A Practical Fuzzer for the Python Runtime System

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

Haipeng Cai*
University at Buffalo, SUNY
Buffalo, NY, USA
haipengc@buffalo.edu

Abstract

Python's widespread usage underscores the critical need for the reliability and security of its runtime system. However, automated tools for detecting runtime bugs in this impactful language are scarce. This tool demo showcases PyRTFuzz, a practical fuzzing tool for testing the Python runtime which encompasses the language interpreter and runtime libraries. Leveraging a blend of generation- and mutation-based fuzzing. PyRTFuzz unveils 61 exploitable bugs, including those within the interpreter core and various runtime libraries. Demonstrating scalability and cost-effectiveness, PyRTFuzz shows promise for extensive bug detection, with potential applicability to the runtime of other interpreted languages.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Python, language runtime, interpreted language, fuzzing

ACM Reference Format:
Wen Li and Haipeng Cai. 2026. A Practical Fuzzer for the Python Runtime System. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3803437.3806417

1 Introduction

Python has become one of the most widely used programming languages across domains such as data science, web development, and machine learning [7, 9, 10, 25]. Ensuring the security and reliability of the Python runtime system (which includes the language interpreter and its libraries) is therefore critically important. Any bug in this system may have far-reaching implications, impacting a wide range of Python applications such as today's AI systems.

Unfortunately, reports on bugs in the Python runtime have increased in recent years, with approximately 2,000 annual cases reported in the widely used CPython [17], revealing a significant 10% with severe security implications [13]. In this context, automated tool support is imperative, yet currently insufficient. Various generation-based fuzzers like CodeAlchemist [4], TreeFuzz [16], and Skyfire [18], as well as mutation-based fuzzers such as Superion [19] and Fuzzil [2], focus on testing the compiler or interpreter alone without addressing the runtime libraries, which are an integral part of the language runtime. Tools like PolyCruise [11], PolyFuzz [12], and Atheris [1] focus on Python program analysis and fuzzing. Like other tools [3, 5, 6, 8, 14, 15, 20–24, 26, 27], they lack the *holistic* approach to *testing the Python runtime as a whole*.

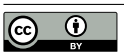
PyRTFuzz is designed to fill this gap by testing the Python runtime system end-to-end via a collaborative two-level fuzzing approach. In PyRTFuzz, two fuzzing engines work in concert: **Level 1** focuses on generating syntactically and semantically valid Python programs (which we call APPs) that invoke runtime APIs, and **Level 2** focuses on supplying those programs with various input data and driving their execution. The two levels share feedback and cooperate to thoroughly explore the runtime. This design enables holistic fuzzing of an interpreted language runtime – while Level 1 ensures broad program coverage of API behaviors, Level 2 provides deep input exploration for those programs.

To achieve this, PyRTFuzz introduces a novel specification language called SLang for program generation, which uses primitives based on Python's syntax and common usage patterns. SLang allows the Level 1 fuzzer to generate diverse yet valid Python code targeting different runtime libraries. Meanwhile, PyRTFuzz leverages runtime API descriptions (extracted via static and dynamic analysis of the Python source) to guide both the program generation and the input mutation. These API descriptions provide type and semantic hints that enable a type-guided mutation strategy at Level 2, producing inputs that conform to expected formats and corner cases. In essence, PyRTFuzz's two-tier design tackles key challenges in Python runtime testing: creating complex, valid test programs and then fuzzing those programs with realistic inputs, all in an automated loop. By fuzzing interpreter and libraries together, PyRTFuzz can trigger bugs that isolated approaches would miss.

PyRTFuzz was initially developed as a research prototype and showed promising results in discovering real-world Python runtime bugs (with over 60 new bugs found) in a controlled experimental setting [13]. That prior work established the core technique and effectiveness of two-level collaborative fuzzing. This paper differs by focusing on the system realization and practical deployment of PyRTFuzz as a reusable tool. It presents previously unpublished tool-centric content, including the tool's architecture, SLang interpreter, remote application-generation service, custom mutator integration, fuzzing orchestration, and engineering trade-offs made to support robustness, automation, and multi-version use. The paper further emphasizes tool usage and usability, demonstrating how the research prototype has been transferred into a functioning and extensible fuzzing tool for practical Python runtime testing.

Our tool demo video can be found at https://youtu.be/cOhi9eG-IK0 and our tool repository at https://github.com/awen-li/PyRTFuzz.

*The corresponding author.



FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3806417

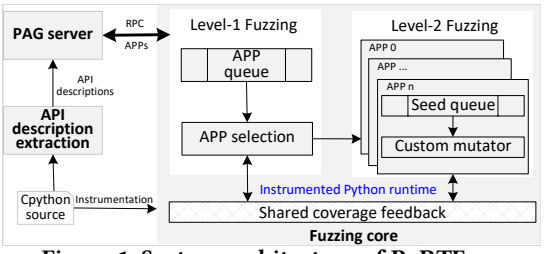


Figure 1: System architecture of PyRTFuzz.

2 System Architecture

We start with the architectural design of PyRTFuzz, outlining how it is structured as an end-to-end system and how its components interact to support automated, scalable runtime fuzzing in practice.

Design Overview. PyRTFuzz is engineered as a practical, fully automated fuzzing tool for Python runtime systems. It targets holistic runtime testing by jointly exercising the Python interpreter core and the runtime libraries within a single instrumented execution environment. The overall design of PyRTFuzz is illustrated in Figure 1. The tool takes the source code of the given Python runtime system, including all the unit tests available, as inputs. Then, it works in three phases, each corresponding to the respective component: (1) *API description extraction* from the runtime source code and unit tests, (2) a Python Application Generation *(PAG) server* that synthesizes diverse test programs, and (3) a coordinated *fuzzing core* that executes these programs with input mutations. If any bug is found, they are reported as the output of PyRTFuzz.

Components and interfaces. PyRTFuzz is designed as a modular system composed of the three major components noted above. These components are connected through well-defined interfaces and a shared representation of extracted API descriptions. Concretely, API descriptions (e.g., definitions and exceptions thrown) produced by the extractor are consumed by both the PAG server and the fuzzing core as a common specification layer. The PAG server is separated as an independent service and is invoked via remote procedure calls (RPC) by the fuzzing core; this separation is an explicit architectural decision to improve extensibility and to avoid unintended interference between generation activities and runtime coverage measurement. The application generator is specification-driven and based on a custom intermediate scripting language (SLang), together with its SLang interpreter, which translates SLang scripts into executable Python applications.

End-to-end workflow and feedback sharing. At a high level, PyRTFuzz operates as a loop that alternates between (i) generating Python applications that exercise runtime APIs and (ii) fuzzing those applications with diverse inputs. The fuzzing core operates in a two-level manner: the Level-1 fuzzer performs generation-based fuzzing by requesting and selecting generated applications (through RPC to the PAG server), and the selected applications are then passed to Level-2 fuzzing, which performs mutation-based fuzzing on a given application. Level-2 uses an input-format-aware custom mutator to generate structured inputs for each application in accordance with the extracted API descriptions. A key architectural feature is that execution (coverage) feedback is shared between the two fuzzers, enabling collaborative fuzzing across interpreter and runtime-library code. This shared-feedback design supports scaling to thousands of runtime APIs while maintaining low coupling between individual components, thereby improving robustness and extensibility.

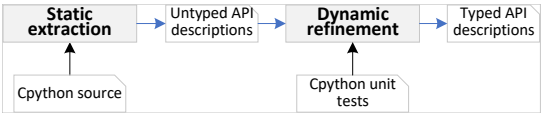


Figure 2: Runtime API description extraction in PyRTFuzz.

Enhancements beyond the research prototype. PyRTFuzz incorporates several engineering refinements that transition the original research prototype into a robust, reusable tool. First, PyRTFuzz was engineered to support multiple Python versions and configurations in practice, including automation scripts to instrument a target runtime system build and to extract API specifications, allowing users to quickly prepare Python runtimes for fuzzing. Second, cross-version compatibility was strengthened by adapting key components (including the SLang-based generator and the Atheris-based mutator) to handle differences across different runtime versions (e.g., CPython 3.7, 3.8, 3.9). Third, PyRTFuzz integrates with LLVM's libFuzzer through Atheris to enable coverage-guided execution at scale, alongside practical optimizations of the fuzzing loop and safeguards to handle runtime errors, crashes, and timeouts gracefully for long-running campaigns.

3 Design and Implementation Details

We now elaborate each of the key components of PyRTFuzz, focusing on system realization and engineering decisions. Currently, PyRTFuzz is implemented for and tested on CPython (the de-facto standard Python runtime) and operates directly on instrumented runtime builds. All fuzzing activities are performed in-process to allow efficient collection and sharing of coverage feedback across interpreter and library code.

3.1 Runtime API Specification Extraction

This component automatically extracts structured descriptions of Python runtime APIs from a target Python implementation. This extraction is performed once per Python version and provides a common specification foundation for both Python application generation and input mutation during fuzzing. By deriving API information directly from the runtime implementation, PyRTFuzz avoids reliance on external documentation and ensures consistency with the exact runtime version under test.

As illustrated in Figure 2, PyRTFuzz employs a two-stage pipeline that combines static extraction with dynamic refinement. The extractor analyzes both the Python runtime source code and its accompanying unit tests to derive rich API metadata, including module paths, function or method names, argument signatures, return types (when available), and exception behavior.

Static extraction. PyRTFuzz parses the Python runtime libraries using Python's standard Abstract Syntax Tree (AST) facilities. Specifically, it constructs an AST visitor based on the `NodeVisitor` class to traverse the runtime source code and enumerate candidate API entries. This visitor handles key syntactic constructs such as `import`, `class`, `function`, and `try`, enabling the extractor to identify API definitions and retrieve information required for invocation, including parameter names and structure.

The result of this step is a set of *untyped API descriptions*, which capture structural metadata but omit concrete type information.

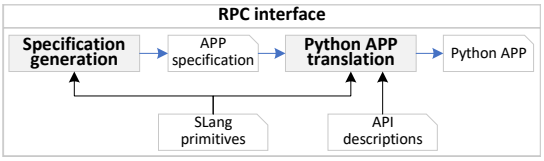


Figure 3: The PAG server in PyRTFuzz.

Dynamic refinement. To enrich the untyped API descriptions with concrete type information, PyRTFuzz performs a dynamic refinement stage by executing the runtime's unit tests under tracing. This step leverages Python's `inspect` module to observe API invocations at runtime. By examining frame objects during execution, PyRTFuzz extracts observed argument types, return types, and exception behavior associated with each API.

The dynamically observed information is merged into the existing untyped API descriptions, producing *typed API descriptions* after all unit tests have been analyzed. This hybrid extraction strategy improves robustness compared to purely static inference while avoiding the overhead of heavyweight program analysis. In practice, it provides sufficiently precise type information to guide both application generation and input mutation.

Caching and reuse. Extracted API descriptions are serialized into a tool-internal format and cached on a per-version basis. This design decouples the relatively expensive extraction process from the fuzzing loop, allowing subsequent fuzzing runs to reuse the same specifications without re-extraction. As a result, PyRTFuzz significantly reduces setup overhead and simplifies support for multiple Python runtime system versions.

3.2 Python Application Generation Server

Given runtime API descriptions as input, the Python application generation (PAG) server generates Python applications (APPs) that invoke specific APIs while preserving valid control flow and data flow from the program entry point to the API call site. The generator does not rely on manual templates. Instead, it systematically constructs applications that (i) target individual runtime APIs, (ii) exhibit diverse control-flow structures, and (iii) are executable and suitable for subsequent input mutation by the fuzzing core.

The PAG server is implemented as a standalone service that is invoked remotely by the fuzzing core. This design isolates code generation from fuzzing execution, prevents generator-side execution from polluting runtime coverage measurements, and allows generation strategies to evolve independently.

Internally, the PAG server is organized around three functional components, as illustrated in Figure 3 and detailed below. This organization enables clear separation between specification logic, code generation, and interaction with the fuzzing core.

Specification generation. To decouple application structure from concrete Python syntax, the PAG server uses a lightweight intermediate specification language, SLang, to describe Python APPs. SLang captures common Python program constructs such as function invocation, control-flow statements, and object-oriented usage patterns, and is grounded in Python's AST representation. Specifications are generated automatically based on a selected runtime API and a configurable application size.

To ensure semantic relevance, each generated specification is required to include an operation involving the target API, and additional constructs are composed to introduce varied control-flow complexity. Random composition and nesting of specification constructs enable the generation of diverse applications without hard-coding API-specific logic.

Python APP translation. The generated specifications are translated into executable Python APPs by an interpreter-style compiler within the PAG server. The translation proceeds incrementally, compiling each specification construct into corresponding Python code while preserving dependencies and execution order. Required module imports are automatically inferred from API descriptions and inserted during translation.

Each generated APP is equipped with a standardized entry point, which enables uniform invocation by the fuzzing core. This design ensures that generated applications can be dynamically loaded, executed, and repeatedly fuzzed without additional instrumentation.

RPC interface. Interaction between the PAG server and the fuzzing core is implemented via a lightweight RPC interface using socket-based communication. The interface supports initialization with a given set of API descriptions, on-demand generation of applications for specified APIs, and graceful shutdown at the end of fuzzing. This minimal interface reduces coupling between components and simplifies integration with the fuzzing core.

The PAG server is designed to be extensible. New generation constructs or translation rules can be added by extending the SLang interpreter and its AST-based compilation logic, without modifying the fuzzing core. In practice, this allows for expanding the space of generated applications or adapt the generator to new Python runtime features while preserving the overall fuzzing workflow.

3.3 Coordinated Fuzzing Core

The fuzzing core orchestrates execution and mutation within an instrumented Python runtime. Its primary responsibility is to coordinate generation-based and mutation-based fuzzing in a unified execution environment, enabling effective exploration of both the Python interpreter core and runtime libraries. To this end, PyRTFuzz integrates multiple fuzzing stages within a single in-process fuzzing core, allowing coverage feedback to be shared efficiently across different fuzzing activities.

Overall design. The fuzzing core repeatedly executes the generated APPs with mutated inputs and monitoring runtime behavior. For each target runtime API, the core requests an APP from the PAG server and dynamically invokes its standardized entry point (`SLmain`). The APP is then executed multiple times with mutated inputs produced by a custom mutator. During execution, coverage feedback is collected from both interpreter and runtime-library code and is used to guide subsequent fuzzing decisions.

A key feature of the design is that all fuzzing activities are performed in-process within a single instrumented runtime. This enables efficient sharing of coverage feedback across fuzzing stages while avoiding the overhead and inconsistency associated with process isolation. At the same time, the fuzzing core maintains private seed queues and execution budgets for individual applications to prevent interference between concurrent fuzzing tasks.

Two-level collaborative fuzzing. The fuzzing core follows a two-level collaborative strategy. At Level-1, the core manages a queue of generated Python applications and selects applications for execution based on observed coverage behavior. At Level-2, the core

performs mutation-based fuzzing on a selected application within a bounded time budget. If execution yields new runtime coverage, Level-1 triggers additional application generation for the same API via RPC, producing more complex applications that are subsequently passed back to Level-2. This process continues until no further coverage is observed around the API, after which the core shifts attention to other applications.

This coordination strategy enables PyRTFuzz to allocate additional fuzzing effort to promising APIs and execution paths while still maintaining broad coverage across the runtime system. By sharing coverage feedback between the two levels, PyRTFuzz achieves collaborative exploration of interpreter and library code without tightly coupling generation and mutation logic.

Implementation foundations. The fuzzing core is implemented on top of both Atheris and libFuzzer. In practice, a single fuzzing instance is initialized per target runtime, with individual seed queues maintained for different applications and fuzzing stages, while a global coverage map is shared. Interfaces were integrated into Atheris to support interaction with the Level-1 fuzzing logic, and dynamic invocation of application entry points (`SLmain`) enables Level-2 fuzzing to be performed within the same process. When using libFuzzer, a unified fuzzing core is instantiated for both levels, ensuring efficient coverage sharing across fuzzing stages.

Initialization and instrumentation. During initialization, the fuzzing core performs two key setup steps. First, an initial set of Python applications is generated—one per runtime API—using the PAG server. These applications are calibrated to eliminate invalid or non-executable cases, and the remaining ones are loaded into the Level-1 seed queue. Second, the Python runtime is instrumented to enable coverage collection. C-language components of the runtime are instrumented statically during compilation, while Python runtime-library code is dynamically instrumented at startup using Atheris. This hybrid instrumentation strategy enables comprehensive monitoring of runtime execution during fuzzing.

Input mutation strategy. At Level-2, the fuzzing core employs a custom mutation strategy that mutates input variable values based on typed API descriptions. Raw fuzzer input is decoded into type-correct argument values and passed to the executing application. During execution, these values are decoded and supplied to API calls at runtime. The current implementation emphasizes random value generation for the top 20 commonly used Python data types, including primitive and container types such as `int`, `float`, `string`, and `list`. This design avoids the need for API-specific mutators while maintaining compatibility with runtime expectations.

4 Discussion

This section discusses the design trade-offs, limitations, and extensibility of PyRTFuzz as a practical runtime fuzzing tool.

Implementation scope and limitations. PyRTFuzz is designed as an *in-process* fuzzer, which enables efficient sharing of coverage feedback between fuzzing stages but limits visibility into execution that spans multiple processes. As a result, runtime features that rely on multi-process execution, such as those in the `multiprocessing` and `pipe` modules, are not fully captured by the current implementation. This limitation reflects a deliberate design trade-off favoring efficiency and stability over comprehensive multi-process coverage.

In addition, the current application generation strategy focuses on producing Python applications that exercise a single runtime API at a time. While this design improves scalability and simplifies fuzzing orchestration, it does not model complex dependencies or initialization requirements across multiple APIs.

Extensibility via SLang. PyRTFuzz is made extensible via its application-generation infrastructure. In particular, this is achieved by adding new SLang primitives that correspond to Python AST operators and common code-manipulation patterns. Each primitive encapsulates a reusable generation pattern and ensures that generated applications remain syntactically valid and executable.

In the current implementation, a range of AST operators has been integrated to support common Python constructs. Implementing a new SLang primitive typically requires approximately 50 lines of code, making it feasible for developers to extend the generator with additional control-flow structures or language features as needed.

Automation and multi-version support. To support practical deployment, PyRTFuzz includes automation scripts that handle runtime instrumentation, API specification extraction, and fuzzing execution. These scripts enable unattended fuzzing campaigns and simplify setup for new target runtimes. Users can configure fuzzing parameters—such as execution budgets, application complexity limits, and target runtime versions—through command-line options.

PyRTFuzz is designed to support multiple CPython versions using a unified workflow. API specifications are generated on a per-version basis, and the same fuzzing pipeline can be applied across different runtime releases. This enables regression testing and cross-version comparison with minimal configuration changes. However, due to compatibility differences across CPython versions, minor implementation adjustments may be required in practice, particularly for components implemented in Python, such as the PAG server and runtime instrumentation logic.

Engineering trade-offs. Several engineering decisions were made to ensure that PyRTFuzz functions as a robust tool rather than a one-off research prototype. All major components are designed to be restartable and scriptable, enabling long-running fuzzing campaigns. The modular architecture allows individual components (e.g., application generation, mutation logic) to be extended or replaced independently. Meanwhile, certain limitations (e.g., in-process execution and single-API application generation) were accepted to maintain stability, efficiency, and ease of deployment.

5 Conclusion

This paper presented a tool demonstration of PyRTFuzz, a practical fuzzing system for testing Python runtime implementations. Complementary to the research prototype, the paper focused on the architectural design, engineering realization, and usage workflow of PyRTFuzz, illustrating how a research prototype has been transferred into a reusable and extensible tool. Future work includes extending application generation to model inter-API dependencies, enhancing support for multi-process runtime features, and adapting the tool to additional interpreted language runtimes.

Acknowledgment

This work was supported in part by the National Science Foundation (NSF) through the grant CCF-2505223, as well as by the U.S. Office of Naval Research (ONR) through the grant N000142512252.

References

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

[2] Samuel Groß. 2018. Fuzzil: Coverage guided fuzzing for JavaScript engines. *Department of Informatics, Karlsruhe Institute of Technology* (2018), 1–60.

[3] Qiuhan Gu. 2023. LLM-based code generation method for Golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2201–2203.

[4] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *NDSS*. 1–15.

[5] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. 2021. Sofi: Reflection-augmented fuzzing for JavaScript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2229–2242.

[6] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*. 2613–2630.

[7] Wen Li, Li Li, and Haipeng Cai. 2022. On the vulnerability proneness of multilingual code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 847–859.

[8] Wen Li, Li Li, and Haipeng Cai. 2022. PolyFax: A toolkit for characterizing multi-language software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1662–1666.

[9] Wen Li, Austin Marino, Haoran Yang, Na Meng, Li Li, and Haipeng Cai. 2024. How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 3 (2024), 1–46.

[10] 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.

[11] 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.

[12] 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.

[13] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*. 1645–1659.

[14] Jiayi Lin, Changhua Luo, Mingxue Zhang, Lanteng Lin, Penghui Li, and Chenxiong Qian. 2026. Fuzzing JavaScript Engines by Fusing JavaScript and WebAssembly. In *ICSE*. (To Appear).

[15] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–26.

[16] 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).

[17] Python. 2023. CPython Repository. https://github.com/python/cpython.

[18] 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.

[19] 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.

[20] Wai Kin Wong, Dongwei Xiao, Cheuk Tung Lai, Yiteng Peng, Daoyuan Wu, and Shuai Wang. 2025. Extraction and Mutation at a High Level: Template-Based Fuzzing for JavaScript Engines. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 2898–2926.

[21] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–13.

[22] Yuanmin Xie, Zhenyang Xu, Yongqiang Tian, Min Zhou, Xintong Zhou, and Chengnian Sun. 2025. Kitten: A Simple Yet Effective Baseline for Evaluating LLM-Based Compiler Testing Techniques. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 21–25.

[23] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 709–735.

[24] Haoran Yang, Wen Li, and Haipeng Cai. 2022. Language-agnostic dynamic analysis of multilingual code: Promises, pitfalls, and prospects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1621–1626.

[25] Haoran Yang, Weile Lian, Shaowei Wang, and Haipeng Cai. 2023. Demystifying issues, challenges, and solutions for multilingual software development. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1840–1852.

[26] Haoran Yang, Yu Nong, Tao Zhang, Xiapu Luo, and Haipeng Cai. 2024. Learning to detect and localize multilingual bugs. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2190–2213.

[27] Chibin Zhang, Gwangmu Lee, Qiang Liu, and Mathias Payer. 2025. REFLECTA: Reflection-based Scalable and Semantic Scripting Language Fuzzing. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 1772–1787.

APPENDIX: PyRTFuzz DEMO WALKTHROUGH

In this section, we guide you through the essential operational steps of our PyRTFuzz demonstration. We utilize illustrative screenshots to visually demonstrate the installation process and showcase the tool's usage.

1. PyRTFuzz access

We provided three ways to access our tool package:

1. DOI is provided through FigShare
https://doi.org/10.6084/m9.figshare.22758932.v1

2. An evolving version is maintained on GitHub
The GitHub repository is:
https://github.com/awen-li/PyRTFuzz.git

3. A docker image is provided with PyRTFuzz installed
docker pull daybreak2019/pyrtfuzz:v1.1

2. PyRTFuzz setup

2.1. Installation

- Step 1: Download the Docker image and run a Docker container based on the image
  docker pull daybreak2019/pyrtfuzz:v1.0
  docker run -it --name "pyrtfuzz" daybreak2019/pyrtfuzz:v1.1
- Step 2: Update PyRTFuzz to the latest version and build the project within the container
  cd /root/PyRTFuzz
  git pull
  ./build.sh pyrtfuzz

2.2 Basic Test

To validate whether the environment is ready, a simple test can be run as follows:
cd /root/PyRTFuzz/test

a. Python APP generation: python pyfuzz_appgen.py
The test results should be True

b. Fuzzing setup: python pyfuzz_setup.py
The test should be run successfully.

c. Run Fuzzing on python3.9
cd /root/PyRTFuzz/experiments && setPyEnv.sh python3.9
python -m fuzzloop -pyscript=seeds_python3.9 &

Fuzzing can run successfully as shown in Figure 4.

3. PyRTFuzz usage

For evaluation on the tool, we conducted tests using PyRTFuzz on three widely used CPython versions: Python 3.7.15, Python 3.8.15, and Python 3.9.15. The standard workflow for applying PyRTFuzz to a CPython version is outlined below:

- [1] Static Instrumentation of C Language Units in CPython.
  This step is done at compilation time, primarily for the source of the interpreter core and C parts of runtime libraries. In PyRTFuzz, we have integrated a script to quickly implement this functionality:

  `PyRTFuzz/experiments/buildCPython.sh`

  Specifically, the corresponding CPython (Python-***.tar.xz, e.g., Python-3.9.15.tar.xz) should be copied into directory:

  `PyRTFuzz/cpython`

  then edit `buildCPython.sh` to set the corresponding version to the `ALL_VERSIONS`. Finally, run the script `buildCPython.sh` to install the CPython with instrumentation.

```
#1393   REDUCE cov: 685 ft: 704 corp: 7/1523b lim: 4096 exec/s: 116 rss: 413Mb
#1401   REDUCE cov: 685 ft: 704 corp: 7/1514b lim: 4096 exec/s: 116 rss: 413Mb
#2048   pulse  cov: 685 ft: 704 corp: 7/1514b lim: 4096 exec/s: 170 rss: 413Mb
#3147   REDUCE cov: 685 ft: 704 corp: 7/1513b lim: 4096 exec/s: 262 rss: 413Mb
#4096   pulse  cov: 685 ft: 704 corp: 7/1513b lim: 4096 exec/s: 341 rss: 413Mb
#6356   REDUCE cov: 685 ft: 704 corp: 7/1509b lim: 4096 exec/s: 529 rss: 413Mb
#8060   REDUCE cov: 685 ft: 704 corp: 7/1475b lim: 4096 exec/s: 671 rss: 419Mb
#8192   pulse  cov: 685 ft: 704 corp: 7/1475b lim: 4096 exec/s: 682 rss: 420Mb
#8207   REDUCE cov: 685 ft: 704 corp: 7/1472b lim: 4096 exec/s: 683 rss: 420Mb
#8463   REDUCE cov: 685 ft: 704 corp: 7/1469b lim: 4096 exec/s: 705 rss: 423Mb
#8479   REDUCE cov: 685 ft: 704 corp: 7/1468b lim: 4096 exec/s: 706 rss: 423Mb
#8613   REDUCE cov: 685 ft: 704 corp: 7/1467b lim: 4096 exec/s: 717 rss: 424Mb
#8885   REDUCE cov: 685 ft: 704 corp: 7/1466b lim: 4096 exec/s: 740 rss: 427Mb
#8928   REDUCE cov: 685 ft: 704 corp: 7/1464b lim: 4096 exec/s: 744 rss: 427Mb
#9006   REDUCE cov: 685 ft: 704 corp: 7/1463b lim: 4096 exec/s: 750 rss: 428Mb
#9071   REDUCE cov: 685 ft: 704 corp: 7/1462b lim: 4096 exec/s: 755 rss: 428Mb
#16384  pulse  cov: 685 ft: 704 corp: 7/1462b lim: 4096 exec/s: 1260 rss: 505Mb
```

Figure 4: Output message of PyRTFuzz during Fuzzing

- [2] Install PyRTFuzz to the Installed CPython.
  After installing the CPython (Python-version) with instrumentation at step1, users can run the following commands to install PyRTFuzz to the corresponding Python environment:

  ```
  cd PyRTFuzz/
  setPython.sh Python-version
  ./build.sh pyrtfuzz
  ```

- [3] Collect API Descriptions on the Given CPython.
  Users can edit script PyRTFuzz/apispec/PySpec/genSpec.sh to add support for the version (line 9-12), then use following commands to collect API description:

  ```
  cd PyRTFuzz/apispec/PySpec
  ./genSpec.sh Python-version
  ```

  the API description XML will be generated under the directory with the name of "CPY_*****_apispec.xml".

  Clarification Resolving compatibility issues across various CPython versions often necessitates modifying the implementation of PyRTFuzz. This is primarily because critical components such as Python application generation and the Atheris Fuzzer are developed using Python. Adjustments to these components ensure the seamless operation of PyRTFuzz across different CPython versions. For the versions that are compatible with Python3.7/3.8/3.9, the guidelines above are sufficient.