Detecting Memory Errors in Python Native Code by Tracking Object Lifecycle with Reference Count

Xutong Ma$^{1,3,†}$, Jiwei Yan$^{2,‡}$, Hao Zhang$^{1,3,†}$, Jun Yan$^{1,2,3,§,†}$ and Jian Zhang$^{1,3,§,†}$

$^1$State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
$^2$Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
$^3$University of Chinese Academy of Sciences
Email: †{maxt, zhanghao19, yanjun, zj}@ios.ac.cn, ‡yanjiwei@otcaix.iscas.ac.cn

Abstract—Third-party Python modules are usually implemented as binary extensions by using native code (C/C++) to provide additional features and runtime acceleration. In native code, the heap-allocated PyObjects are managed by the reference counting mechanism provided in Python/C APIs for automatic reclaiming. Hence, improper refcount manipulations can lead to memory leaks and use-after-free problems, and cannot be detected by simply pairing the occurrence of source and sink points. To detect such problems, state-of-the-art approaches have made groundbreaking contributions to identifying inappropriate final refcount values before returning from native code to Python. However, not all problems can be exposed at the end of a path. To detect those hidden in the middle of a path in native code, it is also crucial to track the lifecycle state of PyObjects through the refcount and lifecycle operations in API calls.

To achieve this goal, we propose the PyObject State Transition Model (PSTM) recording the lifecycle states and refcount values of PyObjects to describe the effects of Python/C API calls and pointer operations. We track state transitions of PyObjects with symbolic execution based on the model, and report problems when a statement triggers a transition to buggy states. The program state is also expanded to handle pointer nullity checks and smart pointers of PyObjects. We conduct experiments on 12 open-source projects and detect 259 real problems out of 280 reports, which is twice as many bugs as state-of-the-art approaches. We submit 168 real bugs to those active projects, and 106 issues are either confirmed or resolved.

Index Terms—Python Native Code, Static Analysis, Reference Counting, Memory Error

I. INTRODUCTION

In recent years, Python has become one of the most popular languages [1, 2]. It is widely used as the host language for many application fields, especially for machine learning [3]. These Python scripts are usually executed on top of elaborate third-party modules, such as NumPy [4], TensorFlow [5], and PyTorch [6]. And to accelerate the onerous computational tasks and expand language features, these modules are usually implemented with native code (C/C++) as binary extensions by using the Python/C API to interact with the Python interpreter and corresponding user scripts [7].

In the Python interpreter, everything is a heap-allocated object called PyObject. To guarantee PyObjects in native code are properly recycled, the reference counting mechanism is employed to manage the lifecycle of PyObjects. The refcount of each PyObject is explicitly manipulated via increment and decrement APIs. And when the refcount is decreased to zero, the PyObject will be recursively recycled immediately.

Similar to lifecycle management bugs of other resources, i.e. unpaired source–sink API calls [8], forgetting to decrease the refcount will make the PyObject get leaked, whereas decreasing the refcount of a PyObject still being used can lead to a use-after-free or double-free defect. We call the memory errors caused by improper refcount operations the Refcount Bugs. Since the reference counting mechanism delays the destruction of a PyObject until its refcount is decreased to zero, the missing and redundant decrements are difficult to be observed and located. Furthermore, there has been limited research focusing on this problem [9].

Hence, the situation is severe. Figure 1 presents the trend of GitHub issues and pull requests about the Python reference counting mechanism. By August 2023, there have been more than 4 million issues and 7.9 million pull requests under this topic, which may indicate there would be a lot of Refcount Bugs hidden in real-world projects, and a usable checker to detect such problems is urgently required in the industry.

Typically, static analysis approaches check for resource management bugs by tracking the lifecycle of resources with typestate analysis on a Finite State Machine (FSM) [10, 11]. Checking refcount bugs is similar except that we cannot use an FSM to represent all values of a reference counter. Fortunately, groundbreaking contributions have been made in verifying the final refcount values before the control flow returns from native

Figure 1: Trend of issues and pull requests about the Python reference counting mechanism on GitHub

Number of (×$10^6$)

Issues
Pull Requests

$^†$Corresponding authors
and pointer assignments are extended to handle pointer nullity. For pointer nullity, the program state recording the PyObject state and refcount, the lifecycle states to avoid tracking dead PyObjects. Besides, changes and activate or deactivate the reference counter based on APIs and non-API functions. And new transition rules are added for PyObjects returned from different kinds of APIs and pointer operations. Based on the basic FSM transition for heap objects, new lifecycle states are added for PyObjects returned from different kinds of APIs and non-API functions. And new transition rules are added to the PyObject State Transition Model (PSTM) to describe the effects of Python/C APIs on the lifecycle and refcount of PyObjects (Section III-A).

- Compared with the basic FSM transition for heap objects, we make an extension to the program state to model nullity check onPyObject pointers and operations of PyObject smart pointer objects (Section III-B).
- We implemented PyRefcon based on the novel approach, with which we have found hundreds of real bugs verified by developers in large-scale projects (Section V-B).

II. BACKGROUND

In this section, we will briefly introduce the Python binary extensions from two aspects: the reference counting mechanism and refcount operations in APIs. Then we will propose the patterns of refcount bugs, and present the limitations of the state-of-the-art research with two examples extracted from real-world Python modules.

A. Python Binary Extensions

The Python binary extension represents the modules implemented with native code (C/C++) on top of the Python/C API [7]. In contrast with Python code executed on the Python interpreter [15], such modules are compiled as plugins to the interpreter and executed on a real CPU. Hence, memory errors in native code will directly crash the interpreter.

Everything in Python is an object of class PyObject. And nearly all PyObjects are allocated on the heap [16]. Hence, PyObject pointers are used almost everywhere in native code [16]. To track the lifecycle of PyObjects and destruct PyObjects properly, the interpreter employs the reference counting mechanism.

B. Reference Counting Mechanism

In native code, developers manipulate PyObjects and communicate with the interpreter via more than 1,000 APIs [9]. Among them, in addition to the refcount increment and decrement APIs, another 384 APIs, according to our statistics, will also modify the refcount of their parameters and return values. And the refcount operations in each API are presented in the documentation [7].

PyObjects are shared in native code. Each PyObject has a refcount field to track the number of its references. The counted conceptual references represent the shared ownership of a PyObject, whereas the concrete pointer variables are used to access the PyObject. Hence, the number of references and pointers to a PyObject are not necessarily equal, as refcount increments and decrements in a scope are mutually canceled and can hence be pruned for simplicity [16].

1) Directly Manipulate Reference Count: The refcount is explicitly increased and decreased via refcount APIs. Increasing the refcount acquires a reference to the PyObject and hence blocks the destruction of the PyObject until the reference is released via decrement. When the refcount is
decreased to zero, it means all references to the PyObject are released. And the PyObject will be destructed recursively and automatically.

2) Reference Count Changes in APIs: If we consider every API as a closed box, we can omit the internal refcount changes for the functionalities in the API. Apart from these modifications, an API can still have influences on the refcount of the PyObject passed to or returned from it in two ways as illustrated in the documentation [7].

On one hand, for an API returns a PyObject, it can acquire a reference to the PyObject being returned on the caller’s behalf, such as PyObject constructors like PyLong_FromLong. They are tagged as returning a reference in the documentation. And the caller needs to consume the reference when the PyObject is no longer used. We can analogize these APIs as additional sources of references.

On the other hand, when a PyObject is passed to an API as an argument, the API can decrease its refcount, or take a reference from the caller. For instance, API call PyList_SetItem(List, i, Item) equals to Python code List[i] = Item, which stores PyObject Item to the i-th element of list List. And the API will transfer a reference to Item from the caller context to List instead of acquiring one via refcount increment inside the API. These APIs are tagged as stealing a reference for the argument, which can be seen as extra sinks releasing references.

3) Smart Pointers of PyObjects: In addition to manually decreasing the refcount when a reference is no longer needed, the refcount of a PyObject can also be automatically decreased with C++ Smart Pointers.

For C++ objects wrapping a PyObject pointer and decreasing the refcount in their destructors, which have similar structures and behaviors to the C++ smart pointers, we call them the refcount monitors. Modeling operations of refcount monitors is essential, as it is a common idiom in C++ to wrap a handle to a resource and release it in destructors [17].

C. Refcount Bugs

Different from the state-of-the-art work that defines bug patterns based on refcount values [13], [18], we analogize the counted references to heap objects and define refcount bugs based on two kinds of lifecycle bugs of heap objects: memory leak and use-after-free.

Similar to memory leak bugs, when a reference is not released before all its pointer variables go out of scope, the reference can no longer be released. It will block the destruction of the PyObject and make it occupy the system memory for a long period. We define such a symptom a reference leak (RL) bug.

And analogize with use-after-free bugs, when the references acquired in a scope are released, the PyObject will be destructed. If the PyObject is then used again, a use-after-release (UaR) bug will be triggered. Besides, for PyObjects returned from APIs that do not acquire a reference for the caller, decreasing its refcount is also considered a pending use-after-release bug.

D. Motivating Example

The state-of-the-art approaches draw attention to the total refcount changes on a program path. Without the lifecycle state of PyObjects, they will continue tracking destructed PyObjects and miss bugs in the middle of program paths.

- File: src_webp.c
  117 PyObject *_anim_encoder_new(...) { ...
  165    encp = PyObject_New(...); // Created
  166    if (encp) {
  171      if (...) return encp; // Sink-1: Returned...
  175      PyObject_Del(encp); // Sink-2: Destructed
  177      PyErr_SetString(...);
  178      return NULL; // No leaks
  179 }  

Fig. 2: A correct function from Pillow, where approaches based on only refcount will generate a false leaking report

- The first example shown in Figure 2 is a snippet without refcount bugs. The API PyObject_New createsPyObject encp and returns a reference on line 165. Then on line 171, when the if condition is satisfiable, encp is returned to the caller together with the reference (Sink-1). Otherwise, it is explicitly destructed on line 175 (Sink-2).

With lifecycle state transition, we can use two states to distinguish a live PyObject from a destructed one, and properly deactivate the refcount of encp after its destruction. As dead PyObjects cannot be simply represented with refcount changes, the approaches based on refcount changes will report that encp leaks.

- File: numpy/core/src/multiarray/ctors.c
  2849 PyObject *PyArray_Zeros(...) { ...
  2857    ret = PyArray_NewFromDescr_int(...); // 1. Captured...
  2859    if (_zerofill(ret) < 0) {
  2870      Py_DECREF(ret); // 3. Use-after-release
  2871      return NULL;
  2872    } ...
  2876    return (PyObject *) ret;
  2877  }

- File: numpy/core/src/multiarray/common.c
  47 int _zerofill(PyArrayObject *self) { ...
  154    if (...) {
  155      Py_DECREF(self); // 2. Decrease, destructed
  156      return -1;
  163      return 0;
  164 }  

Fig. 3: A confirmed use-after-released bug found in NumPy, which is detected by PyRefcon and fixed after submitted

- The second example in Figure 3 shows a use-after-release bug found from NumPy [1]. On line 2857 of the first file, the PyObject ret is created from a function pointer call inside non-API function PyArray_NewFromDescr_int

[https://github.com/numpy/numpy/issues/19859](https://github.com/numpy/numpy/issues/19859)
and passed to function _zerofill as parameter self. Then on line 155 of the second file, the refcount of the PyObject is decreased. And finally, on line 2870 of the first file, the refcount is decreased for the second time. After digging into the code base, we discovered that function PyArray_NewFromDescr_int returns a reference. Hence, the second decrement on line 2870 can be recognized as a use-after-release bug.

Unfortunately, such bugs cannot be precisely reported by state-of-the-art approaches. Since the refcount is implicitly increased in the function pointer call, which is difficult to be inferred statically, available approaches will give up tracking its refcount and hence leads to a false negative. And due to the limitation of defining bug patterns based on refcount values, forcing reporting such issues will lead to a lot of false positives. Since the initial refcount is unknown, to precisely report such issues, we need to design a specific lifecycle state for PyObjects returned from non-API functions.

III. MODELING AND CHECKING PyOBJECTS

In this section, we first introduce the PyObject State Transition Model (PSTM) with the designation of lifecycle states and transition rules under lifecycle and refcount operations. Then, we present solutions for handling pointer nullity checks and refcount monitors with enhanced program state. Figure 4 shows the workflow of modeling PyObjects and checking refcount bugs.

![Fig. 4: Workflow of checking refcount bug with PSTM](image)

According to the workflow, the inputs of the analysis include source files, models for refcount operations in Python APIs, and the definition of refcount monitors. The Statement Evaluator will model the semantics of the code and store the results in the program state. The encountered lifecycle, refcount, and pointer operations will be passed to the PyObject Manager to apply the models. And the modified program state on each program point is checked finally with Bug Checkers.

A. PyObject State Transition Model

For a PyObject, we track its lifecycle state transition and refcount changes with a triple \( (id, st, rc) \), where
- \( id \) is the unique identifier of each PyObject;
- \( st \in \Sigma \) represents its lifecycle state;
- \( rc \in \mathbb{N} \cup \{N/A\} \) measures its refcount value.

The set of lifecycle state \( \Sigma \) is defined as \( \Sigma = \{ Created, Borrowed, Captured, Released, Escaped, Leaked, Reused \} \). And among the states in set \( \Sigma \), refcount is activated only for state Created and Escaped. For others, we use N/A indicating that the refcount is not tracked.

We define the lifecycle state transition by expanding the existing FSM transition for heap objects \([10], [11]\), as presented in Figure 5. Different from the available model designed for a Created–Released switch state under paired source–sink APIs \([8]\), designing the lifecycle state for PyObjects needs to consider the refcount and the API effects together.

![Fig. 5: Basic FSM transition model for heap objects](image)

When designing the transition rules, we need to consider the mutual influences between lifecycle states and refcount. Lifecycle operations modify the lifecycle states, as well as activate or deactivate the refcount. Whereas refcount operations can manipulate the refcount and trigger lifecycle state transitions simultaneously. Figure 6 presents the transition rules of lifecycle states. And we will introduce the states by following the transitions from left to right.

![Fig. 6: Lifecycle state transition of a PyObject, where reference counted states are tagged with a star mark. Increase and Decrease denote the refcount increment and decrement; Prune represents the destruction triggered automatically after refcount decreased to zero or explicitly by calling lifecycle APIs; Detach indicates the PyObject is returned or assigned to non-stack pointers; Use represents all dereference operations to a PyObject pointer; and Prune means that all pointers to the PyObject go out of scope, and the corresponding PyObject symbol is being removed from the program state.](image)
returned pointers are not aliased. We will discuss the reasons for this assumption in Section [VI].

- APIs returning a reference. Most of these functions are PyObject constructors creating a new PyObject, such as function PyObject_New in Figure [2]. Since the returned PyObject is usually newly created with a refcount of one, users should consider that it is neither referenced by nor an alias of another PyObject. To emphasize the difference with other kinds of PyObjects, their initial states are set to \( (id, \text{Created, 1}) \), which is similar to the initial state of heap objects.

- APIs not returning a reference. In native code, such APIs are usually the getter methods, such as PyTuple_GetItem accessing the content of a tuple object. By calling them, the caller context only receives a pointer to the returned PyObject for temporary use. This means destructing the returned PyObject or releasing its references will introduce a use-after-release problem for the sources providing the returned PyObject. To distinguish them from newly created ones, we use state \( (id, \text{Borrowed, N/A}) \) to represent such PyObjects until the caller acquires a reference to it.

- Non-API functions returning a PyObject from an unknown source, such as function PyArray_NewFromDescr in Figure [3] that returns a PyObject from a function pointer. Since we cannot know the behavior of the callee, the returned PyObject can be either of the above cases. Hence, we use state \( (id, \text{Captured, N/A}) \) representing the superposition state of Created and Borrowed. And it will collapse to either of them through future operations.

2) PyObject Detachment: For a PyObject in the Created state, when it is (a) returned from the entry function of the analysis, or (b) assigned to a pointer variable accessible from outside of the analysis context, operation DETACH will change its lifecycle state to Escaped without modifying its refcount.

\[
\text{DETACH; } s = (id, \text{Created, rc})
\]

As we do not know whether external pointers need to hold a reference, this state represents that we cannot precisely know what value its refcount should exactly be. And we only report use-after-release bugs for an Escaped PyObject when it is explicitly destructed, or its refcount is decreased to less than zero, which is similar to a Borrowed PyObject.

3) Refcount Operations: The refcount of a PyObject is manipulated by only one unit with operation INCREASE and DECREASE. In the PSTM, the refcount is measured with the number of references in the current analysis context, rather than simply accumulating increments and decrements. Hence, for a PyObject passed to a reference-stealing API, we also decrease its refcount by one with operation DECREASE.

\[
\text{INCREASE; } s = (id, st, rc) \land rc \neq N \\
\text{DECREASE; } s = (id, st, rc - 1)
\]

As mentioned in Section [II-B3] the stolen reference is no longer managed by the current context, it can hence be released at any time. This may lead to use-after-release bugs for further dereferences. By modeling the behavior of reference-stealing APIs with decrement, we can now additionally detect such use-after-release threats rather than consider them as safe, just like what the state-of-the-art approaches do.

When decreasing the refcount from one to zero \( (rc = 1) \) through either refcount decrement or reference-stealing APIs, for Created PyObjects, the decrement will automatically trigger its destruction as required by the refcount decrement mechanism mentioned in Section [II-B1].

\[
\text{DECREASE; } s = (id, \text{Created, 1})
\]

whereas for an Escaped PyObject, we will not destruct the PyObject but continue tracking its refcount changes by decreasing it to zero to check further use-after-release bugs as illustrated in Section [III-A2].

\[
\text{DECREASE; } s = (id, \text{Escaped, 0})
\]

Since state Captured represents the superposition state of Created and Borrowed (Section [III-A1]), we define the following two transition rules based on assumption and inference. First, when decreasing the refcount of a Captured PyObject, if the function returning it does not return a reference (i.e., this Captured PyObject is actually a Borrowed PyObject), the decrement will trigger a use-after-released bug. Therefore, to avoid generating false alarms, we handle such Captured PyObjects as Created ones.

\[
\text{DECREASE; } s = (id, \text{Captured, N/A})
\]

Second, if a refcount increment appears in the code, there are great chances that the current context is not holding a reference to the PyObject. Hence, when increasing the refcount of a Captured PyObject, we will first collapse the superposition state to Borrowed before the refcount increment. While for a Borrowed PyObject (both returned and collapsed), a refcount increment will make the current context hold one reference to it. Therefore, we change state Borrowed to \( (id, \text{Created, 1}) \), which is the same as a PyObject returned from an API returning a reference, and begin tracking its refcount changes since then.

\[
\text{INCREASE; } s = (id, st, \text{N/A}) \land st \in \{ \text{Borrowed, Captured} \}
\]

\[
\text{DECREASE; } s = (id, \text{Created, 1})
\]

4) PyObject Destruction: Similar to deallocated heap objects, we also need a state to represent a destroyed PyObject whose occupied memory is reclaimed. We use this state to distinguish live PyObjects from dead ones to check use-after-release bugs.

PyObject destruction will be triggered automatically with refcount decrement, or directly invoked when lifecycle APIs are called explicitly (e.g., calling PyObject_Delete on line 175 in Figure [2]). Operation DESTRUCT will change the lifecycle state to Released and deactivate the refcount. As illustrated when introducing refcount decrements, only PyObjects in state Created and Captured can be destructed, i.e., state Captured is collapsed to Created before the destruction.
5) Transition to Buggy States: Transitions to buggy states show how a refcount bug is triggered. And the buggy states represent that an identified bug on the PyObject has been reported. State *Leaked* represents reference leaks and state *Reused* stands for use-after-released bugs.

When a PyObject *s* loses all pointers pointing to it, it cannot be accessed again in the analysis. Operation *PRUNE* will remove it from the program state. If its refcount is not zero, its lifecycle state will be changed to *Leaked*, and report a reference leak bug for it.

For *Borrowed* PyObjects and *Escaped* PyObjects with a refcount of zero, refcount decrement will release a reference not measured in the current analysis context and make an external reference dangling. Hence, we report such behaviors as use-after-release bugs.

Similarly, explicitly destructing PyObjects in state *Borrowed* or *Escaped* with operation *DESTRUCT* is also recognized as use-after-release bugs.

B. Enhanced Program State

We employ symbolic execution to analyze the code and track the state changes of PyObjects. To model the behavior of refcount monitors (Section II-B3), we enhance the program state with a new set *M* and define the new program state as *P* that *P* = (*V*, *S*, *C*, *M*) where

- *V* contains the assignments of all tracked variables as a map from a variable to its assigned value (var → value);
- *S* stores PyObject state tuples currently being used;
- *C* is the set of path constraints;
- *M* stores refcount monitor objects.

In this section, we introduce operations modifying the program state during the symbolic execution.

1) Modeling Refcount Monitors: We model the behavior of refcount monitors with a simplified C++ smart pointer model in our previous research [19].

When a monitor object *m* is constructed, it will be added to the monitor set *M*. When assigning a PyObject to a monitor via method calls, we will store the assignment in the variable set *V*, and remove it during reassignment.

The additional monitor assignment can extend the lifetime of thePyObject like other pointer variables, which can delay the leak checker in operation *PRUNE* until the monitor object goes out of scope and gets removed from set *M*.

The reference held in the monitor will not trigger a refcount decrement immediately after the assignment. And when the monitor goes out of scope, operation *PRUNE* will then remove the assignment from variable set *V* and *decrease the refcount of the PyObject*.

We apply the monitor model on an object if it is an object of the C++ standard Unique Pointer [20] with a refcount-decreasing deleter, or (a) it contains *one and only one* private PyObject pointer field, (b) it has a constructor accepting a PyObject pointer argument and initializing the field with the argument, and (c) its destructor decreases the refcount of the wrapped PyObject pointer. Besides, users can also manually add other classes of monitors if they are not automatically recognized with the above rules.

2) Pointer Nullity Check: Nullity check on PyObject pointers is frequently used in native code to check whether an API call succeeded. The state-of-the-art approach forks the program path to non-null and null branches when a PyObject is created and returned from an API [13]. To simplify the path constraints and reduce unnecessary path forks, we introduce the lazy-check strategy for nullity check on PyObject pointers.

After the construction of a PyObject, its nullity is kept undetermined until checked on branch conditions or dereferenced directly. On the null branch of the nullity check, operation *PRUNE* will remove the PyObject from symbol set *S* and reset the assignments of PyObject pointers to NULL.

Whereas on the non-null branch, we will add a non-null constraint *σ* ! = NULL to the constraint set *C* with operation *USE*. Besides, if a PyObject is directly used without a nullity check, we will also append the non-null constraint to set *C*, as developers believe that it cannot be NULL.

IV. Case Study

In this section, we will use three examples to illustrate the detailed program state transition. For assignments of PyObject pointers in set *V*, we use their *ids* for simplicity.

- Figure [7] shows the transition of program state for the example in Figure [2]. As no monitors are present in the example, we omit the monitor set *M* in the program state.
On line 165, we apply the returning-a-reference operation on API PyObject_New according to the API model, which creates PyObject <S165, Created, 1> and assigns it to pointer encp. Then on line 166, we apply the pointer-nullity-check operation on the if statement. On the then branch, a non-null constraint is appended; and on the else branch, the original assignment is cleared with the assigned symbol removed. Next on line 171 if the path constraint C171 is satisfied, PyObject S165 is returned with its lifecycle state changed to Escaped (Sink-1). Otherwise (~C171), PyObject S165 is destructed with its lifecycle state changed to Released and refcount deactivated (Sink-2). And hence, we can avoid a false reference leak report for S165.

On line 2849, we create PyObject S2857 with an initial state of Captured, according to the PyObject model. As it is used directly by passing to function _zerofill, we append a non-null constraint for S2857. And the parameter self of the callee is also aliased to S2857 before the function call.

In the callee, if the path follows the then branch (C154) on line 154, the lifecycle state of S2857 is transited to Released with the refcount decrement according to the transition rules. Whereas on the else branch (~C154), the state of S2857 is not changed.

After returning from the callee, the dead assignments, symbols, and path constraints are pruned from the state. And the return value is stored via built-in variable $? in the variable assignment set V. As the branch condition on line 2869 can be directly determined with the return value, the analysis will follow the correct path of the if statement. On the then branch, the dangling pointer to PyObject S2857 is used. This makes the lifecycle state changed to Reused indicating a use-after-released bug. Whereas on the else branch, the PyObject is returned with its lifecycle state unchanged.

- File: torch/csrc/Module.cpp

592  THPObjectPtr list(PyList_New(...));  // Monitor assignment ...
595  if (...)  throw python_error;  // Monitor destructor call

(a) An example function extracted from PyTorch, where our tool generates a false positive without the model of refcount monitors. Class THPObjectPtr implements the refcount monitor.

(b) Program state transition on the exception branch

Fig. 9: Example of automatic refcount decrement

- Figure 9 presents the program state transition when the analyzer applies the model of refcount monitors. When the PyObject symbol S592 is created on line 592, the constructor of the monitor object list is invoked. It assigns the pointer field list.ptr to S592. The monitor model then adds list to the monitor set M and adds the monitor assignment of list→S592 to the assignment set V.

If the execution takes the then branch on line 595 it will throw a python_error exception and terminate the path. The analysis engine then removes the dead assignment...
list.ptr -> S592, and the monitor assignment helps to extend the lifetime of S592. Finally, monitor object list is removed during stack unwinding, which decreases the refcount of S592 and changes its lifecycle state to Released. Without the monitor model, the analyzer will generate a false alarm reporting S592 leaks a reference, when assignment list.ptr -> S592 is removed.

V. EVALUATION

To evaluate the effectiveness and usability of revealing refcount bugs, we carried out the following experiments to answer the three research questions below.

- **RQ 1**: How many refcount bugs can we detect? What are the reasons for false positives?
- **RQ 2**: How is the result compared with other tools? What are the reasons for our false negatives?
- **RQ 3**: How much system resource is consumed during the analysis compared with other tools?

A. Setup of Experiments

**Tool Implementation.** We implement our tool, PyRefcon, on top of a static symbolic execution engine, the Clang Static Analyzer (CSA) [14]. The PSTM and enhanced program state are added together with new checkers to detect refcount bugs based on state transition. Refcount operations of 384 APIs are semi-automatically extracted from the formatted documentation and added to the analyzer.

To speed up the analysis process, we execute PyRefcon on each translation unit separately in parallel with Panda [21]. And the interfile function calls are handled with the cross-translation-unit analysis mechanism of CSA. To remove redundant bug reports generated for the same buggy site from different analyzer instances, we deduplicate bug reports with a categorization and sorting based strategy [22].

**Environment and Tools.** The experiments are executed on a Linux server with Intel® Xeon® E5-2680 v4 CPU of 56 threads and 256 GB of memory. Unfortunately, the state-of-the-art tools, Pungi [13] and RID [18], are not publicly available. We hence evaluate our tool against CpyChecker [23], which is the tool that both Pungi and RID have compared with. It implements refcount bug checks with a path-sensitive approach similar to Pungi. Besides, we also literally compare our results with the original data provided in the paper of Pungi and RID.

We build CpyChecker from source on an Ubuntu 18.04 Docker image with GCC 6.5, which are the highest versions that CpyChecker can work with [24]. The experiments are executed under Python 3.8 header files for both PyRefcon and CpyChecker.

**Benchmark Composition.** The benchmark is composed of two parts: the open-source Python binary extension modules used by the state-of-the-art research, as well as popular large-scale open-source modules written in C/C++. The detailed information is presented in Table I.

For the first part, we select six out of thirteen projects from the benchmark of Pungi [13] and RID [18]. Among the expunged projects, four of them (krbV [37], ldap [38], py-OpenSSL [39], and netifaces [40]) cannot support the Python 3 API under the selected versions; whereas for the other three (gst, canto, and yum [41]), we cannot find a Python binary extension module with the names on GitHub.

For the second part, six other projects are selected for their reported refcount bugs in our previous research [42]. These projects mainly cover the scene of machine learning, scientific computation, and so on. These usage scenarios have high efficiency and usability requirements and sometimes need to be executed with limited system resources. Hence, it is important for these projects to avoid memory leaks and use-after-free bugs that can be triggered with refcount bugs.

B. Effectiveness and Correctness (RQ 1)

**Reports generated by PyRefcon.** To evaluate the effectiveness of checking real-world projects, we run PyRefcon on the benchmark. Table I shows the statistics of the reports generated by PyRefcon.

Reports are manually reviewed by two authors to determine their correctness. As seen in the table, 259 true bugs are identified from 280 reports according to the manual revision of two authors. Beyond the 280 reviewed reports, the deduplication strategy for bug reports removes 270 redundant reports (Column Redu.), which report the same bug triggered from different entries with similar paths.

**Reports of initial state Borrowed and Captured.** Among the bug reports, 18 reports have these two kinds of initial state (Column B.&C.), where 16 of them are real bugs together with

---

### TABLE I: Information of evaluated open-source Python binary extension modules.

<table>
<thead>
<tr>
<th>Project</th>
<th>Source</th>
<th>Kloc</th>
<th>PyRefcon</th>
<th>CpyChecker</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyaudio</td>
<td>Pungi</td>
<td>2.86</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>pycrypto</td>
<td>Pungi</td>
<td>17.48</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>pyxattr</td>
<td>Pungi</td>
<td>1.23</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>rtdtool</td>
<td>Pungi</td>
<td>1.48</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dbus</td>
<td>Pungi</td>
<td>12.59</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cpychecker</td>
<td>Pungi</td>
<td>0.30</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NumPy</td>
<td>-</td>
<td>298.05</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SciPy</td>
<td>-</td>
<td>1,146.91</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Numba</td>
<td>-</td>
<td>251.09</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pillow</td>
<td>-</td>
<td>22.08</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>-</td>
<td>43.74</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PyTorch</td>
<td>-</td>
<td>2,617.65</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
2 false positives due to the side effects of third-party function calls.

Reasons for false positives. The main reason for false positives is due to the unmodeled behaviors of PyRefcon. The root causes can be categorized into three clusters.

We mark six reports as false positives since we do not model the functionality but only the refcount operations of API functions. The side effects of API functions and implicit constraints between PyObjects make PyRefcon follow an in-feasible program path, which finally leads to a false positive.

Another five false positives are caused by the analysis engine. When a noreturn function is called on the path, such as function abort, the engine will first invoke the reference leak checker before terminating the path. This makes PyObject with unreleased references reported as leaked. These reports can be eliminated with an option of CSA, which will move the dead symbol deletion to the end of the path, but the memory usage will be increased.

There are also three reports marked as false positives as we assume that the developers did this on purpose. And four other false positives are due to the path provided in the report is not complete and cannot be reviewed.

False positives pruned by monitors. The recognition strategies for monitors find three wrapper classes as refcount monitors. When applying the monitor model to objects of these classes, as shown in Column FP\textsubscript{Mo}, all 59 false positives related to refcount monitors are eliminated with one additional true positive newly reported, and no true positives originally reported are suppressed with the monitor model.

Among the pruned 59 false positives, 48 of them are triggered by throwing exceptions like the example in Figure 9.

3 false alarms are triggered by noreturn functions in assertions, which is similar to the previous type; and the rest 8 reports are generated because the destructor of the monitor object is not called for the invocation limitation of the analysis engine. The experimental results prove that modeling refcount monitors are helpful in suppressing related false positives.

Revisions from Developers. To further confirm the fairness of our revision on reports, we submit 168 (60%) true positives to the developers. Among them, 22 of them have been confirmed; 84 of them have been fixed according to our reports; 53 of them are still pending; and 9 of them are closed as developers do not think there are problems with their knowledge of the code.

The rest 112 reports are not submitted, where 21 of them are false-positive reports; 9 of them are fixed or deleted before our submission; 51 of them from two projects (pyaudio and dbus) are due to the projects do not have an interactive issue system; and 31 of them from two projects (pycrypto and rrdtool) are due to the projects being out of support.

C. Comparison with Related Tools (RQ 2)

Comparison with CpyChecker. To further compare the effectiveness, we compare our reports against those generated by CpyChecker. The number of common true positives with CpyChecker is shown in Figure 10a.

After reviewing the reports, we found that most of the reference leak bugs missed by CpyChecker are caused by incomplete program paths. The CpyChecker will miss the bugs far from the entry, whereas our PyRefcon can track deeper call stack. This observation may be caused by the reporting strategy of CpyChecker and its stability problem when analyzing Python 3 projects. And 11 use-after-release bugs are not reported by CpyChecker. This is due to lacking the lifecycle state of PyObjects when tracking only the refcount values as mentioned in Section II-D.

Among the reports generated by CpyChecker, one reference leak bug is not detected by PyRefcon. The reason is that the corresponding source file is not analyzed, as it is not compiled when building the project. The difference in compiler versions used by PyRefcon and CpyChecker leads to this result.

Comparison with Pungi and RID. To evaluate the performance of PyRefcon against the state-of-the-art approaches, we literally compare the results provided by Pungi and RID in their papers [13, 18]. The number of true positives of these four tools is shown in Figure 10b.

According to the figure, our tool can report twice more true bugs as CpyChecker does for the total number from both Pungi and RID. And compared with the data provided by Pungi, our tool can still find approximately twice as many true bugs from the latest versions of the compared projects, where the bugs previously found may have been fixed if they were submitted to the developers. Whereas for RID, it can find four more bugs than PyRefcon on the evaluated project.

Since neither tools nor detailed reports generated by them are publicly available, we cannot tell the intersection and
TABLE III: Time and memory consumption of PyRefcon and the CSA engine when executing them in PyRefcon.

<table>
<thead>
<tr>
<th>Project</th>
<th>Time (Seconds)</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Overhead</td>
</tr>
<tr>
<td>pyaudio</td>
<td>1.18</td>
<td>3.28×</td>
</tr>
<tr>
<td>pycrypto</td>
<td>2.89</td>
<td>1.52×</td>
</tr>
<tr>
<td>pyxattr</td>
<td>1.54</td>
<td>1.49×</td>
</tr>
<tr>
<td>rrdool</td>
<td>5.30</td>
<td>1.70×</td>
</tr>
<tr>
<td>dbus</td>
<td>6.25</td>
<td>1.74×</td>
</tr>
<tr>
<td>duplcity</td>
<td>0.02</td>
<td>1.08×</td>
</tr>
<tr>
<td>Numba</td>
<td>489.93</td>
<td>1.63×</td>
</tr>
<tr>
<td>SciPy</td>
<td>588.19</td>
<td>2.03×</td>
</tr>
<tr>
<td>Numba</td>
<td>10.44</td>
<td>1.50×</td>
</tr>
<tr>
<td>Pillow</td>
<td>67.94</td>
<td>1.77×</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>7,243.75</td>
<td>1.28×</td>
</tr>
<tr>
<td>PyTorch</td>
<td>4,735.28</td>
<td>1.15×</td>
</tr>
<tr>
<td>Total</td>
<td>13,152.89</td>
<td>1.26×</td>
</tr>
</tbody>
</table>

Fig. 10: Comparison among all tools. Subfigure (a) shows the number of common and respective true positives of PyRefcon against CpyChecker, where the projects for which CpyChecker can generate reports (the projects marked as ✓ or ✓ in Table III) are measured. Subfigure (b) presents the number of true positives of Pungi, RID, and CpyChecker from their original papers against PyRefcon, where the data of the corresponding source (Column Source) is measured.

differences between PyRefcon and the other tools. And hence, we cannot draw further conclusions from this comparison.

D. Resource Consumption (RQ 3)

In this subsection, we show the resource consumption of PyRefcon against the CSA engine (by executing PyRefcon with all our checkers and models disabled) and CpyChecker. The data is measured with the average value of five executions.

Table III shows the resource consumption comparison of PyRefcon and the CSA engine when executing them in parallel. For all projects in the benchmark, the total time consumption under a concurrency of 16 parallel processes is 13,152.89 seconds or 3.65 hours, which spends 1.26× time than the CSA engine. Besides, we also estimate the upper bound of peak memory usage under concurrent analysis with the sum value of peak memory usage of the top 16 translation units. According to the peak memory usage of each translation unit, the estimated upper bound of memory consumption under a concurrency of 16 parallel processes is 3.47 GB, which uses 1.84× memory as the CSA engine.

To evaluate the detailed resource consumption, we also measure the time cost and peak memory usage for each translation unit separately. Among the 12,208 translation units in 12 projects, 4,415 kilo lines of code are analyzed. The average resource consumption per Kloc is 81.29 seconds and 2.00 GB. Compared with the CSA engine, where resource consumption is 30.46 seconds and 0.91 GB, the overheads are 2.67× and 2.20× respectively.

Furthermore, we also measure the resource consumption of PyRefcon and CpyChecker on the three projects that CpyChecker can correctly analyze. Among them, PyRefcon uses 49.99% memory and 38.71% time on average compared with CpyChecker.

VI. Threat to Validity

Handling Pointer Aliases. Symbolic execution does a precise pointer analysis that can detect all alias relations inside the current analysis context. Whereas for external pointers, such as parameters and return values of a callee function, we assume that they point to separated PyObjects, and analyze their refcount and lifecycle state respectively.

Since it is difficult to have a full vision over the project, third-party libraries, and the interpreter, a preferred solution for developers to avoid potential refcount bugs is to follow this assumption, especially for developers in large projects. And the submitted bug reports also prove that it will not introduce false alarms on aliased pointers, as developers can neither tell whether two external pointers can be aliased.

Captured PyObjects. In PyRefcon, we design the inference-based transition rules for Captured PyObject by following a conservative manner. And our evaluation has successfully shown the low false positive rate of the strategy. Unfortunately, there would be some bugs missed by our approach. To overcome this problem, users can add specific model definitions for the private and third-party API functions in their projects by imitating our built-in models for Python/C APIs.

Besides, only the PyObjects returned from functions will be tracked, whereas PyObjects accessed from global variables and parameters are not considered. Due to the conservative management strategies of CSA for global variables, the PyObjects assigned to such pointers will be soon invalidated and pruned, even though the PyObjects that are pointed to by global variables can also be analyzed with our approach. To avoid false alarms and unreasonable reports, we also drop them in PyRefcon, which may also lead to false negatives.

Selection of compared tools. In this paper, we compare our tool against CpyChecker in practice, as it is the only tool publicly available and compared with both state-of-the-art work, even though it is outdated and out-of-support.
To compensate for this unfairness, we carry out the literal comparison with state-of-the-art work through the original data from their papers. Besides, as all these three tools cannot support the latest stable version of Python (version 3.9 when writing this paper), the results may be affected by the behavior changes of API functions, which can affect the fairness of our comparison.

VII. RELATED WORK

As a static analysis approach checking for refcount bugs in Python binary extensions, PyRefcon is mainly related to the topic of Reference Counting checkers, and native code analysis on languages with virtual machines or interpreters.

Reference Counting Checkers. For projects using the Reference Counting mechanism to manage heap objects, developers usually have their dynamic checkers to detect cyclic structures or check refcount values during execution. In the Python interpreter, a generational garbage collector is used to detect circular reference problems for container objects [43]. Related PyObjects are required to be registered to make the garbage collector check their references. In Firefox, they use a tracking and balancing approach to find out potentially forgotten decrements [44]. Logs are dumped during the tracking process, and then an analysis step will locate the bugs.

Li and Tan proposed a tool called Pungi [13] for checking the Python Reference Counting Errors. They transformed the original C code to affine programs [12] and checked the property violations of reference counts with a path-sensitive verification tool on the affine code. But during the transformation, a lot of information such as path constraints were dropped. Whereas our approach can precisely analyze the code with a very low false-positive rate. Similarly in CpyChecker made by Malcom [23], the same error patterns are followed. They used the Static Single Assignment (SSA) form in GCC to analyze the code for reference counting bugs with an intraprocedural and path-sensitive approach.

Mao et al. created RID [18] that also checks the Reference Count Bugs. They used a summary-based approach to collect operations on reference counts and search for path pairs whose refcount changes are different but external effects (return values, etc.) are the same. Similar to Pungi, the input code is also transformed to affine programs before being analyzed. Tan et al. proposed an approach to checking reference counts by pairing the increment and decrement operations via their path conditions [45]. And they tried to further report reference count bugs by comparing the same increment and decrement function pairs and reporting the cases different from the majority.

Besides, Emmi et al. presented an approach to verifying the implementations of reference counting mechanisms [46]. They modeled the reference counts of target objects and used a model-checking tool to verify the correctness of the properties and hunt for bugs. And Férey et al. provided PVS2C [47] to convert a high-level language to C for runtime efficiency with the help of the reference counting approach to manage dynamic memory deallocations.

Native Code Analysis. Tan et al. proposed an approach to detecting the Python code that may lag the program execution [48]. They mapped the memory addresses and their operations in binary libraries to the original Python code and searched for efficiency issues via memory operations. Hu and Zhang presented their research on the evolution of Python/C API usage [9]. They also provided the error patterns they found during the research. Monat et al. create Mopsa [49], which statically scans Python exceptions and C runtime errors for a Python program using binary extension modules. They use an abstract interpretation based approach to analyze both the Python and C code by sharing abstract domains between the two languages. However, they cannot check refcount bugs.

Besides, Kondoh and Onodera checked the Java native code for memory errors of four patterns with a typestate analysis approach and a coding style checker [50]. Li and Tan focused on finding mishandled Java exceptions thrown in native code in their research [51], [52]. Brown et al. attempted to report bugs for the intermediate binding layer between JavaScript and C++ [53]. Kalubandi et al. also focused on this problem [54]. They tried to prevent JavaScript exceptions and C++ runtime crashes by checking the type errors. And Degenbaev et al. proposed an approach to tracking the objects crossing the borderline to JavaScript heap and help developers to find potential memory errors [55].

VIII. CONCLUSION AND FUTURE WORK

In this paper, we proposed an approach to tracking the lifecycle of PyObject with the help of its refcount manipulations to detect refcount bugs in native code. We defined the lifecycle states and transition rules in the novel model. Beyond the model, we also track the changes in lifecycle state and refcount triggered with pointer operations and refcount monitors. We implemented the PyRefcon analyzer based on the approach and found 259 real bugs from 12 open-source projects.

In the future, we will continue our research on checking the Captured PyObjects, and improving the efficiency of the approach with summary-based strategies, and automatic approaches to searching for refcount monitors.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (NSFC) under grant No.62132020. Many appreciations to Mr. Xin Zhang for his discussion on the idea of this paper, Mr. Shuo Sun for gathering the data, the developers of project NumPy for their time in feedback on our bug reports, and anonymous reviewers of this paper for their suggestions for improving this work.

REFERENCES

[29] “PyCrypto,” f37bd12db3b887d5457232c3088f9445630b4cc, version 0.2.8. [Online]. Available: https://people.csl.illinois.edu/hubert/pcrypto
[31] “pycrypto,” commit 040ab49d3d4d4d21fe0cbe725f430ab86c8e2e. [Online]. Available: https://github.com/pycryptodome/pycrypto