Improving Test Case Generation for Python Native Libraries Through Constraints on Input Data Structures

Xin Zhang\textsuperscript{1,3}, Xutong Ma\textsuperscript{1,3}, Jiwei Yan\textsuperscript{2,3}, Baoquan Cui\textsuperscript{1,3}, Jun Yan\textsuperscript{1,2,3}, Jian Zhang\textsuperscript{1,3,†}

\textsuperscript{1} State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
\textsuperscript{2} Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
\textsuperscript{3} University of Chinese Academy of Sciences

Email: \{zhangxin19, maxt, yanjw, cuibq, yanjun, zj\}@ios.ac.cn

Abstract—Modern Python projects execute computational functions using native libraries and give Python interfaces to boost execution speed; hence, testing these libraries becomes critical to the project’s robustness. One challenge is that existing approaches use coverage to guide generation, but native libraries run as black boxes to Python code with no execution information. Another is that dynamic binary instrumentation reduces testing performance as it needs to monitor both native libraries and the Python virtual machine.

To address these challenges, in this paper, we propose an automated test case generation approach that works at the Python code layer. Our insight is that many path conditions in native libraries are for processing input data structures through interacting with the VM. In our approach, we instrument the Python Interpreter to monitor the interactions between native libraries and VM, derive constraints on the structures, and then use these constraints to guide test case generation. We implement our approach in a tool named PyCing and apply it to six widely-used Python projects. The experimental results reveal that with the structure constraint guidance, PyCing can cover more execution paths than existing test cases and state-of-the-art tools. Also, with the checkers in the testing framework Pytest, PyCing can identify segmentation faults in 10 Python interfaces and memory leaks in 9. Our instrumentation strategy also has an acceptable influence on testing efficiency.

Index Terms—Python native library testing, test case generation, structure constraints

I. INTRODUCTION

Benefitting from its flexible syntax and vast third-party libraries, Python has become the dominating programming language in many popular fields, including machine learning, scientific computation, and image processing.\textsuperscript{[1]} Projects develop calculating functions utilizing native libraries and give Python code with interfaces to fulfill the performance expectations in these fields. These native libraries are mainly written with statically typed languages such as C, C++, FORTRAN, and Cython\textsuperscript{[2]} and made available to Python code through dynamic link libraries. Many high-performance computing projects have developed their own native libraries, like NumPy\textsuperscript{[3]} and SciPy\textsuperscript{[4]} for scientific computing, and Sklearn\textsuperscript{[5]} for machine learning. In this paper, we refer to a Python interface provided by native libraries as a native method, and refer to the type and attributes of one Python object as the structures of the Python object.

Because data structures change across programming languages, defect detection mechanisms in Python, such as automatic memory management, cannot cover these libraries. While utilizing native libraries has several advantages, such as better execution performance and interface with the operating system, they may also result in fatal errors, such as type conflict\textsuperscript{[6]–[8]}, performance loss\textsuperscript{[9], [10]}, and dependency issues\textsuperscript{[11], [12]}. Besides, native libraries run on the operating system, not the Python virtual machine. Some defects in native libraries may cause VM crashes, like segmentation faults. The rising popularity of native libraries emphasizes the critical nature of their testing.

Fuzzing methods are effective in detecting bugs under the guidance of code coverage\textsuperscript{[13]–[15]}. They get code coverage by instrumenting the source code, which is not difficult in cases when just focusing on one programming language. However, they fail in instrumenting the native libraries in real-world Python projects since they cannot parse dynamic link libraries and are incompatible with the complicated compilation process. One option is dynamic binary instrumentation. Because the Python virtual machine is essential for running native libraries, this option needs to instrument both the VM and native libraries, which can significantly lower test efficiency. Moreover, invoking native libraries without a VM necessitates rewriting all VM-related functions, which is time-consuming and labor-intensive\textsuperscript{[16]}. Some approaches for other languages, such as Java, test native libraries through summaries\textsuperscript{[17]–[20]}. They employ statically analyzing C code to capture constraints in summaries, which they subsequently use to guide test case generation. Because Java and native libraries are both statically typed languages, the input data structures are known, so these approaches only need to focus on the input value. However, building summaries for Python native libraries can be challenging. On the one hand, native libraries in Python have multiple languages, including C, C++, FORTRAN, and Cython. Combining summaries for different languages is complex. On the other hand, Python does not limit or validate input
types to native methods due to the dynamically typed language feature. To provide the same functionality for input data of varying types and attributes, native libraries include lots of path conditions related to structures. Building summary necessitates both modeling functions for processing input data structures and dealing with the situation where native libraries call Python code. It poses considerable difficulties in producing effective summaries for testing Python native libraries.

Compared with other test scenarios, testing Python native libraries has two key challenges.

- **Lacking information about structure constraints in native libraries.** If there is no range of structures that native libraries can handle, it can be hard to improve test effectiveness by randomly generating input data with Python built-in types. Besides, going through all the common types and attributes will result in a combinatorial explosion.

- **Lacking execution information of native libraries.** Execution information is critical not just for test case generation but also for evaluation. Native libraries operate as black boxes for Python code, offering no execution information. Moreover, dynamic binary instrumentation will dramatically degrade test efficiency.

In this paper, we propose an automated test case generation approach guided by structure constraints in Python native libraries. Given the overhead of directly invoking native libraries, we choose to test native libraries from the Python code layer. Our insight is that many path conditions in native libraries are for input data structures, and they must process the structures using API functions provided by the Python Interpreter. Thus, we can track the calls to these APIs when executing the native libraries, and build structure constraints based on the API call logs. We chose to instrument these API functions to address the first challenge. In this way, we can retrieve the arguments and return values every time the native libraries use these functions, then utilize the outputs to build constraints on input data structures. To address the second challenge, as the structure constraints are part of path conditions in native libraries, they can reflect the number of paths explored by test cases. Our approach takes the source code of a project as input. We first retrieve native methods from the documentation. Then, we execute these methods to build structure constraints for existing input data. Following that, an example from the NumPy project will be used to demonstrate the significance of input data structures in testing native libraries.

### A. The Python Execution Environment

Python is an interpreted language with many interpreters available, including CPython written in C/C++, IntelPython developed by Intel, and JPython written in Java. Because CPython is the official version and is commonly used in the development, all the keywords Python Interpreter and interpreter in this paper refer to CPython.

The interpreter executes Python code on a stack-based virtual machine and saves the stack frame throughout execution using Python objects called PyFrame. Each PyFrame provides access to the execution context, including the current local and global variables, the line of executed code, and the location of the Python file. Users can acquire PyFrames by registering a tracer to the code. Many Python debug tools, like pdb, employ this tracer mechanism.

### B. Python Native Libraries

Developers can compile functions developed by other programming languages into dynamic link libraries, which can subsequently be invoked from Python code. To establish correlations between Python methods and native functions for analysis, some researchers analyze the source code of native libraries statically. Because developers can specify the names of Python interfaces in many ways, these approaches are inaccurate when applied to real-world projects.
A Python object is handled in the native libraries as a C struct PyObject, where types and attributes of the object correspond to the fields in the PyObject. Thus, in this paper, we regard all the operations on a PyObject in native libraries as operations on the attributes and types of the corresponding Python object.

To help native libraries manage Python objects and access the Python virtual machine, the interpreter provides a set of C/C++ functions named Python/C API [30]. These API functions enable native libraries to check types, extract attributes, allocate and free memory in the Python heap area, and throw exceptions to Python code.

C. A Motivating Example

The interpreter manages the lifetime of objects with a reference counting mechanism, relieving developers of the work of manually freeing objects and reducing the danger of memory leaks [31]. But this mechanism demands developers to manage memory manually while processing Python objects in native libraries. Thus, if the reference count of a Python object is increased in native libraries, the interpreter will not free it even though it is no longer in use [32, 33].

Figure 1 shows one of the C functions in the implementation of the native method numpy.power. This native method takes two Python objects as input and calculates their powers. The presented C function receives one PyObject and one double value as input and returns the power value through the second argument, where argument o2 points to the second argument of this native method. In line 4, this function uses a Python/C API function PyLong_Check to check if o2 is of type int. On line 9, the API function PyFloat_Check checks for the type float. API function PyIndex_Check at line 13 checks if o2 includes a member method __index__. Then at line 14, this function invoke the member method __index__ through API function PyObject_Index.

```c
static NPY_SCALARKIND is_scalar_with_conversion(PyObject *o2, double* out_exponent){
    ...
    if(PyLong_Check(o2)){
        return NPY_NOSCALAR;
    }
    ...
    else if(PyFloat_Check(o2)){
        PyObject* value = PyObject_Index(o2);
        ...
        return NPY_INTPOS_SCALAR;
    }
    ...
}
```

Fig. 1. One of C functions in the implementation of the native method numpy.power.

The API function PyObject_Index will raise the return value’s reference count to prevent this value from being recycled throughout the caller’s procedure. A leak will occur if the native method finishes without decreasing the reference count. To trigger this leak, the second argument of the native method must make the API functions on lines 4 and 9 return False and the API function on line 13 return True. Thus, the input data cannot inherit from the types int and float, and must have the method __index__.

An example of input data to trigger this leak is as follows, in which we construct a class and inherit it from type str, then add the needed method to this class from lines 2 to 4. In line 5, we instantiate this class and pass it a random string. However, such input data does not exist in the project’s test cases. Existing Python testing tools can only mutate values, and they cannot generate test case to trigger this leak when their seeds do not contain such structures.

III. Terminology

In this section, we will introduce some conceptions used in our approach.

A. Python object. A Python object can be described as a pair:

\[ \text{obj} = (\text{Type}, \text{Attrs}) \]

, where Type denotes the object’s type and Attrs is a collection of its attributes. One attribute can be a member method, a member variable, or an element in a collection type object. Each attribute is a Python object as well.

Constraints on type inheritance. Let set \( T \) represent all Python types, and \( \forall t_1, t_2 \in T, t_1 \preceq t_2 \) denotes that types \( t_1 \) and \( t_2 \) are the same, or that type \( t_1 \) inherits from type \( t_2 \).

If two sets \( A_{bt} \subseteq T \) and \( A_{nbt} \subseteq T \) satisfy \( \forall t_1 \in A_{bt}, t_2 \in A_{nbt}, \neg(t_1 \preceq t_2) \wedge \neg(t_2 \preceq t_1) \), we can define path constraints on the type inheritance of an object (Type constraints for short) as:

\[ \forall t_1 \in A_{bt}, t_2 \in A_{nbt}, (\text{obj}.\text{Type} \preceq t_1) \wedge \neg(\text{obj}.\text{Type} \preceq t_2) \]

Set \( A_{bt} \) represents all the types that an object should be or should inherit from, whereas the set \( A_{nbt} \) contains all the types that should not be or should not inherit from.

Constraints on attribute existence. Let set \( A \) represent all Python objects, and \( \exists a \in A, a \in \text{obj.attrs} \) indicates that Python object \( \text{obj} \) contains the attribute \( a \).

If there are two sets \( A_{bt} \subseteq A \) and \( A_{nbt} \subseteq A \), and they fulfill \( A_{bt} \cap A_{nbt} = \emptyset \), we can express the constraints on attribute existence of an object (attribute constraints for short) as:

\[ \forall a_1 \in A_{bt}, a_2 \in A_{nbt}, (a_1 \in \text{obj.attrs}) \wedge (a_2 \notin \text{obj.attrs}) \]

Set \( A_{bt} \) represents all of the attributes that an object needs to contain, whereas the set \( A_{nbt} \) represents the attributes that should not possess.
Structure Constraint. Every Python object can be mapped to a quad \((T_{i\_bt}, T_{i\_nbt}, A_{i\_bt}, A_{i\_nbt})\) based on the definitions of constraints on type inheritance and attribute existence.

Because distinct attributes in a Python object correspond to different quads, one structure constraint of a Python object (SC for short) is a conjunctive normal form of a set of mappings:

\[
SC_{\text{object}} = \text{map}_1 \land \text{map}_2 \land \cdots \land \text{map}_i
\]

. Each \(\text{map}_i\) is a mapping connections between an object and a quad: \(\text{map}_i = \text{obj}_i \mapsto (T_{i\_bt}, T_{i\_nbt}, A_{i\_bt}, A_{i\_nbt})\), where \(\text{obj}_i\) can be the Python object \text{object} or one of its attributes.

IV. The Proposed Approach

Figure 2 depicts the workflow of our approach, encompassing two phases: Static analysis and Structure constraint-guided Testing. In the first phase, we construct a value seed to house all of the values used in the projects. Also, we instrument the Python/C API to capture execution information. The second stage uses a loop to iteratively explore the execution paths, stopping when no new \(SC\) appears. In each loop, we construct \(SCs\) for existing input data based on API call logs. Guided by the \(SCs\), our approach generates new Python objects as input data for the next loop.

A. Static analysis

This section illustrates how we collect project values and instrument the Python Interpreter.

1) Value Set Construction: In addition to \(SCs\), there are constraints on input values, such as branch conditions that compare the input value to a certain value. However, extracting value constraints from native libraries exactly can be problematic. On the one hand, extracting from source code requires complete call function graphs and path-sensitive interprocedural analysis, which are challenging to achieve with present static analysis tools. On the other hand, it is tricky to relate the instructions to input data if monitoring the instructions. We cannot find vulnerabilities lying in paths if we generate values randomly.

To circumvent value constraints, we apply a lightweight strategy, where we collect all of the values used in the projects and employ them in the generating phase. We can produce values that satisfy such branch conditions in this manner.

In detail, we collect values from two aspects: the literals in the source code and objects passed to native methods in the projects. We first scan Python and native library source code and store all the literals. Then, to collect objects passed to the native methods, we utilize the debug tool \(pdb\) to insert breakpoints to each native method call. In this way, we can stop at each callsite and dump the input data while running the test suites in the projects. We build the value set \(\text{ValueSet}\) to hold all of the data we collect.

2) Python Interpreter Instrumentation: As mentioned in Section 1, native libraries serve as black boxes for Python code. In our approach, we utilize Python/C APIs to monitor native library execution. We instrument these API functions at the source code level and recompile the interpreter.

The interpreter supports roughly a thousand API functions, with parts implemented using function-like macros. Because not all the functions are for processing input data, we manually review the documentation and filter two categories of functions: one for type checking and another for attribute extracting. We utilize two separate instrumentation strategies to output in the same format while their implementation is not the same.

- **API functions for type checking.** This category of functions takes a reference and returns a boolean value indicating whether the referred Python object is of the given type. We collect 50 functions from the documentation and insert code to output the reference, the verified type, and the result.

- **API functions for attribute extracting.** This category of API functions receives two pointers as input, one to a Python object and the other to a key, and returns the attribute if it exists or NULL if it does not. We gather 14 functions and insert code to print the two pointers and the return value.

Using the output of the instrumented code, we can abstract each API function call into a 4-tuple: \(\text{API}(o, act, val_{act}, ret)\), where \(o\) stands for the input data, \(act\) for the functionality, such as type checking or attribute extracting, \(val_{act}\) for the argument, and \(ret\) for the return value.

Example for instrumentation: From lines 2 to 6 in the following code, we give the type-checking API function \text{PyDict\_Check} implementation, where we instrument code at line 4 to output the checked type \(\text{dict}\), the checking result, and the argument. Besides, we present the simplified implementation of API function \text{PyMapping\_GetItemString} from lines 8 to 13. This function is for extracting attributes from the input data. Lines 10-11 are the instrumented code that prints the attribute \(r\), the key \(key\), and the input data \(o\).

```
1 // Objects/dictobject.c
2 int PyDict\_Check(void *op){
3 int result = PyType\_FastSubclass(
4 Py\_TYPE(op),
5 Py\_TFFLAGS\_DICT\_SUBCLASS);
6 printf("%p, Chk, dict, %d \n", op, result);
7 return result;
8 } // Objects/abstract.c
```
B. Execution

In this part, we construct SCs for existing input data. On a high level, we use an instrumented Python Interpreter to call native methods. Then, using the interpreter output, we gather associated API function call logs and construct the SCs.

We will receive a series of API function calls after executing. However, we cannot match the C structures processed by the API functions with the input data just based on logs. This is because native libraries can also employ API functions to process locally created objects or one of the arguments passed to the native method. Without knowing which calls are for input data, we will construct plenty of constraints that are not related to the input data, limiting our exploration’s efficiency. Consider the code shown in Figure 1, API function calls to object o2 are for this native method’s second argument. It will be hard to explore paths in this C function if we wrongly assume that these calls are for processing the first argument.

Thus, we scan the logs twice to build the SCs. We gather type and attribute constraints on input data for the first time. The second time, we collect constraints on attributes of input data to fulfill the SCs.

Because all Python objects are stored in Python heap space, an object’s memory address will not change when transferred from Python code to native libraries. Thus, we can filter the function calls that process the input data using the memory address. Before invoking a native method, we first retrieve the memory addresses for the input data at the Python code layer, then append these addresses to the SCs. If the address in one call appears in SCs, it indicates that this function is for processing the input data. We can establish the type and attribute constraints for the input data in the first scan.

However, using memory address matching to create constraints for attributes is inefficient. Because an attribute is also a Python object, the Attrs of a Python object is tree-structured. Acquisition of memory addresses for all the attributes can cause recursive traversal and consumes a lot of space.

As the addresses of input data were added to the SCs in the first step, we filter the calls of attribute extraction and add the address of the extracted attribute to the SC. We can then use the extracted attribute to filter API function calls and progressively generate SCs for the input data.

The two guidelines in Table I define the building process. We present example statements, API function call patterns, and updating operations for each rule. When we process an API function for type checking, such as the call patterns in rule R1, we adjust the type constraints of obj based on the return result if the obj is recorded in the SC. Specifically, if the return value is true, we add the valact to the set Tobj through operation op1.1, or to the set Tnbt by operation op1.2.

When dealing with an API function for attribute extracting, as described in rule R2, we modify the attribute constraints if the obj is in the SC. Firstly, we add the attribute with an empty set to the SCs using the operation op2.1. Then, similar to the operations in rule R1, if the return result is not null, indicating that the object obj possesses the attribute, we will add it to the set Aobj in the operation op2.2. Otherwise, we shall add the attribute to the set Aobj as this object does not contain it.

Example for SC construction. We pass in two Python objects [1,2,3] and "abc" for the native method numpy.power, and we can retrieve call logs when executing the C function in Figure 1 as the follows.

```
API(o2, ‘Chk’, Long, False)
API(o2, ‘Chk’, Float, False)
API(o2, ‘Get’, ‘__index__', False)
```

We add the object o2 to the SCs as o1 before calling this native method. When dealing with the calls on lines 1 and 2, we use operation op1.2 to update the SC: o2 \(\rightarrow\) \(\{\text{float, long}\}\), \(A_{o2\_nbt} = \{\text{__index__}\}\). For the call on line 3, we first add the attribute to SCs with the operation op2.3. After line 3, SCs are:

\[
o2 \rightarrow \left(T_{o2\_nbt} = \{\text{float, long}\}, A_{o2\_nbt} = \{\text{__index__}\}\right) \land A_{o1\_index} \rightarrow (').
\]

C. Reverse

The purpose of reverse is to flip the result of the current branch condition and then create a new Python object to fulfill the opposite path conditions. We select to build the original SC and its reverses simultaneously.

As indicated in Section IV-B, we traverse the call logs and use the operations in Table II to create the original SC. Before we update the original SC for each operation, we produce one reversed SC by making a copy and adding the valact to the opposite set. Thus, a reversed SC can represent the constraints from the native method entrance to the current branch condition, and another path to the current branch condition.

The specifics of the reverse process are present in Algorithm II. According to the API call, the get_operates function on line 4 obtains operations from Table II. From lines 5 through 16, we go over each operation. Line 6 generates a copy before updating the original SC, while function apply in line 7 changes the original SC depending on the operation. Lines 8-15 update the set against the operation in the copied SC. If this reversed SC does not appear before, we include it in the result mut_list.

Because each rule only increases one constraint, and we create one reversed SC for each rule, the number of reversed SC does not exceed the number of constraints in the original SC.

An example for the reverse. Consider the API function calls in the example for SC construction. When we process the call on line 1, we add long to the T\_nbt in the original SC, while building a reverse SC: \(o1 \rightarrow T_{o1\_nbt} = \{\text{long}\}\). After we process the calls on lines 2 and 3, we can produce two reversed SCs. The first is: \(o1 \rightarrow T_{o1\_nbt} =\)
Algorithm 1: Reverse on one SC

Input: API function call logs api_list
Output: One original SC cons, a set of reversed SCs mut_list
1 mut_list ← list()
2 cons ← dict()
3 for api ∈ api_list do
4     ope_list ← get_operates(api)
5         for ope ∈ ope_list do
6             mut_cons ← copy(cons)
7                 apply(cons, ope)
8                 if ope is OP1.1 then
9                     update(mut_cons, ope, T_nbt)
10                else if ope is OP1.2 then
11                     update(mut_cons, ope, T_bt)
12                else if ope is OP2.2 then
13                     update(mut_cons, ope, A_nbt)
14                else if ope is OP2.3 then
15                     update(mut_cons, ope, A_BT)
16                else
17                 apply(mut_cons, op2.1)
18             mut_list.add(mut_cons)
19 return cons, mut_list

{float}, T_o1_nbt = long,
and the second is: o1.__index__ → ()
∧ o1 → (T_o1_nbt = {long, float}, A_o1_BT = {__index__}).

D. Generation

Algorithm 2 depicts the generation process. From a high level, we first produce a piece of class definition code that causes the class to inherit from the types in the T_BT but not in the T_nbt. Then we can instantiate this class and modify the attributes based on A_BT and A_nbt.

Because SC imposes constraints on both the object and its attributes, we must first generate attributes before producing the object. In line 1, we topologically order all keys in SC based on their containment relationship. For example, if object o1 appears in A_BT or A_nbt of object o2, we consider o2 contains o1 and sort o2 behind o1. We use a map attrimap in line 2 to store all the attributes we have produced.

The two language features dynamic attribute and dynamic type cannot be utilized to generate objects since the API functions only check for types and member methods based on the class definition code. Thus, each object generation is completed in three phases. First, we construct a class that inherits needed types from lines 5-10. Then, in line 11, following the attribute constraints, we change the member variables and methods in the class definition code. Finally, if the object is of collection types, as demonstrated in lines 12 to 13, we instantiate this class and adjust the elements depending on attribute constraints.

We compute the types that the class needs to inherit from...
lines 5 to 9, where we subtract the types in the set $cons.T_{nbt}$ from the Python built-in type set $builtins$, then intersect the result with $cons.T_d$. If the set $cons.T_d$ is empty, we do not conduct the intersection operation since the object is not required to inherit from a specific type. In line 10, we produce a piece of class definition code to make the self-defined class inherits from the types in the set $inherits$ in the function build_class.

We change member methods and variables in the self-defined class depending on $cons.A_{bt}$ and $cons.A_{nbt}$ in the function up_class on line 11. We extract a member method or variable from the attriset and add it to the class definition code if this attribute is required to exist. For example, if the set $cons.A_{bt}$ is $\{\_\_index\_\_\}$, we can add a method named __index__ and a member variable as this method’s return value to the class definition code. The member variable’s name can be generated randomly while its value comes from the value set ValueSet.

We instantiate this class on line 12. The object obj is then updated at line 13 via the function up_object if it is of collection types and there exist constraints on its elements, where we add required attributes to this object.

In line 14, we append the obj object into the attriset. The input object we produced for the $SC$ is the last element added to the attriset after finishing the loop from lines 4 to 14.

An example for the generation. The following $SC$ is used as an example to demonstrate the generating process:

\[
o1 \mapsto (T_{o1_{bt}} = \{dict\}, A_{o1_{bt}} = \{\_\_index\_\_, names\})
\land o1._\_index_\_ \mapsto (T_{o1._\_index_\_bt} = \{float\})
\land o1.names \mapsto (T_{o1.names_{bt}} = \{list\}).
\]

We start by sorting the three keys in the $SC$, and we can retrieve gen_set as $o1._\_index_\_, o1.names,$ and $o1$, which implies we need to produce two attributes before building object $o1$.

```python
class self_class(dict):
    returnval = 1.0
3. def __index__(self):
4.     return self.returnval
5. obj1 = self_class()
6. obj1['names'] = []
```

We produce a method __index__ for the key $o1._\_index_\_$. Because $T_{o1._\_index_\_bt}$ contains $float$, this method must return an $float$ value. Thus, we generate a member variable of type $float$ and return it in this method.

If the attribute o.names exists in the attriset, we can utilize the matching value to produce a key-value pair and add it to the object obj. Or we can randomly generate a value to produce the key-value pair. We generate an empty list for constraints on o.names, as required by $T_{o1.names_{bt}}$.

For constraints on o, we first create a class that inherits from type dict based on its $T_d$, then add member method and variable to the class definition code (lines 2-3), and last instantiate the class and insert the required key-value pair (lines 4-5). After this, object obj1 is the object we generate for this $SC$.

V. Evaluation

A. Evaluation Setup

We instrument Python 3.8.5 and develop PyCing with Python 3. All of the code and data are open to the public.

We traverse a list of the top 50 Python projects on GitHub by star rating. As the target of PyCing is testing Python native libraries, We choose projects based on the three criteria listed below: (1) incorporate native libraries, (2) do not use lazy evaluation methods since we cannot guarantee that a native method runs or only returns a symbol, and (3) have bug reports about native libraries. We remain six projects from the fifty for our experiments.

In contrast to statically typed languages that determine the method types and definitions during compilation, obtaining native methods can be challenging as Python only determines the method definitions when executing. Besides, no type inference tools support method types. Thus, we opt to collect native methods from two sources: existing studies and project documents. Several studies have analyzed native libraries for some Python projects [9], [28], [36]. We can glean native methods from their publicly available experimental data. Besides, we review the project documentation for all available methods and then verify their implementation to collect native methods.

1) Research Questions: We propose three research questions, and by answering them, we will present the results of our experiment.

• RQ1. What is the efficiency of PyCing?
• RQ2. What is the test effectiveness of PyCing in real-world Python projects?
• RQ3. What is the test effectiveness of PyCing when compared to other fuzzers?

In answer to the first question, we evaluate the impact of PyCing instrumentation and settings on test efficiency. To answer the second question, we collect modern Python projects from GitHub and compare test cases in these projects with PyCing. Because existing tools cannot obtain code coverage from these projects, we compare PyCing against two state-of-the-art Python fuzzers on a set of handwritten benchmarks to answer the third question.

2) Evaluation Criteria: As indicated in Section 1, native libraries do not provide code coverage. Thus, we evaluate testing effectiveness from 2 factors: the number of $SC$s and the types of API functions. When the number of test cases is identical, the number of $SC$s can indicate the number of examined execution paths, while the types of API functions can reflect the structure coverage that native libraries can handle.

3) Configuration: All experiments run on a PC equipped with an Intel Core i7 3.60GHz CPU and 32 GByte RAM. The GitHub projects are built on Ubuntu 18 using Python 3.8.5. We establish a time restriction of 24 hours for each project.

[https://anonymous.4open.science/r/pycing-present-6BC7](https://anonymous.4open.science/r/pycing-present-6BC7)
determined by the comparative experiments specified in papers or extra materials of existing Python testing tools.

B. Performance of PyCing (RQ1)

Two aspects will influence the efficiency of PyCing: instrumentation to the interpreter and configurations in the generation phase of PyCing.

We first assess the influence of instrumentation on execution performance. The instrumented interpreter is compared to a standard interpreter and a commonly used dynamic binary instrumentation framework pin [37]. We utilize the test cases in these projects as benchmarks since they can represent common usage, and we measure the execution time with the testing framework Pytest. Because Pytest supports multi-processing, we can run test cases with different numbers of processes. We can directly invoke Pytest as a Python module for the two interpreters. For the framework pin, we use the default pintool to monitor at the function level and execute on the standard interpreter inside the framework. Project CPython is excluded from this experiment as many native methods in this project deal with suspended execution or multi-processing, both of which have a substantial impact on the running time count.

Table II shows the number of test cases and execution times under a single process (1), two processes (2), and four processes (4). If one test case runs for more than an hour or the operating system terminates the Python virtual machine, we will mark it as NA for this configuration. According to the data, our instrumentation strategy has an acceptable influence on execution performance, while dynamic binary instrumentation increases the time by 5x to 10x. Increasing the number of processes might lower the time under dynamic binary instrumentation. However, testing one native method requires several rounds of sequential execution, and increasing the number of processes has little effect on test efficiency. In the SciPy project, the pin fails because of conflicts when instrumenting one binary file under the multiprocess mode, or it takes too much memory for storing interim results, causing the OS to terminate it.

| Project     | Tests | Standard (Seconds) | Instrumented (Seconds) | Pin (Seconds) |
|-------------|-------|--------------------|------------------------|---------------|
| Matplotlib  | 8K    | 404                | 434                    | 2,762         |
|             |       | 243                | 297                    | 1,489         |
|             |       | 160                | 183                    | 991           |
| NumPy       | 17K   | 151                | 179                    | 1,034         |
|             |       | 107                | 121                    | 667           |
|             |       | 73                 | 89                     | 446           |
| Pandas      | 132K  | 1,644              | 1,992                  | 9,653         |
|             |       | 969                | 1,171                  | 5,476 NA      |
|             |       | 787                | 893                    | NA            |
| Pillow      | 2K    | 49                 | 52                     | 350           |
|             |       | 25                 | 29                     | 217          |
|             |       | 16                 | 18                     | 157           |
| SciPy       | 35K   | 477                | 545                    | NA            |
|             |       | 305                | 333                    | NA            |
|             |       | 245                | 255                    | NA            |

To prevent combinatorial explosions, we restrict the number of SCs in each loop’s generation phase. We choose a value every 200 as the upper limit in the range of 200 to 1200 to find an appropriate configuration. PyCing can complete in under 2 hours in each setting. Due to space limitations, we only display the data from the project NumPy and place the results from other projects on our website.

The results in Figure 3 indicate that increasing the upper limit of SCs helps us to reach the top bound faster, and the exploration can finish in less than 40 loops. However, raising the maximum limit needs more time to complete one loop. Even though PyCing can complete exploration in 12 loops under the 1200 upper limit, this configuration takes longer than others. Considering the resource usage and execution speed, we believe 800 SCs per loop is the acceptable configuration.

Conclusion. The execution time comparison reveals that our instrumentation strategy has less influence on execution efficiency. Obtaining execution information from native libraries via dynamic binary instrumentation can be time-consuming or may fail since it demands monitoring the whole Python virtual machine.

C. Test Effectiveness of PyCing (RQ2)

In answering this question, we will first evaluate the efficiency of test cases in real-world projects, then investigate the improvement made by PyCing.

To quantify the size of native libraries developed by these projects, we use the tool cloc to count the lines of source code in C/C++, FORTRAN, and Cython. Besides, we count the types and numbers of Python/C APIs employed in the native libraries. Columns 3-6 of Table III demonstrate that the six projects develop their native libraries using a variety of languages and a large number of API functions, some of which are multilingual, providing challenges for building summaries.

We first count the number of SCs explored by existing test cases. The number of native methods is shown in the second column of Table IV and the number of types of input data used by test cases to the native methods is listed in the third column. The fourth column shows the number of SC explored by the test cases. According to the results, the test cases employ a wide range of input data to test native methods. But when compared with the number of explored SCs, we can observe that these test cases fall short of addressing execution paths with structure constraints.

We compare the number of SC explored by the test cases to the number identified by PyCing to explore whether PyCing
can increase test effectiveness. Figure 4 shows the comparing results. Because PyCing finds a significant number of SCs for a few native methods, we exhibit the distribution based on the number of SCs explored by the test cases.

As the results show, PyCing can find more SCs than test cases for most native methods, particularly in the SciPy project, where test cases examined 242 SCs whereas PyCing examined 160K totally.

As mentioned in Section IV-A2, we instrument type checking and attribute extraction API functions. To explore which category of API functions aid PyCing to improve test effectiveness, we count two kinds of API functions covered by PyCing and test cases. Besides, because the return values of API functions can cause the code to run alternative branches, we count the different types of return values of these functions. In particular, we regard returning true and false as two types of return values for type checking API methods and returning NULL and non-NULL to be two types for attribute extraction API functions.

Figure 5 presents the results. As shown, PyCing can cover more APIs for type checking and attribute extraction with the same amount of test cases, particularly for attributes. Moreover, PyCing can result in more types of return values for these API calls.

We also consider the influence of structures not covered by the test cases on the test effectiveness. The second column in Table V indicates the new API functions discovered by PyCing, and the third column gives the number of native methods that utilize these API functions. In the table, API functions ending in Exact return true if the argument is of the checked type but not a subtype, while API functions ending in get extract attributes. The results in Figure 5 and Table V demonstrate that PyCing can improve test effectiveness in two ways. First, PyCing can trigger various API return values for the API functions examined by the test cases. Second, PyCing can cover API functions not covered by test cases.

1) Bug Detection Results: In this paper, we use PyCing to detect memory leaks and segmentation faults in native libraries. The testing framework Pytest provides a checker pytest-leaks [38] for detecting memory leaks, while segmentation faults will make the interpreter terminated. We first use PyCing to produce test cases for the native methods in the six projects, and run these test cases via Pytest. We can produce test cases to trigger memory leaks in 9 native methods, and segmentation faults in 10 methods. Table VI shows distributions of bugs among the projects.
TABLE VI

| Projects  | Memory Leaks | Segmentation Faults |
|-----------|--------------|---------------------|
| CPython   | 1            | 1                   |
| Matplotlib| 1            | -                   |
| NumPy     | 4            | 2                   |
| Pandas    | -            | 2                   |
| Pillow    | -            | 3                   |
| SciPy     | 3            | 2                   |
| SUM       | 9            | 10                  |

Based on leaks we report to the developers, we can find that developers often cause leaks when using API functions to extract values from input data while ignoring the side effects of the change in the reference count of the return value. The following code utilizes two APIs to obtain the first element from an object o of type list. However, the first API raises item1’s reference count. Thus, when developers forget to reduce the reference count, a leak occurs. We advise developers to use uniform API categories to minimize misunderstandings about the behavior of the returned response.

```python
1. PyObject item1 = PySequence_GetItem(o, 0); // increase
2. PyObject item2 = PyList_GetItem(o, 0); // not increase
```

We cannot pinpoint the source of segmentation issues. It is due to the lack of debugging information in the project compilation process. Besides, the OS stops the interpreter before it outputs error information. As a result, even if we may produce many test cases that trigger the bugs, we count these bugs at the native method level.

A case study. The following code shows a leak found by PyCing in the project CPython and it affects Python 3.9, 3.10, and 3.11 [39]. When the native libraries _tkinter_flatten from lines 8 to 12 returns 0, function _tkinter_flatten does not reduce the reference of obj before returning at line 5. The input object needs to execute the false branch at line 9 to trigger this leak, and PyCing produces a str object for it. As test suites do not contain such a type, input data from existing fuzzers can not cover the satisfying type to trigger it. The input object generated by PyCing has been added to the test suites of this project.

```python
1. PyObject * _tkinter_flatten(PyObject * obj) {
2.   ... _tkinter_flatten(PyObject * obj) {
3.     if(!_flatten1(obj)){
4.       // fix patch: + Py_XDECREF(obj)
5.       return 0; }
6.   }
7.   int _flatten1(PyObject * obj){
8.     if(PyTuple_Check(obj) || PyList())
9.       return 1;
10.    return 0;
11. }
```

Conclusion. Real-world Python projects include plenty of native libraries written in multiple statically typed program-

D. The Comparison with other Fuzzers (RQ3)

To evaluate the effectiveness of PyCing, we choose to compare it with two state-of-art Python fuzzers, python-fuzz [40] and python-afl [41]. pythonfuzz is a coverage-guided fuzzing tool maintained by Google, which detects unhandled exceptions and memory limitations in Python libraries, while python-afl applies American Fuzzy Lop (AFL) for pure Python code. Currently, python-afl is an experimental module.

As indicated in Section I, current test tools cannot acquire code coverage from native libraries in the experimental projects, and the two Python fuzzers are unable to exploit structure constraints. To compare two fuzzers with PyCing fairly, we generate multiple sets of codes that contain the type and attribute judgments as benchmarks. Because pythonfuzz and python-afl are for pure Python and PyCing for testing native libraries, each set contains C and Python code snippets with the same functionality.

To ensure that Python and C code behave identically, we use Python/C APIs to process the types and attributes of input objects in C code, then write code templates with the same functionality for each API function in Python code. A type-checking API function is equivalent to the built-in method isinstance, and an attribute-extracting API function corresponds to another built-in method in. For simplicity, we do not process the attributes of the input objects in the benchmarks.

Thus, we can generate C code by adding multiple branches and randomly selecting API functions in the branch conditions, where we ensure each function receives the same object. Because the attribute-extracting API functions do not return a boolean variable, we compare the return result to NULL in the branch condition. Then, in the Python code, we may retain the form of the if statements while substituting handwritten templates for the APIs.

Figure 6 presents two pieces of codes. The left piece of code is for python-afl and pythonfuzz, while the right is for PyCing. In both Python and C code, lines 1 to 4 check if the input is of the dict type. Lines 5 to 10 in both sides inspect if the object includes the key ‘names’, and lines 11 to 16 check for the key ‘formats’. We use the Python code as fuzz target for pythonfuzz and python-afl while compiling the C code to produce dynamic link libraries for PyCing.
if 'formats' not in a.keys():
    return
if 'names' not in a.keys():
    return
    print("4")

if not isinstance(a, dict):
    return
    print("3")
    return
    print("2")
    return
    print("1")
    return
    if 'names' not in a.keys():
        print("2")
        return
        if 'formats' not in a.keys():
            return
            print("4")
            return
            print("3")
            return
            print("2")
            return
            print("1")
            return
            if not isinstance(a, dict):
                printf("3")
                return NULL;
            printf("4")
            return NULL;

Python code

C code

```
if not isinstance(a, dict):
    return
    printf("3")
    return NULL;
 PyObject names =
    PyMapping_GetItemString(obj, "names");
    print("2")
    return
    printf("2")
    return
    print("1")
    return
if (names == NULL) {
    printf("2")
    return NULL;
}  
if (formats == NULL) {
    printf("1")
    return NULL;
}
 PyObject formats =
    PyMapping_GetItemString(obj, "formats");
    print("3")
    return
    printf("3")
    return
    printf("1")
    return
    print("4")
    return NULL;
```

Fig. 6. Two snippets of handcrafted code for testing Pythonfuzz and Python-afl.

`pythonfuzz` and `python-afl` can only enter the True branch of the first boolean statement. The two tools are not effective in dealing with constraints on the structures. PyCing can explore all the execution paths based on the output of the handcrafted code. When the input objects generated by PyCing are seeded to these tools, the tools can mutate the values in the objects, covering the paths in these benchmarks.

**Conclusion.** Existing fuzzers are more concerned with the value of the input data than with its structure; as a result, they are ineffective when an execution path incorporates input data structures. Adding support for complicated structures and using feedback other than code coverage to assist generation can be improvable.

**VI. THREATS TO VALIDITY**

We will discuss some internal threats and external threats to our work in this section.

**Internal Threats.** The native method collection is one of the most significant internal threats. There are some native methods not found by the relevant research, or not recorded in the project documentation. But we believe that there are fewer undocumented native methods, or that some are soon to become outdated. On the other hand, the project does not want these methods to be called directly.

Another threat arises with the generation of new input data. Because we do not employ a constraint solver in the generation, there exist some constraints that we can not create new objects. Besides, because PyCing primarily focuses on the types and attributes, it cannot handle numerical constraints between two objects. However, embedding the structure constraints into a solver can result in deviation, and a solver may slow down the fuzzing process. Value constraints between attributes are difficult to find and handle even by analyzing the source code.

**External Threats.** We exclusively use PyCing to identify memory leaks, and PyCing may not perform well on other defects. Besides, because the memory leaks discovered by us came from four Python projects, the results may not apply to other projects. Also, we only build PyCing and instrument APIs with CPython, and may not work on other implementations of Python or native libraries. But, the six projects we choose are essential components in virtually all Python code, and the experiment results show that PyCing can investigate more execution paths in native libraries. The proposed instrumentation method can also be applied to other interpreters.

**VII. RELATED WORKS**

In this section, we briefly describe some related works.

**Detecting Memory Leaks via Testing Approaches.** Some studies employ testing techniques such as fuzzing to uncover new paths and provide methodologies for monitoring memory usage and detecting memory leaks. *Valgrind* is a widely used tool for detecting memory leaks. It monitors the execution of programs to collect information on memory allocation and free. Cristian et al. [14] use a solver on the acquired path conditions to generate new input data. Ma et al. [43] employ UI testing techniques and dump heap files to evaluate memory leaks. For 10 open source apps and 35 commercial apps, they report more than one leaked action or fragment. They create test cases by defining test generating methods for two types of GUI event sequences and following certain typical leak patterns. Wu et al. [44] combines static analysis and model checking to detect resource leaks in Android Apps. However, their test scenarios do not involve cross-language situations.

**Cross-language Analysis.** Some researchers attempt to provide summaries for native libraries to guide native method testing. Fourtounis et al. [27] expand the Java call graph to include both Java and native code. Zhao et al. [18] focus on the calls to several high-risk system APIs in native libraries and collect branch prediction data for the execution path that includes these APIs. To assist fuzzing in Java code, they employ path conditions gathered in native libraries. Lee [17] creates summaries for native libraries, which are subsequently used in the analysis of Java code.

**Testing Approaches in Python.** Many studies use testing techniques to identify flaws or wasteful code in Python projects. Lukaszczyk et al. [13] proposed a method for creating test cases automatically. Holmes et al. [45] put forward a method for guiding generation that relies on relative lines. The two methods focus on value generation and require users to define the types.

**VIII. CONCLUSION**

In this paper, we propose a lightweight approach for testing Python native libraries, and we implement it in a tool called PyCing. By imposing constraints on the structures of input data and instrumenting the Python Interpreter, PyCing can overcome the language barrier and improve test efficiency. The experimental results in six real-world Python projects show that PyCing explores execution paths more efficiently than state-of-the-art Python fuzzers and can generate seeds for these fuzzers to boost their test effectiveness. PyCing also finds memory leaks in 9 native methods and segmentation faults in 10 methods. Developers have confirmed eight leaks. In the future, we will introduce value constraints in native libraries to PyCing to improve test effectiveness.
[39] A memory leak in CPython. https://bugs.python.org/issue44608
[40] Pythonfuzz. https://pypi.org/project/pythonfuzz/
[41] Python-afl. https://github.com/jwilk/python-afl
[42] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, pages 89–100. ACM, 2007.
[43] Jun Ma, Sheng Liu, Shengtao Yue, Xianping Tao, and Jian Lu. Leakdaf: An automated tool for detecting leaked activities and fragments of android applications. In 41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, July 4-8, 2017. Volume 1, pages 23–32. IEEE Computer Society, 2017.
[44] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. IEEE Trans. Software Eng., 42(11):1054–1076, 2016.
[45] Josie Holmes, Iftekhar Ahmed, Caius Brindescu, Rahul Gopinath, He Zhang, and Alex Groce. Using relative lines of code to guide automated test generation for Python. ACM Trans. Softw. Eng. Methodol., 29(4):28:1–28:38, 2020.