PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing

Wen Li
Washington State University
Pullman, WA, USA
li.wen@wsu.edu

Haoran Yang
Washington State University
Pullman, WA, USA
haoran.yang2@wsu.edu

Xiapu Luo
The Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Long Cheng
Clemson University
Clemson, SC, USA
lcheng2@clemson.edu

Haipeng Cai*
Washington State University
Pullman, WA, USA
haipeng.cai@wsu.edu

ABSTRACT

Given the widespread use of Python and its sustaining impact, the security and reliability of the Python runtime system is highly and broadly critical. Yet with real-world bugs in Python runtimes being continuously and increasingly reported, technique/tool support for automated detection of such bugs is still largely lacking. In this paper, we present PyRTFuzz, a novel fuzzing technique/tool for holistically testing Python runtimes including the language interpreter and its runtime libraries. PyRTFuzz combines generation- and mutation-based fuzzing at the compiler- and application-testing level, respectively, as enabled by static/dynamic analysis for extracting runtime API descriptions, a declarative, specification language for valid and diverse Python code generation, and a custom type-guided mutation strategy for format/structure-aware application input generation. We implemented PyRTFuzz for the primary Python implementation (CPython) and applied it to three versions of the runtime. Our experiments revealed 61 new, demonstrably exploitable bugs including those in the interpreter and most in the runtime libraries. Our results also demonstrated the promising scalability and cost-effectiveness of PyRTFuzz and its great potential for further bug discovery. The two-level collaborative fuzzing methodology instantiated in PyRTFuzz may also apply to other language runtimes especially those of interpreted languages.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Theory of computation** → **Program analysis**.

KEYWORDS

Runtime system, Python, language runtime, fuzz testing, greybox fuzzing, collaborative fuzzing, code generation, software security

*Haipeng Cai is the corresponding author.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623166

ACM Reference Format:
Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623166

1 INTRODUCTION

With its versatility and ease of use, Python has for years been one of the most popular programming languages [21, 25, 26, 40]. It is widely used in various domains, ranging from artificial intelligence and data science to web development and scientific computing. In particular, the significance of Python is highlighted in recent years by its pivotal role in building and deploying machine learning systems [46]. Like other Python applications, these systems have their reliability and security heavily and *broadly* depend on that of the environment they run in—the Python runtime, which consists of the *interpreter* and *runtime libraries* of the language. Given the prevalent of Python and the breadth of its software ecosystem, any bugs in the Python runtime would have widespread impact and critical consequences.

Unfortunately, this concerning premise about potentially buggy Python runtimes has been shown to be true: real-world reports on bugs in the Python runtime are prevalent and growing. For instance, the last five years have witnessed nearly 2,000 bugs in CPython [41], the most widely used implementation of Python, annually as per our recent manual studies. Notably, 10% of these bugs have serious security consequences that have been disclosed. Thus, it is essential to thoroughly test the Python runtime. Since manual analysis is clearly undesirable, automated tool support is needed. However, such tools are largely lacking as it stands.

Relevant automated software testing tools do exist. For instance, various fuzzing techniques have been developed for compiler and language runtimes (e.g., JavaScript engines and JVM). Examples include generation-based fuzzers like JSfunfuzz [45], TreeFuzz [37], and Skyfire [48] which learn grammatical features and rules of respective languages from existing samples to generate valid test programs, as well as mutation-based fuzzers like Superion [49] and Fuzzil [15] which generates tests by mutating seed programs via AST editing or an intermediate language with grammar awareness.

Other grammar-awareness approaches (e.g., for JVM testing [7, 8]) have also been developed.

These techniques have proven effective in detecting bugs in their targeted runtime systems. And fuzzing is a successful and promising approach in general—in fact, it has become the primary methodology for bug detection. However, existing relevant approaches are not sufficient for testing Python runtimes due to several *key challenges to Python runtime fuzzing* that they have not addressed.

First, like that of a typical language, the runtime of Python consists of the interpreter core and the language's runtime libraries. Thus, testing the Python runtime requires testing both of these two integral parts. Indeed, our recent studies on real-world CPython bugs show those bugs exist in both parts. Moreover, unlike for the runtime of compiled languages such as C/C++ and Java, focusing just on generating test programs is insufficient for testing Python runtimes because the interpreter's behaviors are mostly exercised only during the execution of Python applications when they interact with the interpreter through various runtime APIs. Thus, fuzzing a Python runtime additionally requires fuzzing the generated test programs (i.e., Python applications) as well, for which concrete application (or APP in short) test inputs need to be generated. As a result, holistically fuzzing Python runtimes would necessitate two *different levels* (i.e., generating programs versus concrete program inputs) of fuzzing *closely collaborating together* (i.e., for extensively exercising the interactions between the interpeter and runtime libraries). However, there is little prior knowledge on how to design such a two-level, collaborating fuzzing technique (Challenge 1). **Second**, fuzzing the interpreter itself would need diverse yet (syntactically and semantically) valid Python APPs. Yet how to achieve these two requirements at the same time is largely unknown especially for Python (Challenge 2). **Third**, it is general challenging for an application fuzzer to generate quality input values, even more so for Python application fuzzing because data types are not available in the program of a dynamically-typed language like Python, making it difficult to achieve format/structure-aware input generation which is important to effectively fuzz Python APPs (Challenge 3).

To address these challenges, we have developed PyRTFuzz, a *two-level collaborative* fuzzing technique for comprehensive testing of Python runtimes. PyRTFuzz combines generation- and mutation-based fuzzing, at Level-1 and Level-2, respectively, with the two fuzzers *collaborating in synergy* and *working in a holistic fuzzing loop* as a singleton based on shared coverage feedback to address *Challenge 1*. In particular, the Level-1 fuzzing generates valid and diverse Python APPs of various levels of control flow complexities through a novel extensible *specification-based* Python code generation method based on the descriptions of each API in the Python runtime extracted using static and dynamic analysis, addressing *Challenge 2*. The APP-generation capability is also enabled by a specialized declarative *specification language* whose syntax and semantics are defined via a set of *APP-generation primitives* that capture common Python language features and real-world Python application programming patterns. At Level 2, the mutation-based fuzzing in PyRTFuzz instruments the given Python runtime for collecting the coverage feedback and runs the instrumented runtime against an APP selected by the Level-1 fuzzer, while generating concrete application inputs to feed the APP through a *custom mutation* strategy that can produce input values in a *type-guided*, format/structure-aware manner by utilizing the runtime API descriptions, hence addressing *Challenge 3*.

We have implemented PyRTFuzz based on Atheris [14] and libFuzzer [30]. To evaluate its effectiveness, we tested PyRTFuzz on three different versions of CPython [41]: `Python 3.7.15`, `Python 3.8.15`, `Python 3.9.15`. In addition to measuring its bug-detection capability, we also evaluated the scalability of its Python application generation across various APP specification sizes, ranging from a minimum of 1 to a maximum of 4096. Furthermore, we investigated several factors that may affect the effectiveness of PyRTFuzz, including APP specification size, level-2 time budget, and typed API descriptions. Within a total budget of 5×24 hours, PyRTFuzz detected a total of **61** bugs across the three Python versions, including **25** in Python 3.9.15, **15** in Python 3.8.15, and **21** in Python 3.7.15. Our results also demonstrated monotonic growth of coverage of the runtime code, suggesting that greater coverage may be attained and potentially even more bugs may be discovered if the fuzzing continues. Regarding the scalability of application generation, we found that both the time and memory expense of PyRTFuzz have linear correlations with the APP specification size, and PyRTFuzz can generate 4.77 KLoC of code in 40 minutes with a memory usage of 291.71 MB. Finally, our ablation studies revealed significant impacts of APP specification size and Level-2 per-APP fuzzing time budget on PyRTFuzz's cost-effectiveness and that the runtime API descriptions with type information contributed significantly to PyRTFuzz's performance.

To the best of our knowledge, PyRTFuzz is the first two-level collaborative fuzzing technique for Python runtime testing. In addition, its open-source implementation and extensible design can also facilitate the development of greybox fuzzing for other compiler and runtime systems. Particularly, the methodology of combining generation-based compiler fuzzing and mutation-based application fuzzing as instantiated in PyRTFuzz can be more broadly applied to interpretation languages beyond Python.

Open science. Source code of PyRTFuzz and our experimental datasets are all available in our artifact package and has been made publicly accessible.

2 BACKGROUND AND MOTIVATION

In this section, we provide a brief background on greybox fuzzing and discuss its relevance to compiler testing. We also discuss the challenges and problems that arise in Python runtime testing, which motivated us to develop our PyRTFuzz approach.

2.1 Greybox Fuzzing

Greybox fuzzing [3, 10, 31, 33, 44] is a software testing methodology that strikes a balance between white- and black-box fuzzing, and the most commonly adopted approach is *coverage-guided* greybox fuzzing [1, 14, 28, 33]. Regarding the strategy for generating input values, greybox fuzzing can be divided into two categories: *mutation-based* and *generation-based* [31].

Mutation-based fuzzing generates new test cases by randomly modifying or mutating existing ones. The modified test inputs are then used to exercise the target application for unexpected behaviors such as crashes [44]. In contrast, generation-based fuzzing generates test cases from scratch based on predefined input grammar

or input-generation rules. This approach requires understanding the application's input specification to ensure the generated inputs are semantically valid and meaningful. Although mutation-based fuzzing is simpler to implement, generation-based fuzzing could be more effective in discovering deep bugs in complex applications.

2.2 Compiler Testing

A key challenge with compiler testing is to generate valid, diverse test programs [5]. Generating invalid programs may not be useful as they may get discarded in the early (e.g., preprocessing) stages of the compiler, while syntactically diverse programs can help exercise different parts of the compiler, hence potentially increasing code coverage and uncovering bugs.

To overcome the challenge, two primary approaches exist: grammar-guided and program mutation-based [5]. Grammar-guided approaches use the formal grammar of a programming language to generate valid programs [2, 18, 39, 47, 52]. In contrast, program mutation-based techniques modify parts of an existing test program while either preserving the program's semantics [22, 23, 29] or changing them [8, 34, 35]. The choice of technique depends on several factors, such as the programming language, type of compiler, and testing goals, each with its pros and cons.

2.3 Python Runtime Fuzzing

Our work in this paper is motivated by not only the prevalence and impact of the Python runtime as generally perceived, but more by what we observed in our study on its historical bugs. We first present key findings from the bugs we analyzed manually and then discuss key challenges to automatically detecting such bugs.

Empirical Study on CPython Bugs. CPython [41] is the most widely used implementation of the Python programming language, and it has been actively maintained for more than two decades. To understand the nature of bugs in CPython, we collected 98.3K historical issues from its source repository, of which we found 23.4K are *bug-related* through a comprehensive analysis. In particular, we revealed that since 2008, more than 1,000 bug-related issues have been reported annually, and the number of bugs reported per year has consistently remained close to 2,000 in the last five years (as shown in Figure 1). To understand how these bugs were detected, we manually analyzed a random sample of 500 bug-related issues. We found that *over 98% of the bugs were triggered by developers during Python application development*, highlighting the need for effective testing tools to ensure the quality of CPython and the opportunity of *testing the language runtime via its applications*.

We then analyzed the distribution of these bugs across different modules in CPython. Our analysis revealed that most bugs (86.8%) occurred in the Python runtime libraries, while the remaining 13.2% occurred in the Python interpreter core. Furthermore, out of 165 modules extracted from the CPython source code, 164 modules were found to have reported bugs. That is, bugs can occur in various Python runtime libraries and the interpreter core, underscoring the importance of thoroughly testing *both* hence the need for *automated* techniques to *test the Python runtime as a whole*.

Python Runtime Fuzzing. As in traditional compiler testing, generating diverse and valid Python applications is crucial for Python runtime testing. However, merely applying existing techniques

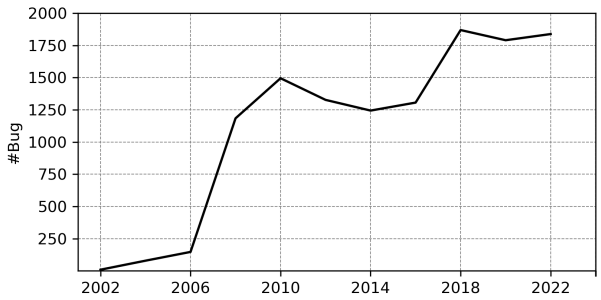

Figure 1: The number of bugs over twenty years in CPython.

```
# program A as seed
n = 4
s = [0,3,5,8,7,9]
for i in range (0, n):
  s[i] = n
  s[n] = i
```

```
#program B generated from A
n1 = 4
s1 = [0,3,5,8,7,9]
for n2 in range (0, n1):
  for n3 in range (0, n2):
    s1[n3] = n2
    s1[n2] = n3
```

Figure 2: Example of grammar-based code generation.

may not be sufficient. As illustrated in Figure 2, a valid program B can be generated using an arbitrary number of code bricks extracted from program A through a state-of-the-art grammar-based approach (e.g., CodeAlchemist [17]). While these generated programs are both semantically and syntactically correct, such testing approaches suffer three significant limitations, among others:

- L1: Unawareness of diverse domains: Behind the empirical results above, most of the bugs occurred in diverse Python runtime libraries that covered 165 domains. Yet existing approaches focus on generating code itself on top of input seeds (as shown in Figure 2) without paying sufficient attention to how these runtime APIs are used. Thus, unawareness of this *diversity* can lead to missed opportunities to detect bugs.
- L2: Insufficient application inputs: While covering the various domains (via diverse Python applications) is important, it is not enough for effectively testing the Python runtime. Even with unit tests being available, the recent growing CPython-bug reports suggest that these tests alone are also insufficient, because they cover only a *limited* space of possible *inputs* to the runtime. As depicted in Figure 2, when using existing compiler testing techniques [17], the newly generated program B will be executed only once with no varying inputs. This limitation hinders the ability to uncover potential bugs that may arise with different input scenarios.
- L3: Lack of holistic testing: Python applications run in the environment of Python runtime, which comprises the interpreter core and runtime libraries. Testing only one of these two parts is inadequate for thorough testing. A *comprehensive* approach to testing the Python runtime should address both the interpreter core and runtime libraries as well as *interactions* between the two.

To illustrate further, consider the example in Figure 3, where the API `locale.format` formats the integer 2 with an input formatter `percent`. This application has three inputs as shown on the

```
import locale
def localeTest(percent):
    try:
        ret = locale.format(percent, 2)
    except:
        pass

if __name__ == '__main__':
    localeTest (sys.argv[1])
```

[input A]: "%2u"
[output]: " 2"
[bug type]: None

[input B]: "%777777777777777u"
[output]: MemoryError in locale.py
[bug type]: Unhandled Exception

[input C]: "%7777777777u"
[output]: Hard crash
[bug type]: Out of Memory

Figure 3: Motivating example: bugs occur in the interpreter and runtime APIs of the Python runtime.

right, leading to three different results. Input A led to a normal run, while against input B the API `locale.format` fails to handle the `MemoryError` exception that the interpreter throws when trying to construct a large-sized string, resulting in an `Unhandled Exception`. Input C led to a hard crash due to `Out of Memory`, caused by missing validation before memory allocation in the interpreter; in this case, a `MemoryError` exception should have been thrown.

This example illustrated an important point: if we do not generate applications for a particular runtime module such as `locale`, bugs within that module may not be triggered otherwise (limitation L1). Further, even generating applications of diverse domains may not be sufficient to detect all bugs if not considering various inputs to the applications (limitation L2). Finally, the different results demonstrate interactions and collaborations between the interpreter core and runtime libraries, highlighting the importance of testing them holistically (limitation L3).

Motivated by the above empirical results and observations, we propose a two-level collaborative fuzzing approach to achieve thorough testing of the Python runtime. This approach consists of (Level-1) generation-based fuzzing for Python application generation and (Level-2) mutation-based fuzzing to generate various application inputs. Together, the two fuzzers collaborate closely at the two levels to detect Python runtime bugs comprehensively.

3 TECHNIQUE DESIGN

This section describes our two-level collaborative fuzzing approach to *Python runtime* testing. We first give an overview of our approach, summarizing the high-level workflow of PyRTFuzz. Then, we elaborate the process of extracting API descriptions (§3.2), following by the two *collaborating* fuzzing phases: generation-based fuzzing at Level 1 (§3.3) and mutation-based fuzzing at Level 2 (§3.4).

3.1 Design Overview

The overall design of PyRTFuzz is delineated in Figure 4. The three **PyRTFuzz inputs** consist of the (source code of the) *Python runtime* including (1) the *interpreter* and (2) its built-in libraries (*runtime libs*), and the *unit tests* as part of the runtime's source package.

With these inputs, PyRTFuzz tests the given Python runtime *through its APIs*, as justified by (1) these APIs are an essential part of the runtime and (2) the interpreter's behaviors are exercised as it interacts with Python applications (APP) via the APIs. To that end, PyRTFuzz first extracts the description of each runtime API in **Phase 1**. These API descriptions, each including the key metadata (e.g., enclosing module/class and the type of each parameter) of an API, enable both the API-centered APP generation for the Level-1 (**L1**) fuzzing and the type-guided input generation during the Level-2 (**L2**) fuzzing. In particular, this phase starts with statically extracting the metadata without resolving types (e.g., of each API's return and parameters), resulting in the *untyped API description*. The description is then refined (i.e., with the types getting resolved) by running the unit tests, leading to the *typed API description*.

Using the (typed) API descriptions, PyRTFuzz now enters the iterative process of two-level fuzzing closely collaborating with each other. During Level-1 fuzzing in **Phase 2**, PyRTFuzz aims to generate test Python APPs using a declarative *specification language* (noted as SLang) based on a set of pre-defined generation *primitives* derived per the interpreter. Then, this phase starts with generating the specification of an APP as triggered by the L1 fuzzer core on demand. Next, from the resulting *APP specification* while referring to the *SLang primitives*, our SLang compiler generates the *Python APP* by translating the specification to respective Python code. With this capability, the L1 core generates an APP around each API of the given runtime and initializes the *APP queue* with all such per-runtime-API Python APPs. Once the collaborative fuzzing loop sets off, the core selects an APP, at random initially (when no coverage feedback is available yet) and per the feedback later on, and feeds it to Level-2 fuzzing (**Phase 3**). To balance between the depth of fuzzing around each API and the breadth of fuzzing in terms of exercising all the APIs, the L1 core schedules the Level-2 fuzzing of each APP for a budgeted period of time. Once the budget runs out, the L1 core decides if to generate another (more complex) APP for the same API as in the previous APP, triggers the APP-generation if so, and selects the next APP for Level-2 fuzzing, all based on the coverage achieved in the previous Level-2 fuzzing iteration.

The Level-2 fuzzing in **Phase 3** aims to exercise a given Python APP under the time budget following a mutation-based application fuzzing strategy. In particular, this phase starts with instrumenting the entire Python runtime for coverage monitoring. Then, during the collaborative fuzzing loop, for each APP received, the L2 fuzzer core runs the APP with the *instrumented runtime* against concrete input values (i.e., the arguments feeding the call to the underlying API in that APP). Given the differences among the incoming APPs, the L2 core maintains a *per-APP seed queue* initially populated by seeds randomly generated. Later on, new input values are obtained by mutating the seeds based on the collected coverage feedback, through a custom mutation scheme. The custom mutator can produce values that fit the input format for each particular APP, as enabled by probes inserted to the APP during the APP generation in **Phase 2**. These probes aim to decode a byte sequence into individual argument values as per the (typed) description of the API called in the APP. Any triggered bugs, along with the triggering seeds, are then produced as the **PyRTFuzz outputs**.

3.2 Runtime API Description Extraction (Phase 1)

While the APIs in the Python runtime are described in Python's official documentation, those descriptions as in the unstructured format are not immediately amenable for testing the runtime. The descriptions could be manually put together in more structured form, yet this manual process is clearly undesirable given its tedious

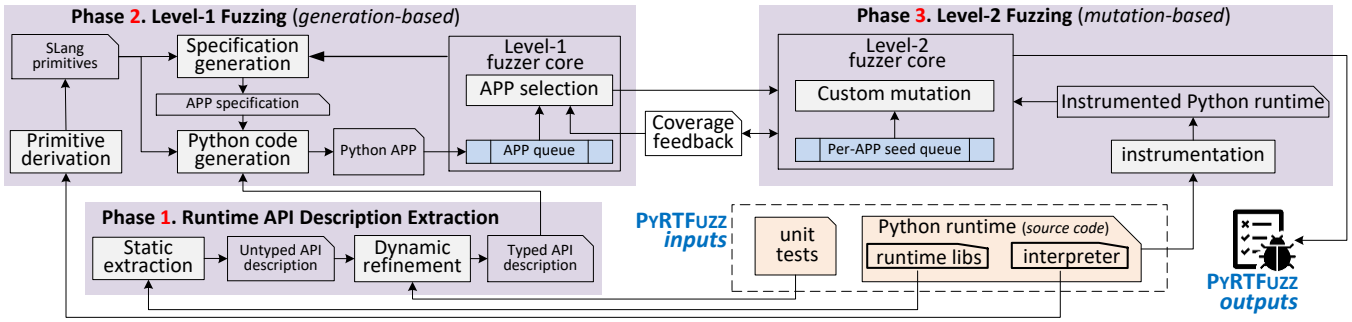

Figure 4: An overview of PyRTFuzz's design, including its inputs, three working phases, and outputs.

nature and unscalable given the constant evolution of the language as well as numerous versions of its implementation. Thus, it is essential to extract the descriptions in an *automated* manner.

3.2.1 Static Extraction. Using the standard AST parser of Python, PyRTFuzz first statically parses the runtime libs to extract API descriptions. A runtime API description in PyRTFuzz is the API's metadata essential for our two-level fuzzing, as defined in Table 1.

Table 1: Definition of *runtime API Description* in PyRTFuzz

Field	Type	Field Description
module	string	name of API's enclosing module
class	string	name of API's enclosing class if any
name	string	canonical name of the API
parameters	{string:string}	dictionary of parameter and types
returns	{string:string}	directory of returns and types
exception	[string]	list of exceptions that may be thrown

All the fields defined are essential for generating executable Python applications. Specifically, to call an API, its enclosing `module` must be imported. If the API is defined within a class, then the `class` must be known also since an object of the class must be created to call the API from. The `name`, `parameters`, and `returns`, including the type of each parameter and each return value, are intuitively important to know in order to generate a Python APP around a call to the API. Given the `name` of an API, all of its arguments are extracted as a dictionary where the key is parameter name and the value is the parameter type. Also, a pair `"...:..."` is appended to the dictionary if the API has a variable number of parameters. Given the inaccuracy of static type inference [19, 32, 38], the types are set as `None` for now and will be refined next. Thus, the static extraction ends with untyped API descriptions.

For the `exception` field, both explicit exceptions possibly thrown by the API or implicit ones thrown from the imported modules are collected as comprehensively as possible—this information guides the exception handling of the API call during APP generation (§3.3), which is essential for avoiding exceptions thrown during the Level-2 fuzzing being spuriously identified as bugs.

3.2.2 Dynamic Refinement. Given the *untyped API description* of a runtime API as input, PyRTFuzz then runs the given unit tests. From the test executions, it extracts the parameter/return types of the API as outlined in Algorithm 1, hence refining the untyped description to produce the typed API description.

Algorithm 1: Dynamic refinement of API descriptions

```
Input: U: unit test set
Input: utDescs: untyped API descriptions
Output: tDesc: typed API description
1  Function dynamicRefinement (U, utSpec)
2      foreach u_i in U do
3          F ← dynamicInspect(u_i);          //get all functions' frames via Python's Inspect
4          foreach f_i in F do
5              N ← getAPIName(f_i);
6              Desc ← getAPIDescription(N, utDescs);
7              if Desc is None then
8                  continue;
9              foreach param in Desc.parameters do
10                 tDesc ← updateParameterType(Desc, param, f_i);
11             tDesc ← updateReturnType(f_i);
12     return tSpec;
```

The algorithm processes one unit test at a time (line 2). For each test, it inspects the frame objects of all APIs exercised during the test execution (line 3), which local variables (e.g., parameters and return values) are captured. Then, from each frame f_i, the API name is extracted (line 5) and used to retrieve the corresponding untyped API description (line 6). If the retrieval succeeds, the parameter names are taken as the keys to further identify the respective arguments (live objects) from the frame f_i; then the parameter types are extracted. Next, the return type is also extracted (line 9–11).

Compared to static type inference, the dynamic analysis here can provide high precision but may suffer from false-negative issues due to the incomplete coverage of the test executions. Also, given Python's dynamic nature, each API's parameter/return types may vary from one execution to another. Nevertheless, getting at least one concrete type instance of each parameter and return suffices for generating valid APP code and application inputs in PyRTFuzz.

3.3 Level-1 Fuzzing (Phase 2)

To fully fuzz the Python runtime, PyRTFuzz aims to generate valid and diverse Python applications around the APIs in the runtime (libraries). Specifically, this aim subsumes three requirements:

R1: API coverage PyRTFuzz should generate applications such that all the different runtime APIs are considered hence covered. This helps test the runtime as comprehensively as possible.

R2: APP diversity For each API, PyRTFuzz should generate a diverse set of applications so that various use scenarios of the API can be tested. This helps detect bugs in the runtime as far as possible, and one way to achieve the diversity is to let the applications have at least diverse *control flow complexity*.

R3: APP validity Each application generated needs to be syntactically and semantically valid since invalid ones would be quickly rejected by the runtime before its deep logic may be exercised during fuzzing. One aspect of the validity is *data flow reachability*: the data should flow from the application entry to the API callsite.

PyRTFuzz meets these requirements through *SLang*, a simple declarative specification language, and specification-based Python APP generation via SLang whose syntax and semantics are defined through a set of generation *primitives* as the lowest-level language constructs. In particular, the syntax of SLang is formulated below:

$$\begin{aligned} P &::= S^* \\ S &::= [c =]C(e)^* \\ e &::= c \mid A \end{aligned}$$

A SLang program (i.e., an APP specification) P is a sequence S^* of statements. A *statement* S is defined with only one type: `assignment`. In each `assignment`, the right-value $C(e)^*$ represents a primitive C taking an expression e; and the left-value c represents the result of $C(e)^*$. An expression e here is one of two kinds: a variable c or a call to a runtime API A.

3.3.1 Primitive Derivation. The SLang primitives were derived according to the Python language reference [43] while referring to the interpreter implementation. We aim to identify the most essential primitives based on the common features of the Python language so that these primitives do not vary with various Python implementations (e.g., varying versions of the interpreter). With respect to **R2**, these primitives mainly (albeit not only) summarize the key control flow structures of Python hence falling in two categories: `basic` and `extend`, as elaborated as follows.

`Basic`. A *Basic* primitive only takes a call to a runtime API as input. It is used to construct the basic (control-flow) structure of a Python program, which may be object- or procedure-oriented, and define the program's entry point. Thus, a SLang program should start with one and only one `basic` primitive.

Algorithm 2: Operational semantics of a `Basic` primitive

Input: *API*: a runtime API description
Output: *P*: a Python APP
1 **Function** `basicPrimitive` *(API)*
2 $T \leftarrow$ `typeList` (API); //probe for API parameter type list
3 P.insert (T);
4 $D \leftarrow$ `newFunction` (demoFunc, demoParam);
5 $p_d \leftarrow$ `getParameters` (D);
6 args $\leftarrow$ `decodeArguments` (T, p_d); //probe for decoding API arguments
7 $s_d \leftarrow$ `newCall` (`API.name`, args);
8 D.insert (`wrapException` (s_d));
9 P.insert (D)
10 $M \leftarrow$ `newFunction` $(P,$ `SLmain`, mainParam);
11 $p_m \leftarrow$ `getParameters` (M);
12 $s_m \leftarrow$ `newCall` (demoFunc, p_m);
13 M.insert (s_m);
14 P.insert (M)
15 **return** P;

Algorithm 2 outlines the general operational semantics of a `basic` primitive (in pseudo code)—as in other algorithm pseudo code in our paper, we use the special (e.g., cyan) color to highlight the key algorithmic steps. The algorithm uses as input the description of the runtime API taken by the primitive and initializes the program P with a probing for the API's parameter type list T (line 3). Next, it creates a `demoFunc` to wrap the invocation of the API (line 4–8). To enable the input format-aware mutation (§3.1), a function called `decodeArguments` is inserted to probe for decoding the input (which is an encoded byte stream from the L2 fuzzer) into individual arguments of the API call (line 6). This ensures the type correctness of arguments. Also, it is important to wrap the invocation of the API in a `try-except` block to filter out expected exceptions hence allowing the fuzzer to capture unhandled exceptions as bug indicators. Then, the definition of `demoFunc` is added to the APP P.

Next, the main function `SLmain` of P is created (line 10–13). Like *demoParam*, *mainParam* is a single parameter used for passing the APP input eventually to the API callsite. Then, a call is added for `SLmain` to invoke `demoFunc` with the same parameters of `SLmain` as arguments, so as to ensure the data-flow reachability in **R3**. Finally, P adds `SLmain` as its entry point to facilitate Level-2 fuzzing.

`Extend`. An `Extend` primitive takes the results of other primitives as inputs. It is used to diversify an APP while increasing its complexity. Derivation of these primitives was additionally informed by programming patterns we observed in real-world Python software.

Algorithm 3: Operational semantics of an `Extend` primitive

Input: P: a previously generated Python application
Output: P': a new Python application
1 **Function** `extendPrimitive` *(P)*
2 $M \leftarrow$ `getMain` (P); //get SLmain
3 $B_M \leftarrow$ `getBody` (M);
4 $p \leftarrow$ `getParameters` (M);
5 $Block \leftarrow$ `newBlock` (B_M, p); //wrap SLmain's body into a new block
6 $P' \leftarrow$ `setBody` $(M, Block)$;
7 **return** P';

Algorithm 3 describes the general operational semantics of an `extend` primitive, which generates new applications via wrapping the input program P in a top-down fashion. The algorithm retrieves the main function M of P (i.e., `SLmain`) (line 2) along with its body (line 3) and parameters (line 4). Next, the primitive wraps M's body as the body of a new (e.g., `for` or `if`) block (line 5) and then replaces M's body with this new block to generate a new program P' (line 6). Such a generation process is intraprocedural—no new function is generated. The process can be interprocedural as well, during which a new function with the same parameters (p) as those of M is generated, M's body is inserted into the new function, and the body of M is replaced by an invocation of the new function.

Table 2: SLang primitives

No	Command	Category	Comment
1	OO	`Basic`	object-oriented program
2	PO	`Basic`	procedure-oriented program
3	While	`Extend`	a while structure wrapper
4	For	`Extend`	a for structure wrapper
5	If	`Extend`	a if structure wrapper
6	Call	`Extend`	a call structure wrapper
7	With	`Extend`	a with structure wrapper

Specifically, we derived seven primitives as shown in Table 2, which are realized on top of relevant Python abstract syntax tree

(AST) operators. Each primitive can operate on any Python runtime API while not dependent on the syntax or semantics of a specific API, which allows for generating APPs to cover all of the APIs hence satisfying requirement **R1**.

Note that these specific primitives induce both intra- (e.g., *If*) and inter-procedural (e.g., *Call*) control flows. The rationale is to diversify the ultimate control-flow complexity of the generated APP for satisfying requirement **R2**.

The design of both categories of primitives follows a top-down wrapping approach where posterior primitives operate only on the body of `SLmain` of prior ones. For primitives that induce interprocedural control/data flows, this approach also allows for seamless data transfer from the top (i.e., `SLmain`) to the bottom (i.e., the API call site) via parameter passing. In all, the design ensures the validity of generated APPs, satisfying requirement **R3**.

```
1  OO (sqlite3.dbapi2.DateFromTicks)
2  ----------------------------------------
3  TypeList = ['ticks:int']
4  class demoCls():
5      def demoFunc(self, p):
6          try:
7              ticks = decodeArguments(TypeList, p)
8              sqlite3.dbapi2.DateFromTicks(ticks)
9          except (AssertionError) as e:
10             pass
11 def SLmain(x): # entry point
12     dc=demoCls()
13     dc.demoFunc(x)
```

Figure 5: Example illustrating the *OO* primitive's semantics.

Figure 5 shows an example of applying the OO primitive to a runtime API `sqlite3.dbapi2.DateFromTicks` (line 1), along with the corresponding Python code (lines [3–13]) of equivalent semantics. The variable `TypeList` denotes the name and type of the API's parameters. The OO primitive creates a new class named `demoCls` and wraps the API inside its method `demoFunc` (lines [6–10]). Specifically, a new variable `ticks` is defined to take the result of decoding the input parameter `p` (line 7). The call site for the API is then created and wrapped inside a `try-except` block. Finally, the entry point `SLmain` is created to form a calling context of `demoCls.demoFunc`.

3.3.2 Specification Generation. As defined in §3.3.1, an APP specification is composed of a sequence of SLang statements. PyRTFuzz generates such a specification by sequentially constructing SLang statements with the primitives, as outlined in Algorithm 4. The algorithm takes three inputs: the full primitive set *SPS*, a runtime API's name *API*, and the number N of statements to generate.

The generation procedure includes two primary steps:

(1) Constructing the statement on a `Basic` primitive (line 2–3) : First, the primitive C_b is randomly selected from *SPS* (line 2). Then, an assignment statement is constructed in the form of $R = C_b(API)$. R is a stack for storing the results of C_b on *API*. As the first statement of the specification, it defines the overall structure of the target Python APP that uses *API*.

(2) Constructing statements on `Extend` primitives iteratively according to N (line 2–3): In each iteration, a statement $R = C_e(R)$ is generated with a randomly selected `Extend` primitive C_e. C_e takes R (the result of the previous statement) as input, and then stores the operation result back to R.

Algorithm 4: Generation of an APP specification in SLang

```
Input: SPS: SLang primitive set
Input: API: Python runtime API name
Input: N: number (≥ 1) of statements to generate
Output: SS: SLang script
1  Function genSpecification (SPS, API, N)
2      C_b ← selectPrimitive (SPS, basic);        //randomly select a Basic primitive
3      S ← genStatement (R, C_b, API);            //statement: R = C_b (API)
4      pushBack (SS, S);
5      N ← N − 1;
6      while N > 0 do
7          C_e ← selectPrimitive (SPS, extend);   //randomly select an Extend primitive
8          S ← genStatement (R, C_e, R);          //statement: R = C_e (R)
9          pushBack (SS, S);
10         N ← N − 1;
11     return SS;
```

```
1  R = OO (sqlite3.dbapi2.DateFromTicks)
2  R = For (R)
3  R = Call (R)
```

Figure 6: An example APP specification of three statements.

As an example, with API `sqlite3.dbapi2.DateFromTicks`, Figure 6 shows a randomly generated SLang-based APP specification consisting of three statements. The first statement uses a `Basic` primitive, namely *OO*, to generate an object-oriented program for the API and stores the result in the stack R. The second statement applies an `Extend` primitive, namely *For*, on R. It wraps a `for`-structure around the code in R and stores the result back in R. Similarly, the third statement uses another `Extend` (*Call*) primitive to wrap a call structure around the current code in R and pushes it back to R. Finally, R stores the Python code corresponding to the entire example specification here.

3.3.3 Python Code Generation. With an APP specification as input, the SLang compiler parses the specification (SLang code) and translates each statement in sequence to corresponding Python code according to the operational semantics of the primitive in that statement. This code generation process is outlined in Algorithm 5.

Firstly, the algorithm initializes the stack R and the API description *Desc* as None (line 2–3). Then, it loads all of the SLang statements into the memory region denoted as L_s (line 4), and then parses and compiles the statements in sequence (line 5–13). For each statement, the compiler retrieves the primitive *Prm* and corresponding input *Arg* (line 6). If *Arg* is R, the compiler loads the value of R as input and then compiles *Prm* (line 9); in this scenario, R stores the program generated by the previous statement, hence the continuous superposition of the next primitives can lead to increasingly complex Python code. When the input *Arg* is an API description, the compiler retrieves the description and stores it into *Desc* (line 11) as input for compiling the primitive *Prm*. The compilation result is then pushed to the stack R (line 12). When all statements are compiled (translated), the compiler retrieves the `Import` information from the API description *Desc* and inserts it to R, hence producing the final Python APP *PyApp* as output (line 14).

Figure 7 shows an example of compiling (translating) a SLang-based specification (as shown in Figure 6) to a Python APP. Overall,

Algorithm 5: Specification-based Python code generation

```
Input: SS: SLang-based specification
Input: PyDescs: Python runtime API descriptions
Output: PyApp: resulting Python APP
1  Function generatePythonAPP (SS, PyDescs)
2      R ← None;                                   //Initialize the stack R as None
3      Prm ← None;                                 //Initialize the API description as None
4      L_S ← parseSpecification (SS);              //parse the specification and get the SLang statement list
5      foreach s in L_S do
6          Prm, Arg ← parseStatement (s);
7          if Arg is 'R' then
8              R ← loadR ();
9              R ← compilePrimitive (Prm, R);
10         else
11             Desc ← getDescription (PyDescs, Arg);
12             R ← compilePrimitive (Prm, Desc);
13         storeR (R)
14     PyApp ← import (Desc, R);                   //import all dependent modules
15     return PyApp;
```

each primitive encapsulates the input program in a top-down manner. For the first statement with a `Basic` primitive `00`, a simple program is generated as shown in Figure 5. Then, in the second statement, with `For` the compiler wraps all the statements of `SLmain` (the top) into a `for` loop and embeds the `for` block back into `SLmain`. Similarly, in the third statement, with `Call` the compiler wraps all the statements of `SLmain` into a new function, `PyCall_1681926341` (where the number suffix is randomly generated simply for unique function naming), and embeds its invocation with input `x` back into `SLmain`. Note that the `Call` primitive induces inter-procedure control flow. Both `PyCall_1681926341`'s arguments (on Line 16) and parameters (on Line 11) are assigned the same variables for `SLmain`'s parameters, to ensure the data flow reachability in **R3**. Finally, *import* information is added to the resulting APP.

```
1  from pyrtfuzz import decodeArguments
2  import sqlite3
3  TypeList = ['ticks:int']
4  class demoCls():
5      def demoFunc(self, p):
6          try:
7              ticks = decodeArguments (TypeList, p)
8              sqlite3.dbapi2.DateFromTicks(ticks)
9          except (AssertionError) as e:
10             pass
11     def PyCall_1681926341(x):
12         for F_g1 in range(0, 1):
13             dc=demoCls()
14             dc.demoFunc(x)
15 def SLmain(x): # entry point
16     PyCall_1681926341 (x)
```

Figure 7: An example Python APP generated from the entire APP specification in Figure 6.

3.3.4 APP Selection. With the specification-based APP generation capability, the L1 fuzzer core is tasked to select one of the generated APPs from the APP queue. Prior to the two-level fuzzing loop starts, the L1 core generates an APP for each of the runtime APIs. The APP is then validated quickly by observing if running it against several arbitrary input values consistently led to crashes. If so, the APP may not have good potential hence is discarded. The validated APPs are pushed to the APP queue to initialize it.

Then, the holistic two-level fuzzing loop (as detailed in §3.5) starts. Now the L1 core randomly selects an APP X and sends it to the L2 fuzzer core for application fuzzing until the time budget allocated for it is used up. If any new *runtime* code coverage is achieved, the L1 core will invoke the APP-generation capability to generate another APP Y around the same API I as called in X. Y is then sent to the L2 fuzzer after getting validated as described above. This repeats until now more coverage can be gained around the API I, when the next APP will be selected from the queue. The rationale here is to maximally exploit the APIs that have good potential via which the Python runtime can be explored further, while exploiting as many different APIs as possible within the total testing time.

3.4 Level-2 Fuzzing (Phase 3)

Focusing on (greybox) fuzzing the test Python APPs generated during Level-1 fuzzing, the Level-2 fuzzing in PyRTFuzz primarily features two components: *instrumentation* and *custom mutation*.

3.4.1 Instrumentation. The mutation-based greybox fuzzing at Level 2 is immediately guided by the coverage of the code of the Python runtime under test. Thus, the coverage needs to be monitored during the fuzzing, for which the runtime is instrumented. The probing is only aimed to enable collecting the coverage feedback shared by the two collaborating fuzzers.

To probe in a Python runtime, static and dynamic instrumentation is performed, for its C and Python code, respectively.

3.4.2 Custom Mutation. A common, default mutation strategy in greybox fuzzers is to mutate the program input holistically as one single byte sequence, even when the input include variables of different types. To improve the (Level-2) fuzzing efficiency, PyRTFuzz introduces a new, custom mutation strategy with which the values of different variables in the APP input are separately mutated according to their types. Since these values are aimed to feed runtime API calls as arguments, the types are readily available in the associated (typed) API descriptions extracted in **Phase 1**.

Then, the mutated values of different input variables are encoded (packed together) as a byte sequence that is passed by the L2 core to the current APP. During the fuzzed execution of the APP, the mutated input byte sequence is decoded into individual API arguments right before the API is called against the arguments. The decoding is enabled by the probe inserted into the APP (e.g., Line 7 of Figure 7) during the APP generation step (at Line 6 of Algorithm 2) in **Phase 2**. This type-guided custom mutator generates type-correct values for each particular runtime API. In some cases, if this input-format-aware mutation does not lead to runtime-coverage gains for several fuzzing iterations on end, the L2 core will try the default strategy.

Algorithm 6: Custom mutation in Level-2 fuzzing

```
Input: F_P: File path for application P generated and selected by Level-1 fuzzer
Output: S: Seed for P
1  Function lv2CustomMutator (F_P)
2      S ← ∅;
3      P ← importApp (F_P);                        //Import the APP by path
4      TL ← P.TypeList;                            //Get argument types of P
5      foreach t in TL do
6          tValue ← dataProvider (t);              //Generate a random value for type t
7          S ← encodeArguments (S, tValue);        //Encode the value to S
8      return S;
```

Algorithm 6 gives the procedure of our custom mutation strategy, which takes as the input the file path F_P of an application P generated and selected by Level-1 fuzzing. At first, the seed S for P is initialized as an empty set $\emptyset$ (line 2). Subsequently, P is imported (line 3) from the given path F_P, and the list TL of P's argument types is extracted from the application P itself (line 4). Then, during the iteration over TL (line 5–7), a type-specific data provider is utilized to generate a type-correct value for each argument (line 6). These argument values are then encoded into S (line 7). Once all elements in TL have been processed, an encoded seed S is generated for P, which will subsequently be applied to P during Level-2 fuzzing.

It is crucial to ensure type-correct value generation, which is achieved through our design of a type-specific data provider (line 6) for each data type. This data provider guarantees that the generated values comply with the specific types involved. For complex Python objects, such as `BytesIO`, the corresponding constructors are invoked, and the arguments are populated with randomly generated values from the type-specific data providers. This approach ensures that even intricate Python objects are adequately handled during the mutation process, resulting in overall more comprehensive and cost-efficient testing of the Python runtime.

3.5 Holistic Two-Level Fuzzing Loop

Algorithm 7: Overall procedure of the holistic two-level fuzzing

```
   Input: PyDesc: Python API descriptions
   Output: Bugs: Bugs detected
1  Function levelOneFuzz (PyDesc)
2      H ← startPyGen (PyDesc)          //initialize the server handle for Python code generation
3      InitApps ← genInitAPPs (H);      //generate initial APPs for each API
4      OkApps ← validate (InitApps);
5      AppQueue ← initAppQueue (OkApps);
6      probPyRuntime ();                //dynamic instrumentation of the Python runtime
7      while true do
8          foreach App in AppQueue do
9              while true do
10                 S ← randomSeeds ();
11                 B ← randomBudget ();
12                 M ← loadFuzzMain (App);
13                 levelTwoFuzz (M, S, B);
14                 covChanged ← covFeedback ();
15                 if covChanged == false then
16                     break;
17                 App ← genPyApp (H, App);   //generate more APPs around the API
```

Algorithm 7 outlines the overall two-level collaborative fuzzing loop in PyRTFuzz. Each loop iteration begins with Level-1 fuzzing, which initializes a Python application generation server in an isolated process and obtains a server handle (H) for remote invocation (line 2). Next, a set of initial APPs ($InitApps$) is generated for all APIs in Python API descriptions ($PyDesc$) (line 3). Validation is then conducted on this set to remove non-executable/low-potential APPs (line 4), and the remaining APPs are inserted into the Level-1 fuzzer's APP queue $AppQueue$ (line 5).

To obtain complete coverage feedback, the fuzzer imports and dynamically instruments all Python code of the runtime libraries (line 6). For the C code of the runtime, static instrumentation is done at compile time. Once the initialization is complete, the fuzzer enters an infinite loop, continuously selecting APP for fuzzing (line 7–17) and observing their behavior until a crash or other unexpected behavior (e.g., hang) is detected.

During each iteration (line 8–17) of Level-1 fuzzing, a nested loop of Level-2 fuzzing (line 9–17) starts after an APP is selected. To initialize Level-2 fuzzing, a set of random seeds (S) and a random time budget (B) are generated, and then the main function (M) (i.e., the fuzzing entry point) of App is loaded. With these inputs, Level-2 fuzzing operates as traditional mutation-based fuzzing but with a limited time budget (line 13). Once the Level-2 fuzzing on App is completed, the L2 fuzzer collects the overall coverage variation (line 14). New coverage detection allows the fuzzer to continue operating on the API used in App, and generate more applications for Level-2 fuzzing.

4 IMPLEMENTATION AND LIMITATIONS

We have implemented PyRTFuzz as per the design shown in Figure 4, currently for CPython, the de facto standard implementation of Python. We summarize key points for the implementation of each of the three phases below.

Runtime API description extraction. The static extraction step is implemented using the standard Python AST parser [42]. The static extractor transforms Python libraries from the CPython source into their ASTs and extracts the API description fields defined in §3.2.1 by iterating through all AST nodes. It then stores this information in XML as untyped API descriptions. The dynamic refinement step is implemented using a Python built-in tracing API (`sys.settrace`) and the standard Inspect module in Python.

Level-1 fuzzing. The L1 fuzzer is implemented as three submodules. First, we created a set of Python AST-based operators that enable AST editing, thereby supporting and simplifying our SLang specification language. Using these operators, we implemented the seven SLang primitives (see Table 2). Next, we implemented the SLang compiler to translate a given APP specification to its Python code according to the operational semantics defined for each primitive (Algorithms 2 and 3). Finally, we developed a set of Remote Procedure Call (RPC) interfaces to support remote invocation by the L1 fuzzer core, which is implemented in Python. This decoupled implementation is justified by the need to avoid coverage feedback collection being interfered by the invocation of the SLang compiler which itself is implemented also in Python.

Level-2 fuzzing. We implemented the L2 fuzzer on top of Atheris [14] and libFuzzer [30]. First, we added relevant interfaces in Atheris to interact with the L1 fuzzer. To support running Level-2 fuzzing in the same process as L1 fuzzer, we imported each application's entry point (i.e., `SLmain`) and ran it dynamically through invoking `SLmain`. Next, in libFuzzer, we instantiated only one fuzzing core for the two-level fuzzing, thereby sharing the coverage feedback between L1 and L2 fuzzer cores. The per-APP time budget is implemented as a resizable time window, and the time window size can be reset based on coverage changes. This means that applications capable of triggering more basic blocks to be covered can obtain more fuzzing time. For the custom mutator, our current implementation supports randomly generating values of the top-20 commonly used data types in Python, such as integers, floats, strings, and lists.

The complete implementation of PyRTFuzz comprises 14KLoC of code, which includes 11.4KLoC of Python, 2.1KLoC of C++, and

0.5KLoC of Shell. We tested PyRTFuzz on three CPython versions: Python3.7.15, Python3.8.15, and Python3.9.15.

4.1 SLang Extensibility

The number of SLang primitives included in PyRTFuzz intuitively affects its capacity for generating diverse and complex Python applications. Hence, we implemented PyRTFuzz with considerable effort to preserve the extensibility of SLang to support new primitives. First of all, Algorithms 2–3 demonstrate the overall idea of designing SLang primitives, following which can ensure that the SLang compiler will generate Python code based on APP specifications that is valid and executable. Moreover, we have implemented many Python AST operators in PyRTFuzz to support common Python code manipulation operations (e.g., add a new function, insert a call). According to our experience, the average code size of implementing a SLang primitive in Table 2 is only around 50 LoC.

4.2 Limitations

PyRTFuzz is designed for *in-process* fuzzing, which means that it can only obtain coverage feedback from a single process where it runs. As a result, PyRTFuzz cannot fuzz APIs that are related to multiple processes [4, 11], such as those in `multiprocessing` and `pipe`. Moreover, PyRTFuzz, like common fuzzers, focuses on exploring program behaviors through various test inputs generated, which may indirectly capture some, but not explicitly all, effects of environment interactions [12, 13].

Currently, PyRTFuzz only generates Python APPs each using a single API, without considering the potential dependencies among APIs. This implementation may result in two possible side effects: (1) the generated APPs may not reflect use of the APIs in practice, leading to unrealistic applications; and the bugs detected may be false positives. (2) some of the generated APPs may not be executable due to incomplete environments. For example, an API may depend on global variables or class members initialized by other APIs, but the APP generated around the API may not have the necessary initializations, rendering it non-executable.

5 EVALUATION

We evaluated PyRTFuzz by answering three research questions:

- **RQ1:** How effective is PyRTFuzz on fuzzing Python runtime? (§5.2)
- **RQ2:** How scalable is Python APP generation in PyRTFuzz? (§5.3)
- **RQ3:** What are the factors affecting PyRTFuzz's effectiveness? (§5.4)

5.1 Experiment Setup

Experiment environment. All experiments were conducted on a 64-bit Ubuntu 18.04 with a 32-core CPU (AMD Ryzen Threadripper 3970X) and 256 GB memory. We ran each fuzzer against each target application with identical configurations on one CPU core for 5×24 hours. All experiments were repeated five times.

Baseline fuzzers for comparison. Since PyRTFuzz is the first fuzzer capable of fuzzing the entire Python runtime, it is currently impossible to directly compare with other tools. However, when using untyped-API descriptions, PyRTFuzz utilizes Atheris' original mutation strategies–which may be considered as Atheris supplied with test applications. This essentially compares PyRTFuzz's Level-2 fuzzing with Atheris as a baseline and evaluates PyRTFuzz's custom mutation strategy against Atheris' default mutation strategy. Moreover, we conducted extensive experiments and evaluations to demonstrate the effectiveness of PyRTFuzz in terms of coverage and bug detection.

Benchmarks and initial inputs. We evaluated the effectiveness of PyRTFuzz on three widely used versions of CPython, namely `Python 3.9.15`, `Python 3.8.15`, and `Python 3.7.15`. Table 3 summarizes these systems as our subjects in the 1st column, providing information on their code size in the 2nd column, the number of API and typed-API in the 3rd column, and the number of valid initial seeds (applications) in the 4th column. The `Typed-API` indicates the API count for which we successfully extracted the descriptions. As the unit tests may not cover all the runtime APIs for all three versions, the percentage of `Typed-API` is around 70%. Regarding the initial seeds for level-1 fuzzing, we first use PyRTFuzz to generate one application with one `basic` primitive for each runtime API, then calibrate these applications and filter out failed ones as initial inputs (for level-1). For example, in `Python 3.9.15`, 3,844 are kept after calibration of 4,208 applications (91.3%). While the initial seeds for level-2 fuzzing are randomly generated.

Table 3: Profiles of the 3 released CPython versions

Benchmark	Size (KLoC)	#API	#Typed-API	#L1-Seeds
Python3.9.15	C: 529.5 Python: 287.6	4,208	2,998 (71.2%)	3,844 (91.3%)
Python3.8.15	C: 487.7 Python: 277.3	4,184	2,855 (68.2%)	3,481 (83.2%)
Python3.7.15	C: 416.4 Python: 267.9	4,115	2,773 (67.4%)	3,244 (78.8%)

Performance metrics. We considered two common metrics to evaluate the effectiveness of PyRTFuzz: the number of basic blocks covered and the number of bugs triggered. We measured the coverage results (i.e., block coverage of both the runtime libraries and the interpreter) by averaging the number of basic blocks covered over five repetitions of 5×24 hours each, which ensured that all runtime APIs were covered. Across these repetitions, we did not see substantial spreads. Thus, we mainly report the average numbers.

While evaluating PyRTFuzz, we also considered the number of bugs detected as an important metric. We manually validated all reported issues, including crashes or unexpected behaviors such as hang-ups or unhandled exceptions. To validate each issue, we developed a proof of concept (PoC) to reproduce the bug-triggering inputs. If the bug could be reproduced, we considered it a new bug only when its call stack differed from all other confirmed bugs.

5.2 Effectiveness of PyRTFuzz

Coverage. Figure 8 displays the coverage evolution of PyRTFuzz on `Python 3.9.15` with specific parameter settings. In particular, we set the time budget for **L2** fuzzing as 90 seconds and the maximum APP specification size as 256, based on the empirical results obtained from our experiments.

Specifically, the left subplot of Figure 8 shows the number of basic blocks and the number of applications generated by PyRTFuzz

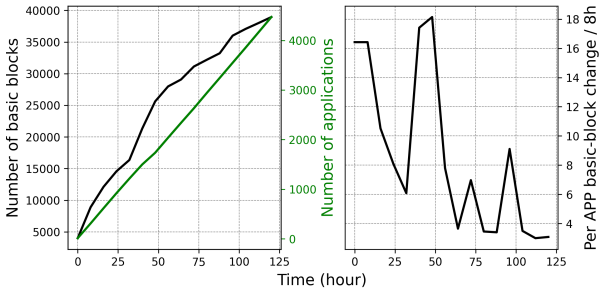

Figure 8: Coverage (y axis) evolves over the timeline (x axis) with parameters of L2 budget=90 (s) and maximum APP specification size=256 on Python 3.9.15.

over 5×24 hours. As **L1** fuzzing continuously generates new applications, the number of applications always increases (as indicated by the green line), depicted as an almost straight line. However, in some time zones (40h–50h), the growth of applications slows down somewhat. This indicates that the applications generated in these time zones can cover more basic blocks than others, according to the design of **L2** budget in §4. Regarding the coverage (the black line), it continues to rise along with the increasing number of applications, although the growth rate varies among different time zones due to differences in the generated applications.

The right subplot in Figure 8 presents the same data as the left subplot but normalized per unit time zone (8 hours). This allows us to observe the coverage changes per application in a standardized manner. As shown, the applications generated in the time zone (40h–50h) can trigger more basic blocks covered, which means that **L2** fuzzing spends more time on these applications while **L1** generates fewer applications. This is consistent with the observation in the left subplot that the growth rate of applications generated during this time zone slows down somewhat.

Bug triggering. We created three PyRTFuzz instances for the three Python versions respectively. With the 5×24 hours time budget, PyRTFuzz succeed in detecting a total of **61** bugs across the three Python versions, with **25** in Python 3.9.15, **15** in Python 3.8.15, and **21** in Python 3.7.15 respectively. After Removing the duplicated bugs over the three benchmarks, 45 unique ones are reported.

In a time frame of 5x24 hours, PyRTFuzz successfully detected 45 unique bugs across the three Python releases without getting stuck.

5.3 Scalability of Python APP Generation

Figure 9 illustrates how the measured results change with different APP specification sizes. The left subplot displays the time cost (black line) and memory usage (green line) measured for APP specification generation (dotted line) and Python APP generation (solid line). Concerning the APP specification generation, both the time cost and memory usage remain at a low level. With a specification size of 4,096, it only takes 0.06 seconds and 0.26 MB of memory. Therefore, the impact of this step on Python APP generation can be neglected.

For Python APP generation, both time and memory usage exhibit a linear relationship with the APP specification size. With a maximum specification size of 4,096, it takes 2,714.47 seconds to

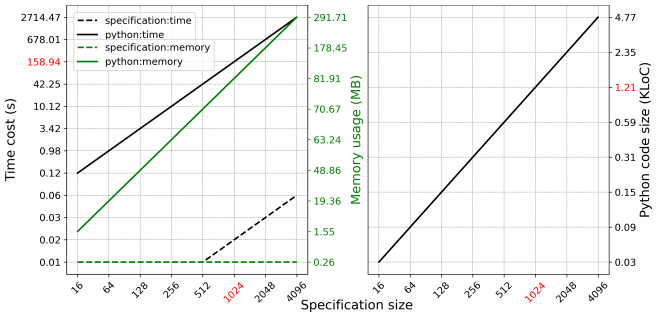

Figure 9: The time costs (y axis) of Python application generation over increasing APP specification sizes (x axis).

generate Python APPs, using 291.71 MB memory. However, such a long time cost is unacceptable for **L1** fuzzing as it would consume most of the time budget. Therefore, the maximum acceptable APP specification size for PyRTFuzz is 1,024 (with a time cost of 158.94 s), as shown in the figure, based on an empirical threshold of 300 seconds for fuzzing timeout [14].

With the 7 implemented specification primitives shown in Table 2, the right subplot of Figure 9 demonstrates a linear correlation between Python APP and specification sizes. It should be noted that this correlation is not always true when more primitives are implemented, as it depends on how the primitives are implemented. Nevertheless, increasing the specification size can generally help generate more complex Python APPs.

The cost of Python APP generation has a linear correlation with APP specification size, and increasing the APP specification size can generally help generate more complex Python APPs.

5.4 Factors Affecting Effectiveness

According to the design, we have identified three primary factors that may affect the effectiveness of PyRTFuzz: APP specification size, **L2** time budget, and the use of typed/untyped API descriptions. For each of these factors, we sampled several values and conducted 5×24 hours of fuzzing on `Python 3.9.15`.

5.4.1 APP Specification Size. Figure 10 shows the coverage evolution with different APP specification sizes. Based on our observation, we noticed that different PyRTFuzz instances with an APP specification ranging from 1 to 256 had almost identical coverage trends, which were consistently higher than the coverage trend of the specifications [512, 1024]. One possible explanation is that generating APPs (**L1**) with the [512, 1024] specification is more time-consuming than other specifications, as shown in §5.3. As a result, the number of generated APPs for this specification is lower than for other specifications, as seen in the middle subplot of Figure 10. On the other hand, with the same time budget, when **L1** takes more time for APP generation, then it may leave less time for **L2** fuzzing, resulting in less code coverage achieved.

After normalizing the data in the third subplot, it became apparent that the [1024] specification achieved the highest basic blocks per application between [75, 100] hours. This peak value was higher

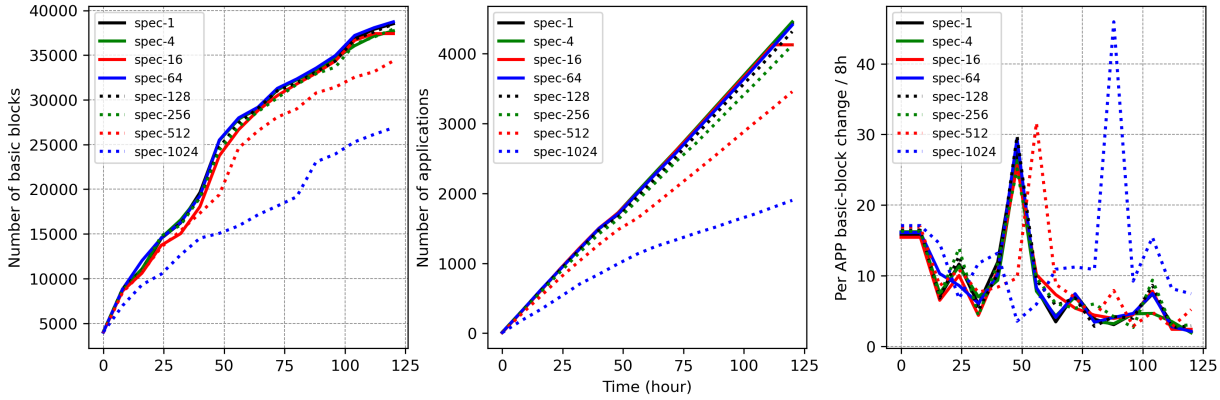

Figure 10: Coverage (y axis) evolves over the timeline (x axis) with different APP specification sizes on Python 3.9.15 (L2 budget=90 (s)).

than that of any other specification, suggesting that more complex APPs generated with larger specification sizes can potentially trigger more block coverage in **L2**. The underlying reason for the increased coverage is that generally data flow is carried by control flow; as the APP specifications become larger hence include additional or modified control-flow structures, the resultant APPs contain more or modified data flow. This, in turn, leads to more extensive exercising of the runtime's behaviors, resulting in increased code coverage.

Therefore, our results suggest that finding an appropriate balance between generating APPs (**L1**) and the APP fuzzing process (**L2**) could improve the overall coverage achieved by PyRTFuzz.

> *The specification sizes of [1–256] tend to lead to similar improvements in overall coverage, while striking a reasonable balance between the two fuzzing levels.*

5.4.2 **L2** *Time Budget.* Figure 11 illustrates the coverage evolution of PyRTFuzz on `Python 3.9.15` with four different **L2** time budgets. The plot shows that the overall coverage varies significantly based on the budget, as depicted in the left subplot. Notably, the coverage increases rapidly with a budget of 10 (s) (black solid), as does the number of generated applications (middle plot). However, after 20 hours, PyRTFuzz gets stuck, although it continues to generate more applications than other budgets. This is because too small a budget will lead to insufficient **L2** fuzzing and make it difficult to exploit the deep paths.

On the other hand, with a budget of 90 seconds (blue solid), the coverage shows a continuously increasing trend, and PyRTFuzz can discover significantly more basic blocks than those with budgets of 180 and 360 seconds. Additionally, PyRTFuzz generates more applications under this budget. The normalized data in the right subplot also shows that PyRTFuzz with a budget of 90 seconds obtains the highest peak value among the four budgets.

Thereby, **L2** time budget can greatly affect PyRTFuzz's effectiveness. Increasing the **L2** time budget can lead to uncovering more basic blocks; however, as the budget becomes too large, PyRTFuzz may become less effective due to excessive redundancy in L2 fuzzing, resulting in diminishing returns.

> **L2** *time budget can greatly affect PyRTFuzz's effectiveness, and a budget of 90 (s) is ideal for Python runtime fuzzing.*

5.4.3 Typed versus Untyped API Descriptions. Figure 12 shows the results of PyRTFuzz fuzzing on `Python 3.9.15` with typed (solid) and untyped (dot) API descriptions.

The left subplot in the figure shows that PyRTFuzz with typed API descriptions consistently outperforms the untyped version regarding coverage feedback. While the untyped version may generate slightly more applications, this is only because, within a fixed time window, untyped applications have fewer chances of uncovering deep execution paths, resulting in fewer new basic blocks being triggered. This can lead to the untyped version quickly exhausting the time window and moving on to **L1** APP generation for the next **L2** fuzzing. In contrast, typed applications are more likely to touch deep paths and trigger new coverage, allowing PyRTFuzz to reset the time window and stay in **L2** fuzzing longer than untyped ones, resulting in fewer applications generated within a given overall time frame. Furthermore, after normalizing the data (as shown in the right subplot), it can be observed that the peak value of the typed version is almost 50% higher than that of the untyped version. Additionally, the typed API descriptions are more effective in triggering basic blocks, as each APP can trigger more blocks on average with typed API descriptions.

The experiment results provide clear evidence that the type information in API descriptions significantly impacts the effectiveness of PyRTFuzz. Even though we have only implemented support for the top-20 commonly used data types (§4), and only around 70% of the APIs are type-extracted successfully (Table 3), the difference in results between the typed and untyped versions is quite significant, resulting in an improvement of up to 20%. Moreover, when employing untyped-API descriptions, PyRTFuzz leverages Atheris' original mutation strategies. Consequently, the ablation study also demonstrates the superior efficiency of PyRTFuzz's custom mutation strategy compared to Atheris' default mutation strategy.

> *PyRTFuzz can greatly improve its effectiveness when using typed API descriptions (over using untyped ones).*

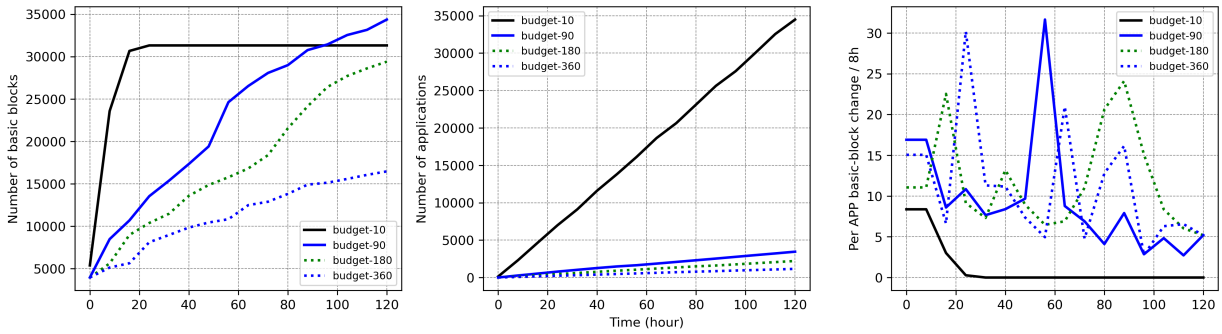

Figure 11: Coverage (y axis) evolves over the timeline (x axis) with different L2 budget on Python 3.9.15 (maximum specification size=256).

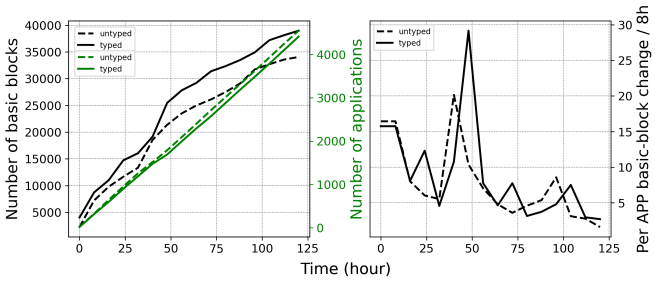

Figure 12: Coverage (y axis) evolves over the timeline (x axis) with typed and untyped API descriptions on Python 3.9.15 (L2 budget=90 (s) and maximum specification size=256).

5.5 Regarding the Vulnerabilities Discovered

Table 4 summarizes the 61 bugs over the three Python versions discovered by PyRTFuzz during our evaluation. Notably, 19 of these bugs have been validated to occur in the source code of the interpreter, while still impacting the runtime libraries. To ensure reproducibility and further investigation, we have developed Proof of Concepts (PoCs) for these vulnerabilities. Additionally, we have actively engaged with developers to confirm the bugs, comprehend their root causes, and work towards appropriate solutions for fixing them. As of the paper submission, 21 bugs have been mutually agreed upon and acknowledged by the developers. And we have not encountered any bug marked by them as "Won't fix" at this stage, signifying a positive and productive collaboration with the development community thus far. The details of these vulnerabilities are documented in **Python3.9_Vul.pdf**, **Python3.8_Vul.pdf** and **Python3.7_Vul.pdf** within our artifact package, along with the corresponding PoC scripts.

5.5.1 Case Study. In the following section, we present two case studies from the bugs PyRTFuzz detected, to demonstrate how to trigger the bugs and discuss the exploitation and security impact.

Case 1: `locale.format_string`**.** Figure 13 shows an application developed with API `locale.format_string`. When this application is run with the input `"%8663511110u"`, it causes an out-of-memory error, while the input `"%18663511110u"` triggers an unhandled `MemoryError` exception. This bug has been confirmed to affect all tested Python versions. An attacker can exploit this vulnerability to induce a denial-of-service (DoS) attack by causing the applications that use this API to crash or exit due to an unhandled exception.

Table 4: Bugs detected by PyRTFuzz.

Benchmarks	Bug Type	#Bug	#Confirmed	PoC
Python 3.9.15	MemoryError	15	1	✓
	Out of Memory	4	2	✓
	RecursionError	3	3	✓
	Hang up	3	3	✓
Python 3.8.15	MemoryError	12	1	✓
	Out of Memory	2	2	✓
	Stack Overflow	1	1	✓
Python 3.7.15	MemoryError	8	2	✓
	RecursionError	6	3	✓
	Stack Overflow	4	1	✓
	Hang up	2	1	✓
	Out of Memory	1	1	✓
Total		61	21	-

```
1 import sys
2 import locale
3 def demoFunc(format):
4     try:
5         ret = locale.format_string (format, 2)
6     except (ImportError,AttributeError,OSError,\
7         NameError,AssertionError,TypeError) as e:
8         print (e)
9 demoFunc (sys.argv[1])
```

Figure 13: Bug case 1: Out-of-memory and unhandled memory error exception in `locale.format_string`

Case 2: `ssl.SSLContext.set_alpn_protocols`**.** Figure 14 shows an application of `ssl.SSLContext.set_alpn_protocols`. When running the application with an empty string, a `MemoryError` exception is thrown, which is unexpected behavior. This bug has been confirmed to affect all tested Python versions. An attacker can exploit this vulnerability to induce a denial-of-service (DoS) attack by causing the applications that use this API to exit due to an unhandled exception.

```
1  import sys
2  import ssl
3  from ssl import *
4  def demoFunc(alpn_protocols):
5      try:
6          obj = ssl.SSLContext()
7          obj.set_alpn_protocols(alpn_protocols)
8      except (CertificateError,NotImplementedError,
9              PermissionError,SSLError) as e:
10         pass
11 demoFunc (sys.argv[1])
```

Figure 14: Bug case 2: Unhandled memory error exception in `ssl.SSLContext.set_alpn_protocols`

6 RELATED WORK

Prior works closely related to PyRTFuzz fall in three areas: *compiler fuzzing*, *collaborative fuzzing*, and *Python analysis and testing*.

Compiler fuzzing. Generation-based fuzzers, such as JSfunfuzz [45], TreeFuzz [37], and Skyfire [48], learn grammatical features and rules from existing samples to generate valid applications. Another significant contribution in JS-engine testing is CodeAlchemist [17]. This approach employs a semantics-aware split-combination method to create tests that are both semantically and syntactically correct, building upon initial seeds. However, its effectiveness is constrained by the quality and diversity of these initial seeds. Additionally, it primarily focuses on generating valid code to test the JS interpreter specifically, rather than enabling a broader and more comprehensive testing of the runtime and its interaction with applications via runtime libraries.

On the other hand, mutation-based fuzzers, such as Superion [49] and Fuzzil [15], mutate input programs on ASTs or intermediate languages with grammar awareness. LangFuzz [20] uses a grammar to generate random programs based on code fragments. Moreover, grammar-aware approaches [7, 8] targeting JVM testing focus on generating valid Java applications. Additionally, DeepSmith[9] promotes generative models for compiler testing based on machine learning. These approaches are focused on and limited to testing the compiler/interpreter alone while ignoring the runtime libraries, an integral part of the language runtime.

The SLang-based approach in PyRTFuzz draws inspiration from state-of-the-art generation-based JS-engine fuzzers, such as TreeFuzz/Skyfire for data-driven generation and CodeAlchemist utilizing semantics-aware split-combination approach. Yet these approaches focus on generating valid code itself, rather than particularly on synthesising API-centred test applications—which is essential for testing Python runtimes holistically including the interaction between interpreter and applications via runtime libraries. In contrast, PyRTFuzz uses a novel specification-based code generation approach to generate diverse and valid Python applications around various runtime APIs from scratch, for *synthesizing API-centred applications*. Furthermore, with a two-level collaborative fuzzing methodology, PyRTFuzz tests the Python runtime *holistically*, including both the interpreter and runtime libraries. PyRTFuzz also differs from existing relevant approaches in its ability to generate applications spanning various domains, with diverse control flow complexities while ensuring application validity (e.g., data-flow reachability).

Collaborative fuzzing. EnFuzz [6] proposes a seed synchronization mechanism to seamlessly ensemble diverse fuzzers to obtain better performance. CollabFuzz [36] allows multiple fuzzers to collaborate under an informed scheduling policy based on central analyses. Cupid [16] provides a collaborative fuzzing framework that automatically selects a set of complementary fuzzers for parallelized and distributed fuzzing, improving the efficiency of software fault-finding. These fuzzers target improving the effectiveness through collaboration, but they function as equal entities working in synergy at the application level.

In all, existing collaborative fuzzers commonly fuzz at one (i.e., application) level. In contrast, our two-level collaborative fuzzing operates at two different levels (i.e., compiler-testing and application fuzzing). **L1** aims to generate diverse applications and feed them to **L2**, while **L2**'s feedback also guides **L1**'s application generation. Both levels collaborate to test the entire Python runtime comprehensively and are indispensable to each other.

Python analysis and testing. PyPrecditor [50] combines dynamic tracing and static symbolic execution to analyze Python programs and predict potential bugs. PolyCruise [27] supports cross-language analysis [51] of Python programs that can analyze their interactions with native libraries [24]. Python program fuzzers such as PolyFuzz [28] and Atheris [14] are used to fuzz Python programs and their bound native libraries to detect bugs and vulnerabilities. These techniques are all designed to test Python applications, with some of them being aware of Python runtime libraries. For instance, PolyFuzz [28] and Atheris [14] can provide coverage feedback by instrumenting Python runtime libraries, but they are only capable of fuzzing the specified applications.

In comparison, PyRTFuzz stands out for its focus on holistic testing of the entire Python runtime, including both the interpreter and runtime libraries. It can generate applications with diverse domains, covering all runtime APIs, and use them to fuzz the Python runtime. Additionally, while not its primary objective, PyRTFuzz can also fuzz Python applications.

7 CONCLUSION

We presented PyRTFuzz, a novel greybox fuzzing technique for systemantically testing hence detecting bugs in Python runtime systems. The design of PyRTFuzz features a two-level collaborative fuzzing methodology that combines (1) a generation-based fuzzer at Level 1 as enabled by a specification-based application generation scheme based on a declarative specification language with (2) a mutation-based fuzzer at Level 2 that explores the Python applications generated and selected at Level 1 through type-guided, format/structure-aware generation of concrete application inputs. Our extensive experiments demonstrated significant merits of the two-level collabroative fuzzing design, which can be potentially applied to the runtime of interpreted languages beyond Python.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Science Foundation (NSF) through the grants CCF-2146233 and 2239605, as well as by the U.S. Office of Naval Research (ONR) through the grant N000142212111.

REFERENCES

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[2] Franco Bazzichi and Ippolito Spadafora. 1982. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering* 4 (1982), 343–353.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

[4] Haipeng Cai and Xiaoqin Fu. 2021. D^2ABS: A Framework for Dynamic Dependence Analysis of Distributed Programs. *IEEE Transactions on Software Engineering (TSE)* 48, 12 (2021), 4733–4761. https://doi.org/10.1109/TSE.2021.3124795 (impact factor: 6.226).

[5] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[6] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers.. In *USENIX Security Symposium*. 1967–1983.

[7] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

[8] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

[9] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 95–105.

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*.

[11] Xiaoqin Fu and Haipeng Cai. 2021. FlowDist:Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 2093–2110.

[12] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li LI. 2020. Seads: Scalable and Cost-Effective Dynamic Dependence Analysis of Distributed Systems via Reinforcement Learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–45. https://doi.org/10.1145/3379345 (impact factor 2.5; journal-first paper).

[13] Xiaoqin Fu, Boxiang Lin, and Haipeng Cai. 2022. DistFax: A Toolkit for Measuring Interprocess Communications and Quality of Distributed Systems. In *IEEE/ACM International Conference on Software Engineering (ICSE), Tool Demos*. 51–55. https://doi.org/10.1145/3510454.3516859

[14] google. 2022. A Coverage-Guided, Native Python Fuzzer. https://github.com/google/atheris.

[15] Samuel Groß. 2018. Fuzzil: Coverage guided fuzzing for javascript engines. *Department of Informatics, Karlsruhe Institute of Technology* (2018).

[16] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic fuzzer selection for collaborative fuzzing. In *Annual Computer Security Applications Conference*. 360–372.

[17] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*.

[18] Kenneth V. Hanford. 1970. Automatic generation of test cases. *IBM Systems Journal* 9, 4 (1970), 242–257.

[19] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-based type inference for Python 3. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*. Springer, 12–19.

[20] Christian Holler, Kim Herzig, Andreas Zeller, et al. 2012. Fuzzing with Code Fragments.. In *USENIX Security Symposium*. 445–458.

[21] Vanshika kakkar. 2023. Top 10 Programming Languages to Learn in 2023. https://www.geeksforgeeks.org/top-10-programming-languages-to-learn/

[22] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

[23] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399.

[24] Wen Li, Li LI, and Haipeng Cai. 2022. PolyFax: A Toolkit for Characterizing Multi-Language Software. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tool Demos*. 1662–1666. https://doi.org/10.1145/3540250.3558925

[25] Wen Li, Austin Marino, Haoran Yang, Na Meng, Li Li, and Haipeng Cai. 2023. How are Multilingual Systems Constructed: Characterizing Language Use and Selection in Open-Source Multilingual Software. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2023).

[26] Wen Li, Na Meng, Li Li, and Haipeng Cai. 2021. Understanding language selection in multi-language software projects on GitHub. In *IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings*. 256–257.

[27] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. {PolyCruise}: A {Cross-Language} Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2513–2530.

[28] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1379–1396. https://www.usenix.org/conference/usenixsecurity23/presentation/li-wen

[29] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76.

[30] LLVM. 2020. LibFuzzer: A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[31] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2018. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140* (2018).

[32] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical deep similarity learning-based type inference for Python. In *Proceedings of the 44th International Conference on Software Engineering*. 2241–2252.

[33] M.Zalewski. 2014. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt.

[34] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 48–53.

[35] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. *IPSJ Transactions on System LSI Design Methodology* 7 (2014), 91–100.

[36] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2021. Collabfuzz: A framework for collaborative fuzzing. In *Proceedings of the 14th European Workshop on Systems Security*. 1–7.

[37] Jibesh Patra and Michael Pradel. 2016. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664* (2016).

[38] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*. 2019–2030.

[39] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12 (1972), 366–375.

[40] Victoria Puzhevich. 2020. Top Programming Languages to Use. https://scand.com/company/blog/top-programming-languages-to-use-in-2020/

[41] Python. 2022. CPython Repository. https://github.com/python/cpython.

[42] Python. 2022. Python 3.8 Abstract Syntax Trees. https://docs.python.org/3.8/library/ast.html.

[43] Python.org. 2023. The Python Language Reference. https://docs.python.org/3/reference/.

[44] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[45] Jesse Ruderman. 2007. Introducing jsfunfuzz. *URL http://www. squarefree. com/2007/08/02/introducing-jsfunfuzz* 20 (2007), 25–29.

[46] Dipanjan Sarkar, Raghav Bali, and Tushar Sharma. 2018. Practical machine learning with Python. *A Problem-Solvers Guide To Building Real-World Intelligent Systems. Berkely: Apress* (2018).

[47] Emin Gün Sirer and Brian N Bershad. 1999. Using production grammars in software testing. *ACM SIGPLAN Notices* 35, 1 (1999), 1–13.

[48] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[50] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. 2016. Python predictive analysis for bug detection. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 121–132.

[51] Haoran Yang, Wen Li, and Haipeng Cai. 2022. Language-Agnostic Dynamic Analysis of Multilingual Code: Promises, Pitfalls, and Prospects. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Ideas, Visions and Reflections*. 1621–1626. https://doi.org/10.1145/3540250.3560880

[52] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.