VarKeyword` parameter with specified keywords in the function body, e.g., the parameter v of the function i, so we introduce a new kind of parameters VarKeywordCandidate, indicating the keywords accessed from `VarKeyword`.

4) Attribute: An attribute $a \in A$ is a value associated with an object and is referenced by its name using dotted expressions [8], e.g., x, y. We focus on its type and scope, e.g., $\textbf{type}(x) = \textbf{int}$, $\textbf{scope}(x) =$ Class, and $\textbf{scope}(y) =$ Instance, because y is defined in the constructor of A, so y can only be accessed by the instances of A, instead of A itself.

5) Type: In the model described above, we consider types of parameters and attributes. Type annotations [8] help to deal with dynamic typing, but they depend on the context, which cannot be used between versions, e.g., the annotation of h is a string containing a reference `opt` from the context, and it has the same meaning as `Union[A, None]`, although the strings are different. So we design a simple, portable type model, covering main Python types, and use it to check type change compatibility described in Section IV-C3. In our type model, a type T is a set of the objects of that type. There are following three kinds of atomic types, e.g., both $\textbf{type}(x) = \textbf{int}$ and $\textbf{return}(i) = \textbf{str}$ are class types.

Literal: specific string, boolean, number literals, e.g., “abc”.
Class: types of a class for non-specific values, e.g., **int**, **A**.
Special: **none**, **any**, and **unknown** (uncovered types).

There are four kinds of composite types, which combine other types in a specific semantic structure. For example, the annotation of the parameter c is `opt['abc']`, in which `opt` references `Optional`. It means c accepts the string literal “abc” or `None`, so we model it by a sum type, $\textbf{type}(c) =$ “abc” $+$ **none**. Then we can judge that it is structurally identical to `Union[None, 'abc']`.

Sum: $T_1 + T_2 + \cdots + T_n$, indicating objects that are of at least one of $T_1, \ldots, T_n$.

Product: $T_1 \times T_2 \times \cdots \times T_n$, indicating objects which are composites of n ordered objects, and the i-th is of T_i.

Callable: $T_{args} \rightarrow T_{ret}$, indicating callables that accept T_{args} and return T_{ret}, e.g., `strlen()` is of $\textbf{str} \rightarrow \textbf{int}$.

Generic: $T_{base}(T_1, T_2, \ldots, T_n)$, indicating a generic type instance, with base type T_{base} and type arguments $T_1, \ldots, T_n$, which is used for array and collection types, e.g., $\textbf{list}(\textbf{float}), \textbf{dict}(\textbf{str}, \textbf{int})$.

6) Alias: To model Python's complex API references, we use the qualified name of e's definition as the identifier of e, e.g., $\textbf{id}(f) =$ “D.f” and $\textbf{id}(g) =$ “D.A.g”, and treat all references targeting to e, except for the definition, as the aliases of e, noted as $\textbf{aliases}(e)$. For example, the member named “t” in the class A references the function f in the module D, so “D.A.t” is an alias of f.

B. Change Classification

According to our proposed API model above, we classify changes into 17 coarse-grained patterns based on three forms of changes (additions, removals, and modifications) that occur directly on six categories of APIs, as shown in Table II.

We then refine these change patterns into 42 fine-grained patterns[2] to give a more detailed classification. For class, we partition inheritance changes by its cause, i.e., base class, ABC, and MRO changes. For functions and attributes, we partition their additions and removals by their scope to distinguish instance members. For parameters, we partition their additions and removals by their kinds and optionality, and partition

[2]Due to space limitations, we do not include all the 42 patterns, which are available at https://aexpy.netlify.app/change-spec.

TABLE II
API CHANGE PATTERN CLASSIFICATION

	Module	**Class**	**Function**	**Attribute**	**Parameter**	**Alias**
Addition	AddModule	AddClass	AddFunction	AddAttribute	*AddParameter*[*]	AddAlias
Removal	*RemoveModule*	*RemoveClass*	*RemoveFunction*	*RemoveAttribute*	*RemoveParameter*	*RemoveAlias*
Modification	-[†]	*ChangeInheritance*	*ChangeReturnType*	*ChangeAttributeType*	*ChangeParameter*	ChangeAlias

[*] Italic change patterns contain breaking changes.
[†] There are no change instances of module modification changes under this classification.

their modifications by its cause, as shown in Figure 2. For alias changes, we distinguish external references by the API location. It needs to note that these API changes are different from code changes, because one single code change can lead to multiple API changes, e.g., a parameter addition at the first place causes Add&MoveParameter changes.

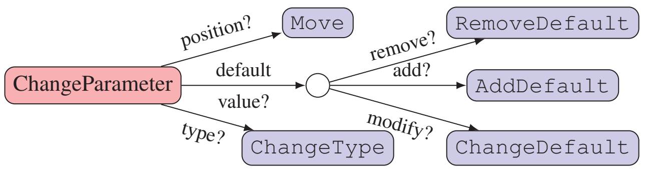

Fig. 2. Refined Classification for ChangeParameter

C. Breaking Grading

According to our observation of repositories and issues, we set up a heuristic breaking level (including *Compatible*, *Low*, *Medium*, and *High*) to each change, indicating the change's severity, i.e., ranges of impact situations. Because of the weak constraints on Python APIs and the lack of compilation type checking, developers have difficulties on restricting client usage. Multiple breaking levels help developers to focus on important changes while getting full results, to improve practicality and keep comprehensive.

1) Compatible: Changes that have no breaking impact on old programs are compatible, such as AddClass and AddAlias.

2) Low: We grade all breaking changes on fake private APIs or aliases of external APIs to low level, to reduce disturbance, because clients usually do not use these APIs directly although they can, and the package developers may provide no compatibility guarantees on these APIs. We specify the details of determining private APIs in Section IV-C1.

3) Medium: We grade changes which indirectly breaks usability of public APIs to medium level, because the breaking situation is conditional and rare. For example, unlike on normal functions, parameter changes on class/instance methods can lead to inconsistency during overriding. Because of Python's overriding design [25], when a subclass overrides the method, the new method copies the signature and replaces the original method. When an signature change occurs on the original method, the signature in the subclass is different from the base, which breaks the overriding convention.

Type incompatibility changes are another important pattern of medium breaking changes. Because of duck-typing, the errors caused by these changes, occur only when the parameter or attribute is accessed or explicitly type-checked in function logics, instead of function invocation. We specify the details about type compatibility in Section IV-C3.

4) High: We grade changes which directly break usability of public APIs to high level. For example, RemoveModule and AddRequiredParameter are high breaking because programs that access the module or call the function will crash because of failing to find modules, or missing required arguments.

IV. BREAKING CHANGE DETECTION

According to our API model and breaking change classification, we propose an automatic method to detect breaking changes, and implement a prototype, AexPy. As shown in Figure 3, AexPy takes two versions of a package as inputs, processes package releases through four stages, and outputs graded API changes. AexPy prepares the distributions and metadata of the versions when preprocessing, and the subsequent stages are described in the following subsections.

A. API Extracting

As mentioned in Section II-B, Python package API extraction is non-trivial. AexPy combines dynamic reflection and static analysis to address those challenges. Specifically, AexPy first discovers and extracts APIs dynamically by reflection to handle dynamic behaviors caused by dynamic features, and then enriches API information by static analysis.

1) Dynamic Reflection Analysis: Python's standard library provides reflection tools to access metadata of runtime objects. Python finds modules by the corresponding directory structure as default [26]. Modules `importlib` and `pkgutil` support standard import mechanisms [27], [28], and module `inspect` provides several useful functions to inspect live objects [29].

AexPy discovers all modules in the package and tries to import them to collect APIs. First, AexPy installs the package by the Python official package manager `pip` [23] into an available environment with the matched Python interpreter version. Then through a top-down and breadth-first search from the package's top-level modules, AexPy discovers all accessible APIs, handles exceptions to improve robustness, and generates identifiers by builtin location attributes [26] of API objects. Finally, AexPy collects metadata of every detected API object by the `inspect` module, and then generates a collection of APIs in the form of our model.

The dynamic extraction method discovers API definitions and their references precisely. Dynamic importing and runtime

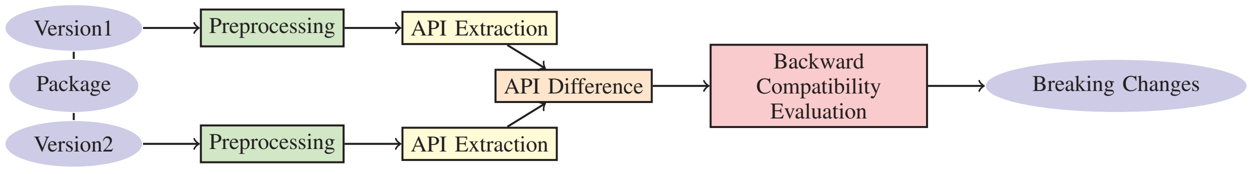

Fig. 3. AexPy's Workflow

reflection addresses object assignments and API modification from decorators. AexPy follows the Python standard to access a package, simulating the client usage, so AexPy discovers what clients can actually access. For API referencing, because the same API referenced by different names is the same object at runtime, AexPy obtains the actual target object of each alias by reflection, identifies these objects by their unique definition locations and then builds a precise API reference relationship.

2) Static Program Analysis: Reflection provides basic information about accessible APIs, but it learns little about function logics, e.g., instance attributes, and candidate keyword parameters, defined in function bodies. Reflection also reads type annotations, but cannot recognize the same type from different annotations. So we map dynamically-extracted APIs to source code by their definition locations, and introduce static analysis to enrich attributes, parameters, and types.

For instance attributes, AexPy traverses all attribute assignment statements [30] on `self` parameter, i.e., the instance object, from the abstract syntax trees (ASTs) of constructor methods to detect instance attributes. For candidate keyword parameters, AexPy uses a simplified AST-based, flow-insensitive, intra-function alias analysis to find all aliases of the `VarKeyword` parameter (if exists). After that, AexPy detects all access operations on the parameter and its aliases, including subscripts, dictionary methods, to find candidate parameters.

There are multiple approaches to obtain types, and AexPy relies on a popular Python static type checker, Mypy [16], because Mypy has a type system including generics, callable types, and so on, and it supports bidirectional type inference, which partly handles absence of type annotations. The types from Mypy cannot be used directly for compatibility checking between versions, because their contexts in Mypy are bound with the specific version, such as class metadata and type aliases. AexPy hooks Mypy to get its internal type representation and then maps the types of APIs ($\mathbf{type}(e) = \bot$ if Mypy fails) to our model, to detach from Mypy's environment.

B. API Difference

We design a difference algorithm to detect the 42 API change patterns. AexPy first pairs the corresponding APIs between the two versions and then checks the API pairs against the defined constraints corresponding to each change pattern.

1) API Pairing: API pairing contains two parts, API entry pairing, and function parameter pairing. For an API e from the old version, API entry pairing finds the API e' from the new version, such that e' can be accessed by $\mathbf{id}(e)$ or one of $\mathbf{aliases}(e)$. Specifically, AexPy first finds e', which has the same identifier and category (e.g., module, class) of e. If no such e', AexPy then locates the aliased API by resolving $\mathbf{id}(e)$ through a series of member accessing in the new version. Specially for methods defined in a class c, AexPy finds the first existing method with the name through $\mathbf{mro}(c)$ to handle inheritance. This pairing algorithm simulates the actual name resolution at runtime to address multiple aliases. If still no such API ($e' = \bot$), AexPy concludes that e is removed in the new version. AexPy pairs $\bot$ to unpaired APIs of the new version. For paired functions, parameter pairing further finds the parameters between two versions, that accept the same arguments in use cases, to address flexible argument passing. Specifically, AexPy distinguishes passing kinds, and pairs the position, keyword, and variadic parameters separately.

2) Constraint-Based Detection: To implement automatic detection, we associate each change pattern with a logical expression as its constraint to check whether the change occurs on API pairs (e, e') or parameter pairs (p, p'), including testing whether the pair contains $\bot$ to detect additions and removals, and comparing properties on non-$\bot$ pairs to detect modifications. Table III shows some selected typical checking constraints. We take following change patterns as example to explain how we design constraints.

RemoveFunction The first two clauses ($e \in F \wedge e' = \bot$) means the function e is removed, because of no paired entry in the new version, and the last clause ($\mathbf{scope}(e) =$ Static) enforces the function is not bound to any objects, distinguishing this pattern from RemoveMethod.

RemoveBaseClass The first clause ($e, e' \in C$) enforces the class exists in both versions, and the second clause ($\mathbf{bases}(e) \not\subseteq \mathbf{bases}(e')$) means there are some base classes removed in the new version.

MoveParameter The first two clauses ($p \neq \bot \wedge p' \neq \bot$) enforce the parameter p exists in both versions, and the last clause ($\mathbf{position}(p) \neq \mathbf{position}(p')$) means the position of p is different between the two versions, which means p is moved.

RemoveAlias The first clause ($e, e' \in M \cup C$) means e, e' exist and are modules or classes, so they have members. The existential quantified clause enforces alias removals. Specifically, $t \in E$ means t is not external, and $n \in \mathbf{aliases}(t)$ means n is an alias of t. The last clause ($(n, t) \in (\mathbf{members}(e) - \mathbf{members}(e'))$), where the difference between two member sets forms the removed members of e, enforces the alias n of t is removed.

C. Backward Compatibility Evaluation

According to our breaking grading, AexPy evaluates change levels by following three steps.

TABLE III
TYPICAL API CHANGE PATTERN CONSTRAINTS

Pattern	Constraint	Level for Public
AddModule	$e = \bot \wedge e' \in M$	Compatible
RemoveFunction	$e \in F \wedge e' = \bot \wedge \textbf{scope}(e) = \text{Static}$	High
RemoveBaseClass	$e, e' \in C \wedge \textbf{bases}(e) \not\subseteq \textbf{bases}(e')$	High
ChangeReturnType	$e, e' \in F \wedge \textbf{return}(e) \neq \textbf{return}(e')$	*Medium*[*]
AddRequiredParameter	$p = \bot \wedge p' \neq \bot \wedge \neg \textbf{optional}(p')$	High
MoveParameter	$p \neq \bot \wedge p' \neq \bot \wedge \textbf{position}(p) \neq \textbf{position}(p')$	High
RemoveVarKeywordCandidate	$p \neq \bot \wedge p' = \bot \wedge \textbf{kind}(p) = \text{VarKeywordCandidate}$	Medium
RemoveAlias	$e, e' \in M \cup C \wedge (\exists (n, t), t \in E \wedge n \in \textbf{aliases}(t) \wedge (n, t) \in (\textbf{members}(e) - \textbf{members}(e')))$	High
RemoveExternalAlias	$e, e' \in M \cup C \wedge \exists n, (n, \bot) \in (\textbf{members}(e) - \textbf{members}(e'))$	Low

[*] The level of type changes depends on type compatibility, described in Section IV-C3.
[†] Due to space limitations, we do not include all the constraints on 42 change patterns, which are available at https://aexpy.netlify.app/change-spec.

1) Filtering Private APIs: The first step is filtering "private" APIs and aliases of external APIs for low level changes. AexPy follows the widely-adopted convention: the name of "private" members starts with underscores. Specifically, AexPy checks all access paths of the API, and treats it as private if all its aliases contain names starting with underscores.

2) Grading by Patterns: After filtering changes on private APIs out, some patterns have clear severities to grade, e.g., the third column shown in Table III.

3) Grading by Contents: Some medium breaking changes are conditional breaking according our change classification, so we go deep into the change content to grade. For example, parameter changes such as AddOptionalParameter, are compatible for normal functions, but would introduce inconsistency during overriding for class/instance methods, as mentioned in Section III-C3. So AexPy grades these changes to medium level if they occur on class/instance methods, i.e., $\textbf{scope}(f) \neq \text{Static}$, indicating its limited breaking impact.

Type changes are also unable to grade by their patterns, e.g., parameter type changes from $\textbf{str}$ to $\textbf{str} + \textbf{int}$ are compatible while the inverse are not. We model the compatibility by the subset relationship between changed types, defined recursively on structures, shown in the following inference rules, in which "S" and "T" are name prefixes for types.

$$\text{ANY: } \frac{}{T \subseteq \textbf{any}} \qquad \text{SUM: } \frac{i \in [1, n]}{T_i \subseteq T_1 + \cdots + T_n}$$

$$\text{CLASS: } \frac{S \in \textbf{bases}(T) \cup \textbf{abcs}(T) \cup \textbf{mro}(T)}{T \subseteq S}$$

$$\text{PRODUCT: } \frac{T_1 \subseteq S_1 \qquad \ldots \qquad T_n \subseteq S_n}{T_1 \times \cdots \times T_n \subseteq S_1 \times \cdots \times S_n}$$

$$\text{CALLABLE: } \frac{T_{args} \subseteq S_{args} \qquad S_{ret} \subseteq T_{ret}}{T_{args} \to T_{ret} \subseteq S_{args} \to S_{ret}}$$

$$\text{GENERIC: } \frac{T_{base} \subseteq S_{base} \qquad T_1 \subseteq S_1 \qquad \ldots \qquad T_n \subseteq S_n}{T_{base}(T_1, \ldots, T_n) \subseteq S_{base}(S_1, \ldots, S_n)}$$

Taking the CALLABLE rule for example, which rules compatibility of signature type changes. A function type change from $T_{args} \to T_{ret}$ to $S_{args} \to S_{ret}$ is compatible, if and only if both its parameter and return type changes are compatible, i.e., $T_{args} \subseteq S_{args}$, which means it still accepts objects of the old type T_{args}[3] and $S_{ret} \subseteq T_{ret}$, which means it does not return objects of new types other than T_{ret}.

V. EVALUATION

We evaluate AexPy's effectiveness based on following research questions.

RQ1 (Recall) Does AexPy detect more known breaking changes than existing approaches?
RQ2 (Practicality) Can AexPy find potential unknown breaking changes?
RQ3 (Efficiency) What is the time performance of AexPy?

We compare AexPy[4] with pidiff [9] and PyCompat [4] on recall and efficiency. We wrap pidiff and parse its outputs for manually checking. We reimplement PyCompat and adjust its API extraction approach based on its open-source repository [31] to analyze packages besides the six framework packages it is designed for. Both pidiff and PyCompat need package metadata, so we share results from AexPy's preprocessing as their inputs. We run the three tools in containers limited in 50 GBs and one hour for each version pair on an Ubuntu 18.04 host with 12 CPUs of 3.8GHz and 64 GBs of RAM.

A. Detecting Known Breaking Changes

We collect API backward compatibility issue reports and pull requests, solved between January 2021 and May 2022, from GitHub by keyword searching. Selected keywords include: 1) general keywords like "breaking change", "backward compatibility", and 2) keywords about errors/exceptions caused by breaking changes, e.g., "TypeError", "ModuleNotFoundError", "AttributeError", indicating exception types, as well as "missing parameter", "unexpected keyword parameter" in exception messages. From the searched results, we read the source code of the breaking version to ensure it is an actual API breaking change, and determine what the exact change is. We also search documented API breaking changes in the changelogs of latest two major versions from 15 popular packages (according to downloads and GitHub stars). Finally, we collect 61 API breaking changes in different categories

[3]This rule simplifies parameter list compatibility because type checking occurs only on a single parameter after parameter pairing.
[4]AexPy is available at https://github.com/StardustDL/aexpy.

TABLE IV
RESULTS ON KNOWN BREAKING CHANGES

Tool	**Module** (2)	**Class** (4)	**Function** (6)	**Attribute** (5)	**Parameter** (34)	**Alias** (4)	**Type**(6)*	**Total** (61)
pidiff	0	1	1	1	16	4	0	23
PyCompat	0	1	2	0	17	2	0	22
AexPy	2	4	6	4	29	4	4	53

* We filter type changes specially out from function, attribute, and parameter changes, to show our effectiveness on type changes.

from 45 open-source packages[5] lying in different areas, such as web, utility, artificial intelligence.

For each known breaking change, we execute the three tools on the corresponding version pair. AexPy crashes on 5 changes, while pidiff crashes on 31, and PyCompat crashes on 13, mainly caused by exceptions on installing and importing. We read the results manually to check whether the tools detect the changes. Table IV presents the number of the breaking changes detected by each tool in each category.

AexPy detects the most known breaking changes in all categories and covers all the known changes detected by other tools. AexPy detects 86.9% (53/61) changes (including 2 low level changes), increasing by almost 50% compared to pidiff and PyCompat. Especially, AexPy detects 66.7% (4/6) type changes while no existing tool detects. Restricted to the known changes for which none tool crashes, AexPy finds 6 more than pidiff and 23 more than PyCompat. The high recall shows our approach detects more breaking changes.

B. Finding Unknown Breaking Changes

To check the usefulness of AexPy for actual development situations, we evaluate AexPy on the latest versions (against their previous ones) of the same 45 selected open-source packages used in the previous evaluation.

Except compatible and low level changes, AexPy detects 433 changes (312 high, 121 medium) on those version pairs. Among them, 405 changes (294 high, 111 medium) are manually checked as true, with a precision of 93.5%. We read changelogs of those versions (if exists) and mark 291 documented changes. The rest 114 of breaking changes are undocumented, and we report 63 changes among them to active package developers. AexPy crashes on bentoml [33] 0.13.1 and clyngor [34] 0.4.2. Table V presents the results on the rest 43 packages, including counts of detected high/medium changes and confirmations on reported issues.

AexPy finds meaningful results on most packages. In 14 packages, AexPy finds no breaking changes, in which 13 versions are patch updates, and the rest one version [36] also keeps its API syntax. In the other packages, most detected breaking changes are documented in changelogs, although most changelogs are high-level with only brief notes on API changes, while AexPy gives details.

AexPy detects undocumented breaking changes confirmed by developers, indicating our approach is useful on checking

[5]Collected data are available at https://aexpy.netlify.app/data. We exclude Tensorflow [18] and PyTorch [32], which cause importing crashes in non-specific analysis environments because of their native dependencies.

TABLE V
RESULTS FOR HIGH/MEDIUM CHANGES ON LATEST VERSIONS

Package	**Find**	**TP**	**FP**	**Doc.**	**Rep.***	**C/I***	**#Issue**
Flask & Other 13†	0						
trio 0.20.0	2	0	2				
python-dateutil 2.8.2	12	0	12				
asyncpg 0.25.0	1	1	0	1	0		
urllib3 1.26.9	1	1	0	1	0		
pooch 1.6.0	2	2	0	2	0		
jmespath 1.0.0	3	3	0	3	0		
Django 4.0.4	4	4	0	4	0		
pystac 1.4.0	4	4	0	4	0		
PyYAML 6.0	4	4	0	4	0		
scikit-learn 1.0.2	4	2	2	2	0		
harvesters 1.3.6	5	5	0	5	0		
PyJWT 2.3.0	6	6	0	6	0		
humanize 4.0.0	9	9	0	9	0		
Jinja2 3.1.2	13	12	1	12	0		
captum 0.5.0	17	17	0	17	0		
ao3-api 2.2.1	20	20	0	20	0		
xarray 2022.3.0	24	24	0	24	0		
pecanpy 2.0.2	27	27	0	27	0		
appcenter 3.0.0	51	51	0	0	0‡		
gradio 2.9.4	1	1	0	0	1	**1**/0	#1169
diffsync 1.4.3	3	3	0	0	3	**3**/0	#108
rpyc 5.1.0	6	6	0	3	3	**3**/0	#488
evidently 0.1.49.dev0	9	9	0	0	9	**9**/0	#216
prompt-toolkit 3.0.29	12	12	0	3	9	-	#1627
tornado 6.1	20	12	8	8	4	**0**/4	#3138
pybinance 1.0.16	23	23	0	19	4	-	#1182
astroquery 0.4.6	34	34	0	19	15	**2**/13	#2397
pyoverkiz 1.4.0	41	41	0	32	9	**9**/0	#477
stonesoup 0.1b8	75	72	3	66	6	**4**/2	#631
Total	433	405	28	291	63	**31**/19	-

* Rep. column represents the count of reported changes. C/I column represents the count of confirmed/ignored changes (dash "-" for no reply).
† We omit the items where AexPy detects no breaking changes: Flask 2.1.2, betfairlightweight 2.16.4, catkin-tools 0.8.5, click 8.1.3, docspec 2.0.1, meshio 5.3.4, paramiko 2.10.4, requests 2.27.1, resolvelib 0.8.1, Scrapy 2.6.1, markupsafe 2.1.1, numpy 1.22.3, matplotlib 3.5.2, pandas 1.4.2.
‡ We do not report as appcenter [35] is inactive and has no changelog.

backward compatibility in actual situations. We group the 63 high/medium breaking changes into 10 issues by their packages and report them. So far, we have received 8 replies involving 50 changes from developers, in which 31 changes are confirmed. The package developers agreed that the rest 19 changes are true but they ignored them because the related APIs (although not recognized as "private") are actually implementation details with no compatibility guarantees according to developers' designs. For example, a developer from tornado [37] replied "I'm sorry you had a frustrating experience due to these changes, but these were all undocumented implementation details that had no guarantees of compatibility."

C. Time Performance

To evaluate efficiency, we run the three tools under the same environment on every pair of adjacent versions from selected 45 packages. We ignore the versions which only support end-of-life Python, i.e., Python 3.6 and before [38]. We measure execution time by Python's `timeit` module [39], and calculate average execution durations for packages with different numbers of code lines. Table VI presents the result.

TABLE VI
AVERAGE TIME (S) ON PACKAGES WITH DIFFERENT LOCS

	0-5k (14)	**5-10k** (12)	**10-100k** (14)	**100-500k** (5)
pidiff	66.4	89.4	216.6	231.3
PyCompat	59.7	47.1	113.5	115.8
AexPy	94.6	103.2	269.5	489.3

The results show that the time performance of the three tools is in the same order of magnitude. AexPy takes more time because it detects deep nested modules and introduces static analysis with type checking to improve analysis results.

VI. DISCUSSIONS

A. Strengths

There are three major advantages and we discuss how our approach addresses the challenges in Section II-B as follows.

1) More Detailed API Descriptions: AexPy extracts APIs more precisely according to our Python-specified API model. To address dynamic language features, we find APIs and handle dynamic behaviors by dynamic reflection. Then we extract types by static analyzers, bind types to dynamic-extracted APIs, and design inference rules to detect type changes. To address flexible argument passing, we include parameter kinds in our model, and match parameters by kinds in our diff algorithm. Taking two changes which only AexPy detected from our datasets as example, diffsync [40] changes the return type of `Diff.action()` from `str` to `Enum` [41], and Flask [42] removes a candidate keyword parameter `encoding` in `json()`, noted in its changelog [43].

2) More Comprehensive Change Detection: AexPy classifies and detects changes systematically, and then grades changes for practicality, which helps to cover more changes. To address complex API references, we use dynamic import and reflection to find precise API aliases, and design the pairing algorithm to detect alias-related changes. To address fake private members, we find and diff accessible APIs, and then use API aliases to filter fake private members more precisely by a customizable convention and grade their changes to low level to reduce disturbance in a practical way. For example, Scrapy [44] removes the constructor defined in `FileDownloadHandler`, and uses its parent's constructor, which causes a parameter `settings` removal, noted in its changelog [45]. The entry pairing helps AexPy to detect this hidden change. Polyaxon [46] uses a fake private instance attribute `XAxis._gridOnMajor` in upstream matplotlib [47], and matplotlib removes this attribute in version 3.3.3, causing polyaxon crashed [48]. AexPy detects this change as a low breaking change while the other tools ignore it.

3) More Robust Analysis: AexPy accesses APIs like a client and isolates importing exceptions to reduce runtime failures and improve robustness and applicability.

B. Limitations

We investigate causes of the 28 false positives in Section V-B, and discuss limitations of our approach.

1) Imprecise API Model: We design our Python-specific API model to cover main Python features and important API changes, while the model can be further refined to get more precise API descriptions and change patterns. For example, we treat generators, and asynchronous functions as normal functions, and cover their specific changes by ChangeReturnType, which can be refined according to their specific behaviors for more precise change patterns. In addition, our type model covers frequently-used types, and can be extended, e.g., bound type variables in generic types, to cover more type changes.

2) Imprecise Breaking Levels: Considering the complexity of API change impact in Python, we introduce heuristic breaking grading according to our knowledge from related issues and existing studies, for a trade-off between soundness and practicality, but the breaking severity may be subjective for different developers. Taking fake private members as an example, although AexPy follows the widely-used convention and grades their changes to low level, developers would have different naming strategies and compatibility guarantees. In practice, this can be fixed by developer-customized filtering strategies for APIs and changes.

3) Imprecise API Extraction: The limitations on dynamic reflection and static analysis cause imprecise APIs extraction, resulting in false positives and false negatives. There are still some cases which are not well-covered because of the various runtime behaviors. For example, developers can import modules conditionally, hook member resolving, or change members by the builtin mechanisms, such as `setattr` [49]. For functions, platform-dependent or native wrapper packages, e.g., NumPy [17], usually consist of compiled C code, which lack metadata like annotations and cannot be analyzed by a Python static analyzer. For parameters, decorators can hide parameters of the wrapped functions, and arguments for the `VarKeyword` parameter can be modified in batch. For types, if there are no type annotations, Mypy gains little information and AexPy works conservatively. Although we have addressed some of these problems by our hybrid analysis, it is still difficult to cover such flexible behaviors adequately.

C. Threats to Validity

We collect known breaking changes from GitHub issues searched by self-defined keywords, which may limit the diversity and representativeness of the collected issues. To mitigate the threat, we introduce changelogs of popular packages as another data source. As the result, the collected changes fit with the intuition that parameter changes are the most common and distribute in different categories, which demonstrates the generality of our change dataset. The selected packages distribute in different sizes (from 0.5k to 440k LOCs with average 36k), and popularity (from 8 to 64k stars on GitHub with average 9.8k), which avoids biased selection.

Our evaluation involves manual efforts, which might be subjective and biased. To reduce the threat, in the experiment on known changes, we judge AexPy more strictly than the other tools to get a conservative result. Specifically, AexPy needs to find the exact change to get a positive point, while other tools only needs to report a closely related change. In the experiment on unknown changes, we generate code snippets automatically for simple changes, e.g., RemoveModule, and manually for complex changes, to trigger them. We provide the snippets in reported issues as a minimal reproduction.

VII. Related Works

A. Finding Breaking Changes & Their Impacts

Breaking changes are common across programming languages and ecosystems [5], [50], [51]. Zhang *et al.* [4] studied six popular Python frameworks and found that more than 40% API changes are breaking. Mezzetti *et al.* [50] studied that at least 5% of JavaScript packages have experienced a breaking change in a non-major update. Brito *et al.* [52] studied the reasons Java developers intentionally break APIs. Many developers want to adopt the semantic versioning strategy, but do not trust that their dependencies adhere the guidelines [53].

For static languages, there are numerous tools that help library developers to detect breaking changes [54], because the explicitly typed APIs and strict grammars make it much easier. UMLDiff [55] uses reverse engineering on class UML models to detect structure changes in Java classes. APIDiff [56] parses Java source code to extract APIs, and classifies breaking changes cooperated with RefDiff [57]. Revapi [58], SigTest [59], Clirr [60], and japicmp [61] analyze Java APIs and track API changes. DeBBI [62] detects backward behavioral incompatibilities between Java libraries and client projects. For C/C++ libraries, abi-compliance-checker [63] extracts the symbol table and the virtual function table in object files to check binary compatibility and source compatibility. Besides compiled binaries, Ponomarenko *et al.* [64] obtained function signatures and type definitions from header files to detect a broad spectrum of backward compatibility problems.

In recent years, approaches about breaking change detection of dynamic languages have been proposed. Pidiff [9] is a checker for semantic versioning on Python packages, and it compares API information recursively on APIs' structure to detects changes. Zhang *et al.* [4] extracted APIs from Python frameworks by reflection to detect class, parameter, and attribute changes. They implemented a tool PyCompat using the manually filtered changes to detect misuses of changed APIs in client code. PyCT [65] extracts fine-grained code changes from the commit history of open-source Python projects. PyCT [65] and PYSCAN [14] give some evidences about the usage of dynamic language features. PyDFix [66] and V2 [67] detect breaking changes causing client build errors to fix build environments. Mezzetti *et al.* [50] proposed NoRegrets+ based on type regression testing to find type-related breaking changes in Node.js libraries. They applied dynamic analysis, leveraging automatic test suites, learned models of library interfaces and compared the models before and after an update. To our knowledge, existing syntax-based diff tools [68] cannot handle inheritance and API aliases well, and other languages' API diff tools are not well-suited for Python. In comparison, AexPy focuses on Python, covers important Python features and considers more change patterns, especially type incompatibility changes in Python.

B. Python API Extraction & Analysis

Besides pidiff and PyCompat, there are other studies extracting Python package APIs for different purposes. PyCRE [69] detects module and attribute APIs by dynamic importing and identifies environment dependencies for Python scripts according to the dependency knowledge base. SnifferDog [70] collects APIs of Python packages by static analysis, to restore execution environments for Jupyter Notebooks. Python API document generators extract document strings from source code, such as Epydoc [71], and the popular Sphinx [72], which supports the official Python documentation. In comparison, API extraction in our work covers more detailed APIs and produces more precise results because of our hybrid analysis.

Python static analysis tools help to analyze APIs. Static type checkers, such as Mypy [16], Pytype [73], Pyright [74], Pyre [75], infer types and detect type conflicts in source code according to annotations. Pylint [76] is a static code analysis tool for programming errors, such as undefined attributes. PyCG [77] analyzes assignment relations between program identifiers of functions statically to generate a call graph. Monat *et al.* [78] reused off-the-shelf analysis to process multilanguage analysis of Python programs with C extensions.

VIII. Conclusion

We propose a systematic approach to detect API breaking changes in Python packages. Via modeling package APIs, classifying and grading breaking changes, we address the challenges about dynamic features and multiple API references by a hybrid extraction method, and address the challenges about argument passing and fuzzy public scope by a constraint-based change detection and grading method. Compared to existing approaches, experiments on known breaking changes show our prototype tool, AexPy, has a high recall, strong robustness, and comparable time performance. On 43 real-world packages' latest versions, AexPy detects documented and undocumented breaking changes with a high precision, and we have received 31 confirmations on reported changes, which demonstrates the effectiveness and practicality of our approach.

In future works, we plan to enhance type analysis, cover more features to improve detection, and make AexPy more useful. We plan to make the reports more clear and easier to be used by developers, e.g., grouping changes by their root cause. We also plan to study more applications of the built knowledge base about APIs and changes, e.g., API documents, package dependencies, change impacts, and related patches.

References

[1] Python Software Foundation. (2022, May) Pypi · the python package index. [Online]. Available: https://pypi.org

[2] D. Dig and R. E. Johnson, "How do apis evolve? A story of refactoring," *J. Softw. Maintenance Res. Pract.*, vol. 18, no. 2, pp. 83–107, 2006. [Online]. Available: https://doi.org/10.1002/smr.328

[3] M. Reddy, *API Design for C++*. Elsevier, 2011.

[4] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do python framework apis evolve? an exploratory study," in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, and M. Zhou, Eds. IEEE, 2020, pp. 81–92. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054800

[5] L. Xavier, A. Brito, A. C. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, M. Pinzger, G. Bavota, and A. Marcus, Eds. IEEE Computer Society, 2017, pp. 138–147. [Online]. Available: https://doi.org/10.1109/SANER.2017.7884616

[6] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, S. Demeyer, D. W. Binkley, and F. Ricca, Eds. IEEE Computer Society, 2014, pp. 64–73. [Online]. Available: https://doi.org/10.1109/CSMR-WCRE.2014.6747226

[7] Z. Chen, Y. Li, B. Chen, W. Ma, L. Chen, and B. Xu, "An empirical study on dynamic typing related practices in python systems," in *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 83–93. [Online]. Available: https://doi.org/10.1145/3387904.3389253

[8] Python Software Foundation. (2022, May) Glossary. [Online]. Available: https://docs.python.org/3/glossary.html

[9] R. McGovern. (2022, May) rohanpm/pidiff: The python interface diff tool. [Online]. Available: https://github.com/rohanpm/pidiff

[10] Stack Overflow. (2022, May) Stack overflow developer survey 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020/

[11] JetBrains. (2022, May) The state of developer ecosystem in 2020 infographic - jetbrains: Developer tools for professionals and teams. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2020/

[12] Tiobe. (2022, May) index - tiobe - the software quality company. [Online]. Available: https://www.tiobe.com/tiobe-index/

[13] P. Carbonnelle. (2022, May) Pypl popularity of programming language index. [Online]. Available: https://pypl.github.io/PYPL.html

[14] Y. Peng, Y. Zhang, and M. Hu, "An empirical study for common language features used in python projects," in *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 2021, pp. 24–35. [Online]. Available: https://doi.org/10.1109/SANER50967.2021.00012

[15] Python Software Foundation. (2022, May) python/cpython: The python programming language. [Online]. Available: https://github.com/python/cpython

[16] The Mypy Project. (2022, May) mypy - optional static typing for python. [Online]. Available: http://www.mypy-lang.org/

[17] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, p. 357–362, 2020.

[18] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[19] Python Software Foundation. (2022, May) 6. expressions. [Online]. Available: https://docs.python.org/3/reference/expressions.html#private-name-mangling

[20] Cpp Reference. (2022, May) Variadic functions - cppreference.com. [Online]. Available: https://en.cppreference.com/w/cpp/utility/variadic

[21] Oracle. (2022, May) Passing information to a method or a constructor (the java™ tutorials - learning the java language - classes and objects). [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html

[22] T. Preston-Werner. (2022, May) Semantic versioning 2.0.0 - semantic versioning. [Online]. Available: https://semver.org/

[23] Python Packaging Authority. (2022, May) pip documentation v22.0.4. [Online]. Available: https://pip.pypa.io/en/stable/

[24] Python Software Foundation. (2022, May) abc abstract base classes. [Online]. Available: https://docs.python.org/3/library/abc.html

[25] Python Software Foundation. (2022, May) 9. classes. [Online]. Available: https://docs.python.org/3/tutorial/classes.html

[26] Python Software Foundation. (2022, May) 5. the import system. [Online]. Available: https://docs.python.org/3/reference/import.html

[27] Python Software Foundation. (2022, May) importlib. [Online]. Available: https://docs.python.org/3/library/importlib.html

[28] Python Software Foundation. (2022, May) pkgutil. [Online]. Available: https://docs.python.org/3/library/pkgutil.html

[29] Python Software Foundation. (2022, May) inspect. [Online]. Available: https://docs.python.org/3/library/inspect.html

[30] Python Software Foundation. (2022, May) ast. [Online]. Available: https://docs.python.org/3/library/ast.html

[31] H. Chu. (2022, May) sqlab-sustech/pycompat. [Online]. Available: https://github.com/sqlab-sustech/PyCompat

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[33] BentoML. (2022, May) Bentoml: Simplify model deployment. [Online]. Available: https://www.bentoml.com/

[34] L. Bourneuf. (2022, May) Aluriak/clyngor: Handy python wrapper around potassco's clingo asp solver. [Online]. Available: https://github.com/Aluriak/clyngor

[35] Microsoft. (2022, May) microsoft/appcenter-rest-python: A minimal python wrapper around the app center rest api. [Online]. Available: https://github.com/microsoft/appcenter-rest-python

[36] MagicStack, Inc. (2022, May) Magicstack/asyncpg: A fast postgresql database client library for python/asyncio. [Online]. Available: https://github.com/MagicStack/asyncpg

[37] B. Darnell. (2022, May) tornadoweb/tornado: Tornado is a python web framework and asynchronous networking library, originally developed at friendfeed. [Online]. Available: https://github.com/tornadoweb/tornado

[38] Python Software Foundation. (2022, May) Python release python 3.6.15 - python.org. [Online]. Available: https://www.python.org/downloads/release/python-3615/

[39] Python Software Foundation. (2022, May) timeit. [Online]. Available: https://docs.python.org/3/library/timeit.html

[40] D. Garros. (2022, May) networktocode/diffsync: A utility library for comparing and synchronizing different datasets. [Online]. Available: https://github.com/networktocode/diffsync

[41] G. Matthews. (2022, May) Breaking api change in diffsync 1.4.0 · issue #101 · networktocode/diffsync · github. [Online]. Available: https://github.com/networktocode/diffsync/issues/101

[42] D. Lord. (2022, May) pallets/flask: The python micro framework for building web applications. [Online]. Available: https://github.com/pallets/flask

[43] D. Lord. (2022, May) Changes - flask documentation (2.1.x). [Online]. Available: https://flask.palletsprojects.com/en/2.1.x/changes/

[44] Zyte. (2022, May) Scrapy - a fast and powerful scraping and web crawling framework. [Online]. Available: https://scrapy.org/

[45] E. Lacuesta. (2022, May) Release notes - scrapy 2.6.1 documentation. [Online]. Available: https://docs.scrapy.org/en/latest/news.html#scrapy-2-0-0-2020-03-03

[46] Polyaxon, Inc. (2022, May) polyaxon/polyaxon: Mlops tools for managing & orchestrating the machine learning lifecycle. [Online]. Available: https://github.com/polyaxon/polyaxon

[47] J. D. Hunter and M. Droettboom. (2022, May) matplotlib/matplotlib: matplotlib: plotting with python. [Online]. Available: https://github.com/matplotlib/matplotlib

[48] M. Mourafiq. (2022, May) New matplotlib breaking change · issue #1172 · polyaxon/polyaxon · github. [Online]. Available: https://github.com/polyaxon/polyaxon/issues/1172

[49] Python Software Foundation. (2022, May) Built-in functions. [Online]. Available: https://docs.python.org/3/library/functions.html#setattr

[50] G. Mezzetti, A. Møller, and M. T. Torp, "Type regression testing to detect breaking changes in node.js libraries," in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, ser. LIPIcs, T. D. Millstein, Ed., vol. 109. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 7:1–7:24. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2018.7

[51] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *J. Syst. Softw.*, vol. 129, pp. 140–158, 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2016.04.008

[52] A. Brito, L. Xavier, A. C. Hora, and M. T. Valente, "Why and how java developers break apis," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 255–265. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330214

[53] C. Bogart, C. Kästner, and J. D. Herbsleb, "When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies," in *30th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE Computer Society, 2015, pp. 86–89. [Online]. Available: https://doi.org/10.1109/ASEW.2015.21

[54] K. Jezek and J. Dietrich, "API evolution and compatibility: A data corpus and tool evaluation," *J. Object Technol.*, vol. 16, no. 4, pp. 2:1–23, 2017. [Online]. Available: https://doi.org/10.5381/jot.2017.16.4.a2

[55] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds. ACM, 2005, pp. 54–65. [Online]. Available: https://doi.org/10.1145/1101908.1101919

[56] A. Brito, L. Xavier, A. C. Hora, and M. T. Valente, "Apidiff: Detecting API breaking changes," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 507–511. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330249

[57] D. Silva and M. T. Valente, "Refdiff: detecting refactorings in version histories," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, J. M. González-Barahona, A. Hindle, and L. Tan, Eds. IEEE Computer Society, 2017, pp. 269–279. [Online]. Available: https://doi.org/10.1109/MSR.2017.14

[58] L. Krejci. (2022, May) Revapi :: Revapi. [Online]. Available: https://revapi.org/revapi-site/main/index.html

[59] Oracle. (2022, May) sigtest - sigtest - openjdk wiki. [Online]. Available: https://wiki.openjdk.java.net/display/CodeTools/SigTest

[60] L. Kühne. (2022, May) Clirr - clirr. [Online]. Available: http://clirr.sourceforge.net/

[61] M. Mois. (2022, May) siom79/japicmp: Comparison of two versions of a jar archive. [Online]. Available: https://github.com/siom79/japicmp

[62] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 112–124. [Online]. Available: https://doi.org/10.1145/3377811.3380436

[63] A. Ponomarenko and V. Rubanov, "Automatic backward compatibility analysis of software component binary interfaces," in *2011 IEEE International Conference on Computer Science and Automation Engineering*, vol. 3, 2011, pp. 167–173.

[64] A. V. Ponomarenko and V. V. Rubanov, "Backward compatibility of software interfaces: Steps towards automatic verification," *Program. Comput. Softw.*, vol. 38, no. 5, pp. 257–267, 2012. [Online]. Available: https://doi.org/10.1134/S0361768812050052

[65] W. Lin, Z. Chen, W. Ma, L. Chen, L. Xu, and B. Xu, "An empirical study on the characteristics of python fine-grained source code change types," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 2016, pp. 188–199. [Online]. Available: https://doi.org/10.1109/ICSME.2016.25

[66] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for python build reproducibility," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 439–451. [Online]. Available: https://doi.org/10.1145/3460319.3464797

[67] E. Horton and C. Parnin, "V2: fast detection of configuration drift in python," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 477–488. [Online]. Available: https://doi.org/10.1109/ASE.2019.00052

[68] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[69] W. Cheng, X. Zhu, and W. Hu, "Conflict-aware inference of python compatible runtime environments with domain knowledge graph," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 451–461. [Online]. Available: https://doi.org/10.1145/3510003.3510078

[70] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of jupyter notebooks," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1622–1633. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00144

[71] E. Loper. (2022, May) Epydoc: Api documentation extraction in python. [Online]. Available: http://epydoc.sourceforge.net/

[72] G. Brandl and the Sphinx team. (2022, May) Overview - sphinx documentation. [Online]. Available: https://www.sphinx-doc.org/en/master/

[73] Google. (2022, May) google/pytype: A static type analyzer for python code. [Online]. Available: https://github.com/google/pytype

[74] Microsoft. (2022, May) microsoft/pyright: Static type checker for python. [Online]. Available: https://github.com/microsoft/pyright

[75] Facebook. (2022, May) Pyre - pyre. [Online]. Available: https://pyre-check.org/

[76] Logilab and PyCQA. (2022, May) Pylint 2.14.0-dev0 documentation. [Online]. Available: https://pylint.pycqa.org/en/latest/

[77] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "Pycg: Practical call graph generation in python," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1646–1657. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00146

[78] R. Monat, A. Ouadjaout, and A. Miné, "A multilanguage static analysis of python programs with native C extensions," in *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings*, ser. Lecture Notes in Computer Science, C. Dragoi, S. Mukherjee, and K. S. Namjoshi, Eds., vol. 12913. Springer, 2021, pp. 323–345. [Online]. Available: https://doi.org/10.1007/978-3-030-88806-0_16