SnakeCharmer: Automatic Fuzzing Harness Generation for Pure and Hybrid Python Libraries

GABRIEL SHERMAN, University of Utah, USA
STEFAN NAGY, University of Utah, USA

With Python's rising popularity, ensuring the correctness of its ever-growing ecosystem of software libraries is more critical than ever. Recently, fuzzing has become a de facto technique for vetting software libraries, enabled via the use of *harnesses*: small wrapper programs that inject fuzzer-generated test cases into the library under test. While harness creation has shed its reliance on human expertise and is now fully automated for languages such as C and C++, Python remains uniquely challenging—both for *pure* Python libraries as well as *hybrid* ones combining Python with native C/C++ extensions—due to (1) limited visibility across language boundaries, (2) the absence of reliable bug oracles, and (3) incomplete type information. Consequently, attempts at automating harnessing for Python fail to both uphold critical runtime behaviors and produce the structured call and data flows needed for effective fuzzing, leaving much of today's Python ecosystem largely unvetted.

To overcome these challenges and broaden fuzzing's reach across Python libraries, this paper introduces SnakeCharmer: the first automated harness generation approach for both pure *and* hybrid Python libraries. At its core, SnakeCharmer leverages static analysis to first capture key interface information from both Python and native code components, subsequently enriching it with runtime-captured type information and exception behaviors. During fuzzing, SnakeCharmer further distinguishes between expected exceptions and true library bugs, filtering out benign exceptions that would otherwise derail testing progress. Together, these techniques significantly enhance the scope and effectiveness of fuzzing across the Python library ecosystem, enabling the automated discovery of bugs in code previously inaccessible to existing Python fuzzing efforts.

We evaluate SnakeCharmer alongside today's leading Python auto-harnessing approach, PyRTFuzz; the actively fuzzed expert-written harnesses from both OSS-Fuzz and PolyFuzz; and the harnesses generated by Google's own state-of-the-art LLM-driven automatic harnessing approach, OSS-Fuzz-Gen. Across 21 diverse Python libraries, SnakeCharmer attains type-recovery precision and exception-filtering accuracy of 95% and 97%, respectively, further attaining 1.48×, 1.87×, 1.78×, and 1.40× the code coverage of the fuzzing harnesses from PyRTFuzz, OSS-Fuzz, PolyFuzz, and OSS-Fuzz-Gen, respectively. Further, SnakeCharmer finds 16, 24, and 24 more Python library bugs than all expert- and LLM-created harnesses as well as PyRTFuzz, respectively—uncovering a total of 20 new bugs, with 18 since confirmed or fixed by developers.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

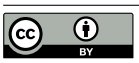
Additional Key Words and Phrases: Fuzzing, Python, Libraries, Harness Generation

ACM Reference Format:
Gabriel Sherman and Stefan Nagy. 2026. SnakeCharmer: Automatic Fuzzing Harness Generation for Pure and Hybrid Python Libraries. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE004 (July 2026), 23 pages. https://doi.org/10.1145/3797066

1 Introduction

With over 8 million users today, Python is a mainstay of modern programming, widely embraced for its simplicity and broad platform support [47, 78]. Its software ecosystem is vast, with 600,000+

Authors' Contact Information: Gabriel Sherman, University of Utah, Salt Lake City, USA, gabe.sherman@utah.edu; Stefan Nagy, University of Utah, Salt Lake City, USA, snagy@cs.utah.edu.



ACM 2994-970X/2026/7-ARTFSE004
https://doi.org/10.1145/3797066

libraries spanning diverse domains like multimedia [34], data serialization [42], and machine learning [31], making Python a central pillar of today's evolving computing landscape [41]. Yet, as Python's popularity continues to grow, defects in its libraries carry significant risks, ranging from logical errors [67] to denial-of-service [68] and even code execution bugs [72]. Even more concerning, many Python libraries incorporate native back-end extensions implemented in C/C++, reintroducing memory-corruption vulnerabilities into an otherwise memory-safe language [65, 69, 70]. Thus, given Python's ubiquity in modern computing, the need for scalable approaches for vetting the security and correctness of its growing library ecosystem has never been more urgent.

In recent years, *fuzzing* has become one of today's most popular software testing strategies due to its immense success in unearthing both logical and security bugs across nearly every software domain today [64]. To test software libraries, fuzzing relies on pre-written *harnesses*: wrapper programs that perform any necessary library setup before feeding fuzzer-generated data to its targeted API functions. However, crafting these harnesses manually is both time-consuming and requires significant domain expertise. To alleviate this burden, recent work has introduced a range of automated harness generation techniques for C and C++ libraries [46, 48, 54, 56, 57, 76], allowing developers to uncover more bugs and higher code coverage with minimal manual effort.

Yet, despite the vast success of fuzzing harness generation for other languages' library ecosystems, **today's diverse Python libraries remain largely unharnessed, and thus untested**, due to three longstanding asymmetries: **First**, limiting source code analysis to Python-only library components misses critical semantics defined in Python libraries' native C/C++ extensions, such as function signatures and library-raised exceptions [34]. **Second**, unlike C/C++, Python lacks clear crash-based bug oracles; its errors are often subtle and context-dependent, making it difficult to distinguish *real* bugs from benign, expected exceptions [4]. **Third**, data type information, which is critical for constructing valid inputs and API sequences, is often incomplete or *missing* in Python libraries, impeding reconstruction of the arguments and data flows required to correctly call API functions. Without addressing these challenges, current state-of-the-art Python harnessing approaches see only limited Python library support (e.g., just CPython core libraries [62]), failing to scale across Python's broader ecosystem of 600,000+ libraries—and leaving countless bugs undiscovered.

To overcome these challenges, we introduce **SnakeCharmer**: the first fuzzing harness generator designed to support both pure *and* hybrid Python libraries. Unlike prior Python harnessing approaches that rely on capturing code semantics from available library unit tests [62], which are seldom thorough—or that violate library semantics entirely [12]—SnakeCharmer *incrementally* builds a semantic model of the target library, refining data type information and bug-oracle semantics on-the-fly during harness generation. For hybrid Python-C/C++ libraries, SnakeCharmer extends this strategy with cross-language static analysis, recovering native-side data type, function signature, and exception information critical for driving semantically correct library execution. Leveraging these recovered semantics, SnakeCharmer automatically synthesizes conformant API interactions, preserving complex data flows across function calls toward uncovering deep bugs, even in libraries with opaque or under-specified interfaces. By prioritizing semantic correctness *throughout* harness synthesis, SnakeCharmer generates harnesses that are both thorough *and* realistic—**greatly expanding fuzzing's reach across Python's ever-growing library ecosystem**.

We evaluate SnakeCharmer on **21** diverse Python libraries, comparing it to state-of-the-art Python harnessing approaches PyRTFuzz [62], Google's LLM-driven OSS-Fuzz-Gen [12], as well as each library's actively fuzzed expert-written harnesses from OSS-Fuzz [75] and PolyFuzz [61]. Across our evaluations, we show that SnakeCharmer achieves high precision, with **95%** type-inference precision and **97%** exception-filtering accuracy relative to expert-written harnesses. Further, SnakeCharmer outperforms PyRTFuzz, OSS-Fuzz-Gen, and expert-written OSS-Fuzz and PolyFuzz harnesses with **1.48**×, **1.40**×, **1.87**×, and **1.78**× their total code coverage, respectively,

while uncovering **24**, **24**, and **16** more bugs than PyRTFuzz, OSS-Fuzz-Gen, and all expert-written harnesses, respectively. Of SnakeCharmer's 24 found bugs, **20** are new, with **18** confirmed or fixed.

Through the following contributions, we extend the reach of software testing and bug discovery by introducing **the first automated harness generation approach directly addressing the unique challenges of fuzzing both pure and hybrid Python libraries**:

- We survey the fundamental challenges of harnessing Python's many pure and hybrid software libraries, identifying three core obstacles—**(1)** cross-language functionality, **(2)** ambiguous bug oracles, and **(3)** gaps in type information—that prevent direct adoption of conventionally successful auto-harnessing techniques within today's ever-growing Python library ecosystem.
- We overcome these limitations via SnakeCharmer: the first fuzzing harness generator supporting both pure and hybrid Python libraries. We show how SnakeCharmer addresses Python's unique introspection challenges across hybrid C/C++ components, the absence of reliable bug oracles, and its frequent gaps in type information—automatically crafting thorough, correct harnesses that extend fuzzing's reach to more Python libraries than previously possible.
- We demonstrate SnakeCharmer's high precision, with **95%** type recovery precision and **97%** exception-filtering accuracy relative to libraries' available ground-truth expert-written harnesses.
- We show SnakeCharmer's effectiveness in expanding code coverage, achieving **1.40–1.87**× the total code coverage of competing state-of-the-art approaches for harnessing Python, including Google's own LLM-driven OSS-Fuzz-Gen, as well as available expert-written fuzzing harnesses.
- We show that SnakeCharmer boosts bug discovery, revealing **24**, **24**, and **16** more bugs than PyRTFuzz, OSS-Fuzz-Gen, and all expert-written harnesses respectively. Of SnakeCharmer's **20** newly found bugs, **18** are so far confirmed, with **15** since fixed following our reporting.
- To enable future research in Python harnessing, we release our prototype and all artifacts at the following URL: https://github.com/FuturesLab/SnakeCharmer.

2 Background and Related Work

This section introduces the topics most relevant to our work: Python software libraries and their bugs, fuzzing techniques for Python, and the challenges of Python fuzzing harness generation.

2.1 Python's Library Ecosystem

In the last two decades, Python has emerged as one of the most widely used programming languages in the world, valued for its simplicity, readability, and broad applicability across domains [47, 78]. A major factor behind Python's success is its extensive and ever-growing ecosystem of reusable libraries. With over 600,000 libraries currently available on the Python Package Index (a.k.a. PyPI [41]), Python software developers can more easily than ever build powerful applications leveraging third-party library functionality such as numerical computing [31], multimedia processing [34], and data storage [23] and serialization [38, 42], among many others.

2.2 Bugs in Python Libraries

Because Python libraries are deeply integrated across a wide spectrum of software systems, bugs within them can pose significant risks and cause widespread disruption. While interpreter-level issues such as denial of service [68], logic errors [67], and arbitrary code execution [72] are already concerning, the risks are amplified in libraries that embed native C or C++ code. These *hybrid* libraries are susceptible to low-level memory corruption vulnerabilities, including buffer overflows [69], use-after-frees [65], and null pointer dereferences [70]. For example, a recent vulnerability in the popular Pillow [34] library revealed that specially crafted image files could trigger heap corruption, potentially leading to system compromise [71]. To proactively uncover such bugs and strengthen the security of Python libraries, considerable effort has been directed toward applying

automated bug-finding techniques to the Python ecosystem. Among these, *fuzzing*—today's leading automated software testing technique [64]—has emerged as a particularly promising approach.

2.3 Fuzzing Python Libraries

Fuzzing (a.k.a. "fuzz testing") stress-tests software by repeatedly performing random mutations on inputs that trigger new code paths, with the aim of gradually uncovering a program's erroneous input-dependent behavior [64]. Having been broadly effective at surfacing complex bugs across many of today's diverse computing domains [6, 51, 75], fuzzing is by now widely considered an essential technique for vetting software integrity both pre- and post-deployment.

Applying fuzzing to libraries requires the creation of *harnesses* (e.g., Figure 1): lightweight wrapper programs designed to feed fuzzer-generated inputs into the library's public interface functions, in addition to performing any prerequisite steps (e.g., library initialization). Google's large-scale OSS-Fuzz initiative [75] continuously fuzzes thousands of harnesses at any given time, finding more than 10,000 vulnerabilities across over 1,000 widely used software libraries. As such, the success of library fuzzing depends heavily on the *quality* of harnesses [52], making the creation of correct *and* thorough library harnesses a task that often demands significant domain expertise.

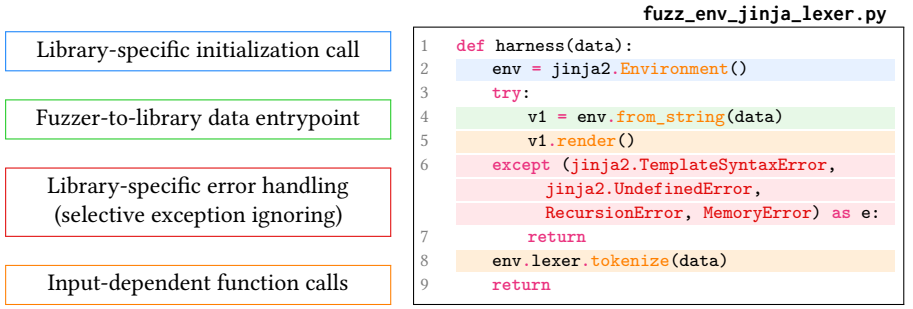


Fig. 1. Visualization of a fuzzing harness for the Jinja2 [35] library, with key code constructs highlighted.

Among today's tools for fuzzing Python libraries, Google's Atheris [53] remains by far the most prominent, currently serving as the primary Python fuzzing engine in OSS-Fuzz [75]. Central to Atheris's design is its support for both pure Python and hybrid libraries, achieved through specialized coverage-tracking tooling for both Python and native-code components. Similarly, PolyFuzz [61] incorporates finer-grained code coverage metrics—such as cross-language branch predicates—to better guide fuzzing on hybrid Python libraries. Yet, despite uncovering numerous Python library bugs in practice, these fuzzers unfortunately remain limited by one universal bottleneck: **the manual effort and expertise needed to construct effective Python harnesses**.

3 Motivation: Challenges of Python Harnessing

While considerable research has emerged on automating harness generation for library fuzzing [46, 48, 54, 56, 57, 76], existing efforts have overwhelmingly focused on C and C++ libraries, leaving Python comparatively underexplored. Though PyRTFuzz [62] represents the largest step thus far toward automating harness generation for Python, it remains limited in scope—supporting only interpreter-level APIs and thus failing to address the broader space of third-party libraries, particularly those incorporating native code. In the following, we survey representative Python libraries, and ultimately identify **three key challenges** that hinder the direct extension of conventional non-Python harnessing approaches to Python: **(1)** cross-language functionality, **(2)** ambiguous bug oracles, and **(3)** gaps in type information. We examine each of these challenges in detail below.

3.1 Challenge 1: Cross-language Functionality

Many Python libraries adopt a *hybrid* architecture [23, 30, 31, 33, 34], combining Python with lower-level components implemented in native C or C++. This approach allows developers to balance Python's ease of use with the improved efficiency, greater memory control, and rich ecosystems of native-code libraries (e.g., H5py's utilization of the HDF5 C API [24]). While native-defined library functionality is indeed accessible to Python-side components [73], purely Python-level code analysis and introspection (i.e., runtime reflection) often *fail* to fully capture native-side exception, data type, and function information—posing a significant barrier to harnessing of hybrid libraries.

Python example:

```python
from hiredis import Reader
r = Reader()
inspect.signature(Reader.feed)
# ValueError: no signature found.
```

Hiredis-py: reader.c

```c
static PyObject *Reader_feed(Reader *self, PyObject *args) {
    if (!PyArg_ParseTuple(args, "s*|nn", &buf, &off, &len))
        return NULL;
} # Types: bytes (required), int (optional), int (optional).
```

Fig. 2. Abridged example of the hybrid Python library Hiredis-py [25]: although the `Reader` class is accessible from Python, Python-level introspection (e.g., via `inspect`) fails to recover its functions' signatures, as their argument types and optionality (e.g., `"s*|nn"`) are encoded only within the library's native-side logic.

Unfortunately, today's state-of-the-art automated harnessing approach for Python, PyRTFuzz [62], relies solely on Python-level code analysis [7], restricting its view of library semantics to only what is visible at the Python layer. For example, the core functionality of the Hiredis-py library [25] is implemented entirely in C (Figure 2), leaving key native-side artifacts—such as argument type information for functions belonging to the `Reader` class—beyond PyRTFuzz's reach. As we observe that **over 15%** of Python libraries in Google's OSS-Fuzz [75] define functionality in native C/C++, **effective Python harnessing must therefore support thorough cross-language analysis** toward recovering the full code semantics of hybrid libraries' Python *and* native-side components.

3.2 Challenge 2: Ambiguous Bug Oracles

Traditional fuzzing workflows [51, 74] rely on distinct bug oracles—typically crashing process signals like `SIGSEGV`—to identify unexpected behavior toward flagging potential bug-triggering inputs. In contrast, Python provides no such oracles: it reports errors through *exceptions*, which often obscure distinction between benign, expected conditions and genuine bugs. Moreover, depending on the library, the *same* exception may indicate either routine behavior (e.g., errors from opening a missing file) or a serious failure (e.g., corrupted internal state). For example, harnesses [75] for the PyJSON5 [38] and AutoFlake [9] libraries exhibit distinct handling of `TypeError` exceptions (Figure 3)—with the former catching and ignoring them entirely, and the latter flagging them as bugs.

PyJSON5: fuzz_json.py

```python
def harness(data):
    ...
    except TypeError: pass
    # TypeError caught and ignored.
```

AutoFlake: fuzz_fix_code.py

```python
def harness(data):
    fix_code(data)
    ...
    # TypeError uncaught (thus signaled as bug).
```

Fig. 3. Abridged OSS-Fuzz [75] harnesses demonstrating how Python libraries PyJSON5 [38] and AutoFlake [9] treat identical `TypeError` exceptions differently: PyJSON5 as an expected behavior, while AutoFlake as a bug.

While existing Python fuzzers [53, 61] assume harnesses encode correct exception handling, this approach unfortunately does not scale, requiring harness writers to anticipate and catch *all* possible benign exceptions themselves. Automated harnessing, in turn, must therefore differentiate benign versus buggy exceptions unique to every library. To this end, PyRTFuzz [62] statically recovers library-raised exceptions (e.g., explicit `raise` or `assert` statements in the library's code), yet fails to recognize when *interpreter*-raised exceptions (e.g., `TypeError`) represent *expected* library

behavior—commonly when libraries instead defer raising input-validation-related exceptions to the interpreter—resulting in numerous false-positive bug reports. For example, when fuzzing CPython's AST [7] library, interpreter-raised `SyntaxError` exceptions are intended as benign [14], yet we see many instances where PyRTFuzz misclassifies them as bugs. Effective Python harnessing thus demands **fully automated, context-aware exception handling on a per-library granularity**.

3.3 Challenge 3: Missing Type Information

Most C/C++ harnessing tools leverage libraries' static type information to construct valid function arguments and to model inter-API dependencies [48, 54, 76]. While Python itself supports type annotations for parameters, return values, and local variables, these annotations are entirely *optional* (e.g., Figure 4)—and in practice, rarely used: recent work shows that **fewer than 10%** of eligible code elements across 9,000 real-world Python projects include any type annotations whatsoever [50]. Without consistent type information, harnessing faces difficulty in reconstructing the function arguments and data flows needed to meaningfully exercise libraries' full range of behavior.

Pillow: IptcImagePlugin.py

```
1  def getiptcinfo(im: ImageFile.ImageFile) ->
2    dict[tuple[int,int],bytes|list[bytes]]|None: # Full ret/arg types.
```

LXML: diff.py

```
1  def htmldiff(old,new):
2      # Zero type info.
```

Fig. 4. Example functions from the Pillow [34] and LXML [28] libraries *with* and *without* optional types.

To compensate for gaps in library type information, PyRTFuzz opportunistically captures additional types through dynamic analysis (i.e., reflection) on libraries' available unit test suites. However, as noted by its authors [62], this strategy is brittle in practice: many libraries ship with incomplete test suites, often exercising just a fraction of a library's total functions [77]. When type information is unavailable, PyRTFuzz falls back to generating primitive arguments, limiting its ability to construct the structured inputs required by many Python library functions. Thus, effective Python harnessing necessitates a robust and fine-grained type inference that is **capable of recovering meaningful type information, even in the absence of library type annotations**.

Impetus: Python library harnessing faces unique challenges not encountered in conventional C/C++ library harnessing [48, 54, 57, 76]. While recent advancements succeed in extending fuzzing to Python's vast ecosystem [53, 61, 62], **existing approaches fall short of offering a comprehensive solution for effectively automating generation of fuzzing harnesses for pure *and* hybrid Python libraries.** Without a strategy that addresses cross-language interactions, ambiguous crash oracles, and opaque or missing type information, large portions of the Python ecosystem will remain inaccessible to fuzzing—leaving critical bugs undetected.

4 SnakeCharmer: Approach & Implementation

To address the challenges of harnessing Python libraries, we introduce **SnakeCharmer**: the first automated approach capable of generating effective fuzzing harnesses for both pure and hybrid Python libraries. We detail our high-level design behind SnakeCharmer's key components below.

4.1 Overview

As Figure 5 shows, SnakeCharmer spans three phases, beginning with pre-processing both Python and native C/C++ components to construct an initial representation of the library's structure. From there, harness synthesis proceeds with the mutation of call sequences and arguments, guided by runtime analysis of code coverage, exception behavior, and type information. Finally, successful harnesses are concretized and prepared for fuzzing via conventional Python fuzzing engines like

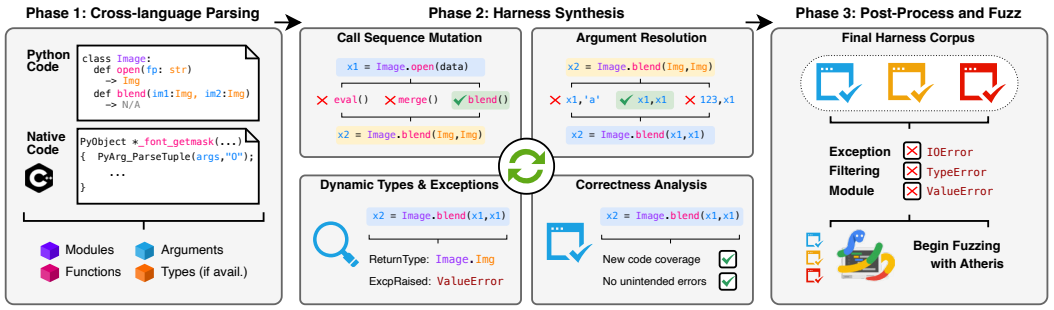


Fig. 5. SnakeCharmer's high-level workflow for automatically generating Python library fuzzing harnesses.

Atheris [53]. Collectively, these steps empower SnakeCharmer's generation of effective, high-coverage harnesses for both pure and hybrid Python libraries—significantly expanding fuzzing's reach across Python's diverse and ever-growing library ecosystem.

4.2 Phase 1: Cross-language Parsing

Effective harnessing requires a comprehensive view of a library's interfaces—not only at the Python level, but also across any underlying native-code extensions. To this end, SnakeCharmer performs cross-language static analysis of libraries' Python and C/C++ components, extracting key artifacts such as class and function definitions, type hints (if present), and module hierarchies. By recovering this information, SnakeCharmer assembles an initial understanding of library behavior, enabling it to reason about cross-language functionality ordinarily invisible to Python-only harnessing (e.g., PyRTFuzz [62]) such as native-code-defined functions and exceptions. In the following, we detail SnakeCharmer's parsing of both Python and native-code library components.

Field	Example
Module	PIL.IcnsImagePlugin;
Parent	ImageFile.ImageFile;
Object	IcnsImageFile;
Function	load;
Arguments	int | None;
Return Type(s)	Image.core.PixelAccess | None;

Pillow/src/PIL/IcnsImagePlugin.py

```
1  class IcnsImageFile(ImageFile.ImageFile):
2      def load(self, scale: int | None) ->
           Image.core.PixelAccess | None:
```

Fig. 6. SnakeCharmer-retrieved information for Pillow's [34] `IcnsImageFile.load()` function.

4.2.1 ***Python Code Parsing.*** SnakeCharmer begins by recursively traversing the library's pure-Python components, extracting structural and type information per each defined function and class (e.g., Figure 6) such as fully qualified paths, owning contexts (e.g., class or module), arguments (including optional ones), and any available type annotations. At a high level, these analyses form SnakeCharmer's initial view of the library's distinct Python-level interfaces—guiding its subsequent construction of harnesses mirroring realistic API usage patterns.

4.2.2 ***Native Code Parsing.*** Following Python code parsing, SnakeCharmer analyzes C/C++ source files in the library's build tree to identify native functions registered with the Python runtime via the canonical Python/C API [73], typically exposed through mechanisms such as `PyMethodDef` tables or type slots within `PyTypeObject` structures. To further recover interface-level details often missing from Python-level analysis alone, SnakeCharmer also inspects calls to native-side argument parsing routines (e.g., `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()`) to infer argument types (via format strings), arity, and optionality (e.g., Figure 2). In parallel, SnakeCharmer

identifies exceptions defined in native code (e.g., Figure 7) by scanning for calls to common error-signaling functions such as `PyErr_SetString()` and `PyErr_Format()`, extracting both their indicated exception types and associated error messages. Together, these cross-language analyses enrich SnakeCharmer's modeling of library behavior—laying the foundation for generating far-reaching harnesses that systematically exercise functionality across the Python-C/C++ language boundary.

Pillow/src/decode.c

```c
if (state->xsize <= 0 || state->xsize + state->xoff > im->xsize ||
    state->ysize <= 0 || state->ysize + state->yoff > im->ysize) {
    PyErr_SetString(PyExc_SystemError, "tile cannot extend outside image");
    return NULL;
}
```

Fig. 7. Example native-raised exception (`PyExc_SystemError`) and corresponding message within Pillow [34].

4.3 Phase 2: Harness Synthesis

Armed with a cross-language view of library functionality, SnakeCharmer begins harness generation aimed at faithfully and comprehensively exercising API behavior. Following similar successful approaches in non-Python library harnessing [58, 76, 82, 83], SnakeCharmer begins harnessing by targeting library *entrypoint* routines: functions that serve as the direct loading point for feeding user-controlled input into the library. To do so, SnakeCharmer prioritizes functions that accept plain strings (`str`), raw bytestreams (`bytes`), or standard I/O interfaces (e.g., `IO[bytes]`, `IO[str]`), aligning with typical entrypoints seen in current actively fuzzed Python harnesses (e.g., `from_string()` in Figure 1). Broadly, these entrypoint routines serve as the foundation upon which SnakeCharmer constructs more complex, multi-function harnesses, as we describe below.

4.3.1 ***Call Sequence Mutation.*** Following library entrypoints, SnakeCharmer begins synthesizing multi-function harnesses by exploring data dependencies between API calls—namely, calling new functions that will meaningfully operate on values or objects produced earlier in the call sequence. To guide its harnessing, SnakeCharmer builds a type-driven graph of candidate API dependencies, derived from type information available statically (e.g., Figure 4) as well as any later recovered dynamically (§ 4.3.4), encompassing the following two forms of inter-API dependencies:

- **Value-flow links:** When data resulting from or modified by one function is consumed by another. These include both return-value chains (e.g., H5py's harness [22] feeding `h5f.open()`'s returned file identifier into `h5py.File()`), as well as shared mutable arguments updated and reused across distinct calls (e.g., LXML's harness [27] passing the same XML `root` node into both `etree.indent()` and `etree.ElementTree()`). Such flows model producer-consumer relationships across API calls, tying together the library's interdependent operations. For functions with *multiple* distinct return types (e.g., Astroid's [15] `extract_node()` returning either a `NodeNG` or a `list` thereof), SnakeCharmer incorporates value-flow links for each, and subsequently constrains the returned type during harnessing to the requirements of candidate return-consuming functions (e.g., permitting only a `NodeNG` to flow into Astroid's [15] `locate_child()`).
- **Shared-object links:** When multiple class methods are invoked on the same object instance, thereby sharing its state. For example, as Figure 1 shows, Jinja2's [35] calls `env.from_string()` and `env.lexer.tokenize()` are both applied to the same `env` object. While no explicit flow exists between the return or argument values of such method calls, the shared object contexts across these functions induces implicit state interactions that influence library behavior.

By modeling these inter-API data dependencies, SnakeCharmer grounds its call sequence construction in feasible API interactions—capturing how data is produced, updated, and reused across

the library's functions—iteratively refining generated candidate call sequences (§ 4.3.3) to ultimately converge on realistic, semantically valid usage patterns for the library under test.

4.3.2 ***Argument Resolution.*** Once a candidate function is selected based on a plausible data dependence on earlier calls (§ 4.3.1), SnakeCharmer must resolve its remaining arguments. Existing harnessing approaches for C/C++ [46, 54, 56, 76] leverage its static typing, using explicit type signatures exposed by the library to drive construction of well-typed call sequences. In contrast, Python offers no such guarantees: type annotations are optional and often incomplete in practice (§ 3.3), with many functions also intentionally designed to accept arguments of *multiple* types [50].

To address this challenge, SnakeCharmer mutates not only argument values (e.g., injecting random values or `None`), but also their *types*, when needed. For each unresolved argument, SnakeCharmer consults available type information—including statically parsed signatures (§ 4.2) as well as dynamically recovered types (§ 4.3.4)—to select a plausible argument type and instantiate a corresponding value. When no argument type information is available, SnakeCharmer falls back to a permissive strategy, randomly selecting from a broad pool of commonly used types (e.g., `str`, `float`). In total, SnakeCharmer's argument resolution spans the following four strategies:

- **Library-defined objects:** For arguments requiring object types specific to the target library (e.g., Pillow [34]'s `PIL.Image.Image`), SnakeCharmer first attempts to reuse a compatible object produced earlier in the call sequence (e.g., the result of Pillow's `Image.open()` entrypoint). If none are available, SnakeCharmer instantiates a new object via the type's available constructor.
- **Primitive types:** When a function argument is determined to be a primitive type (e.g., `int`, `str`, or `bytes`), SnakeCharmer synthesizes a randomly generated value of that type.
- **Built-in container types:** To resolve container types like `list`, `dict`, or `tuple`, SnakeCharmer constructs an instance of the container and populates it with randomly created elements consistent with any available type information. When element types are unknown, SnakeCharmer samples a type at random and generates elements accordingly (e.g., filling a `list` with random `int` values).
- **No type information:** Should an argument's type be unavailable (e.g., due to incomplete type annotations), SnakeCharmer falls back to randomly selecting a type from its pool of primitives, containers, and library-defined object types, and creates a corresponding value accordingly.

While disabled by default, SnakeCharmer additionally supports resolving optional function arguments (e.g., the two optional integers for Hiredis-py's [25] native `Reader.feed()` function, indicated by "`|nn`" in Figure 2) through randomly resolving subsets of optional parameters. After argument resolution, the candidate call is integrated into the call sequence, followed by a runtime correctness analysis (§ 4.3.3) to assess the validity of the resulting harness under execution.

4.3.3 ***Correctness Analysis.*** Following each mutation, SnakeCharmer performs lightweight validation to ensure the resulting call sequence runs free of unexpected errors and meaningfully exercises library logic. At a high level, this pre-fuzzing step serves to detect broken or uninteresting call sequences early, allowing SnakeCharmer to revisit and rectify them before the harness is finalized. Inspired by prior approaches [58, 76], SnakeCharmer executes candidate call sequences on a pre-chosen set of well-formed (e.g., XML files for LXML [28]) and invalid inputs (e.g., random strings)—identical to those conventionally used as fuzzing seeds [55]—monitoring for two key signals:

- **No unintended errors:** To catch unintended exceptions—errors that suggest a candidate call sequence is actually malformed—SnakeCharmer flags and discards any call sequences that result in *interpreter*-raised exceptions (e.g., `TypeError`, `SyntaxError`) on well-formed inputs that should otherwise pass through the library undisturbed (e.g., valid PDF files for PikePDF [33]). Conversely, SnakeCharmer treats *library*-raised exceptions as expected error-handling behavior (e.g., Pillow's C-side image sizing errors in Figure 7), and does not discard triggering call sequences.

- **Changing code coverage:** Beyond unwanted errors, SnakeCharmer also evaluates candidate call sequences by runtime code coverage, retaining those that exhibit both input-dependent coverage variation across seed inputs and new coverage over Python or native-side library code. Call sequences with unchanging code coverage across inputs are discarded, as they typically indicate malformed executions or uninteresting logic that lacks input-dependent behavior, while novel coverage reflects successful calling of library code unharnessed by prior call sequences [48, 58, 76].

Altogether, SnakeCharmer's correctness analysis acts as the final checkpoint to filter-out invalid or otherwise unfruitful candidate call sequences, ensuring that only semantically sound and productive harnesses are retained and later devoted fuzzing resources (§ 4.4).

4.3.4 ***Dynamic Type & Exception Recovery.*** While SnakeCharmer's correctness analysis focuses on culling unusable call sequences, it also provides an opportunity to recover library semantics unavailable via static analysis alone (§ 4.2). To this end, SnakeCharmer augments runtime analysis with dynamic introspection to capture additional type and exception information, enabling refinement of both harness construction (§ 4.3) and downstream harness-side exception handling (§ 4.4.1).

- **Observed type information:** When static type information is incomplete (§ 3.3), SnakeCharmer leverages Python's runtime reflection (e.g., `type()`) to recover concrete types seen during execution. Return types are directly recorded at runtime, while argument types are captured only for call sequences previously validated by SnakeCharmer's correctness analysis (§ 4.3.3). These inferred types are integrated into SnakeCharmer's internal model of the library, enabling more precise inter-API dependency resolution and argument generation in subsequent harnessing iterations.
- **Observed expected exceptions:** As § 4.3.3 details, SnakeCharmer flags erroneous call sequences via interpreter-raised exceptions on well-formed inputs normally accepted by the library without error (e.g., valid HDF5 files for H5Py [23]). Accordingly, SnakeCharmer must also identify and suppress expected exceptions arising from malformed inputs that the library intentionally *rejects* (e.g., empty, random, or corrupted data), unlike prior approaches that misreport these exceptions as bugs (§ 3.2). To accomplish this, SnakeCharmer captures all library- or interpreter-raised exceptions across the library's pre-chosen set of invalid inputs (§ 4.3.3), forming a baseline of benign, bug-irrelevant library error behavior to eventually be filtered-out during fuzzing (§ 4.4.1).

Through dynamic analysis, SnakeCharmer recovers key library information that is fed back into its iterative harness synthesis loop (Figure 5), enabling sound harness generation even in the absence of library type annotations or well-defined exception semantics. Once harness synthesis completes (e.g., after exhausting all feasible call sequence mutations or reaching a user-defined timeout), SnakeCharmer's resulting harnesses are ready for final post-processing prior to fuzzing (§ 4.4).

4.4 Phase 3: Post-processing and Fuzzing

With harnesses synthesized, SnakeCharmer begins preparing them for eventual use by Python fuzzers such as Atheris [53]. To ensure that only true library bugs are propagated to the fuzzer, SnakeCharmer embeds its learned set of expected exceptions (§ 4.3.4) into an *exception filtering module*, which suppresses benign exceptions during harness execution, while allowing only failures corresponding to potential bugs to surface to the fuzzer. We describe this process below.

4.4.1 ***Exception Filtering Module.*** Informed by both its static (§ 4.2) and dynamic (§ 4.3.4) analyses, SnakeCharmer augments all generated harnesses with library-tailored exception filtering that examines runtime-observed exceptions' origin, content, and context, suppressing those arising from intended library behavior on malformed inputs. Specifically, SnakeCharmer applies the following criteria to identify benign Python- and native-side library exceptions to ignore during fuzzing:

- **Filtering Python-side exceptions:** For exceptions raised during Python-side library execution, SnakeCharmer enforces that harnesses ignore **(1)** library-defined *custom* exceptions (e.g., Jinja2 [35]'s `TemplateSyntaxError`); **(2)** Python built-in exceptions explicitly raised by the *library* itself (i.e., Astroid's [15] raising of `ValueError` in `extract_node()`); as well as **(3)** dynamically observed (§ 4.3.4) *interpreter*-raised Python built-in exceptions seen on invalid inputs (i.e., when libraries defer raising of input-validation-related errors to the interpreter instead [7]).
- **Filtering native-side exceptions:** For runtime exceptions surfaced from Cython-based library components, SnakeCharmer inspects Python-level tracebacks to identify and suppress those that map to explicit error checks (e.g., `raise` or `assert` statements) in the library's Python code. For exceptions raised by Python/C API functions (e.g., `PyErr_SetString()`, `PyErr_Format()`)—which expose only Python-level error information and omit native C/C++ call stacks—SnakeCharmer instead compares runtime-encountered exceptions against the exception type-message pairs it previously captured during native code parsing (§ 4.2.2), suppressing those matching the library's native-raised exceptions (e.g., Pillow's [34] "`tile cannot extend outside image`" in Figure 7).

To filter exceptions, SnakeCharmer injects a custom module into each harness that leverages Python's `traceback` tooling to parse exception types, messages, and source locations (where possible), suppressing those identified as benign expected errors, while allowing only true errors (e.g., division-by-zero exceptions or native-code crashes) to surface to the fuzzer. As our C/C++ exception differentiation focuses on *explicit* error messages (e.g., Figure 7), alternative native-side exception mechanisms (e.g., message-less signaling via `PyErr_SetNone()`) remain unsupported in our current prototype of SnakeCharmer. Following integration of the exception filtering module, SnakeCharmer's generated harnesses are ready for fuzzing via Python fuzzers like Atheris [53].

4.5 Implementation

We implement SnakeCharmer as a prototype framework, extending the C-targeting harnessing engine OGHarn [76] with an additional 2.5K lines of code for Python library parsing, harness synthesis, and exception filtering. We detail several key implementation details as follows.

Library Parsing: SnakeCharmer retrieves Python-level code artifacts via the Inspect [26], Typing [45], PkgUtil [5], and ImportLib [11] utilities; and parses hybrid libraries' C/C++ code components via the Tree-sitter [44] abstract syntax tree parser.

Harness Synthesis Internals: We implement a variety of custom Python classes and objects to support harnessing, including abstractions for function signatures, exceptions, recovered type information, and tracking of generated candidate call sequences.

Collecting Code Coverage: To trace code coverage of libraries' Python components, we utilize the Coverage.py [19] module, while turning to fuzzing-standard Clang-instrumented coverage collection for libraries' C/C++ components [51, 74].

5 Evaluation

Our evaluation of SnakeCharmer is guided by the following high-level research questions:

Q1: Does SnakeCharmer accurately synthesize harnesses that uphold libraries' semantics?
Q2: Can SnakeCharmer reach higher code coverage in pure and hybrid Python libraries?
Q3: Is SnakeCharmer effective at uncovering bugs in pure and hybrid Python libraries?

5.1 Experiment Configuration

To evaluate SnakeCharmer's ability to effectively harness Python libraries, we compare it against today's only automated harnessing approaches for Python software, PyRTFuzz [62] and OSS-Fuzz-Gen [12]. We further compare SnakeCharmer's auto-generated harnesses to the expert-written,

actively fuzzed harnesses provided by both OSS-Fuzz [75] and PolyFuzz [61]. Below details our benchmark selection, competitor-specific setup, and results post-processing procedures.

Library	Category	# Funcs	Hybrid?	Types?	SnakeCharmer		OSS-Fuzz-Gen	OSS-Fuzz ∪ PolyFuzz
					# Harnesses	Time/Harness	# Harnesses	# Harnesses
Astroid [15]	Standalone	4266	No	Yes	10	144.00 min	7	1
AutoFlake [9]	Standalone	52	No	Yes	52	4.00 min	✗	1
Babel [16]	Standalone	533	No	Yes	56	25.71 min	15	2
Bleach [17]	Standalone	570	No	No	78	18.46 min	6	4
DateUtil [13]	Standalone	214	No	No	5	23.20 min	✗	3
H5py [23]	Standalone	974	Yes	No	14	102.86 min	✗	1
Hiredis-py [25]	Standalone	18	Yes	No	6	0.67 min	✗	1
Jinja2 [35]	Standalone	1414	No	Yes	83	10.10 min	20	3
LXML [28]	Standalone	1902	Yes	No	67	21.49 min	4	6
MsgPack [29]	Standalone	58	Yes	No	26	15.85 min	20	4
NumExpr [30]	Standalone	156	Yes	No	5	9.60 min	✗	1
PikePDf [33]	Standalone	666	Yes	Yes	74	19.46 min	✗	1
Pillow [34]	Standalone	3803	Yes	Yes	502	2.87 min	✗	6
Psycopg2 [36]	Standalone	1397	Yes	No	57	25.26 min	15	1
PyCParser [37]	Standalone	609	No	No	35	41.14 min	15	1
PyJSON5 [38]	Standalone	41	No	No	5	288.00 min	16	1
PyYAML [42]	Standalone	1981	Yes	No	24	60.00 min	13	4
SimpleJSON [43]	Standalone	41	Yes	No	43	33.49 min	13	1
AST [7]	CPython	252	Yes	No	15	96.00 min	✗	✗
Email [20]	CPython	1854	Yes	No	35	41.14 min	✗	✗
String [2]	CPython	15	Yes	No	8	41.88 min	✗	✗

Table 1. Our 21 real-world benchmarks and their categories (standalone or a CPython core library); total exposed functions; whether or not they are hybrid (containing both Python *and* C/C++ components); whether or not they have built-in type annotations; SnakeCharmer's total 24-hour generated harnesses; SnakeCharmer's mean time spent (in minutes) generating each harness; OSS-Fuzz-Gen's total 24-hour generated harnesses; and total manually written harnesses across OSS-Fuzz and PolyFuzz. ✗ = API unsupported by that approach.

Benchmark Selection: To ensure a rigorous evaluation of SnakeCharmer, we select **21** diverse, real-world Python libraries shown in Table 1, spanning both pure-Python libraries as well as Python libraries with native C/C++ extensions, and libraries with and without type annotations. Of these 21 libraries, 3 are core CPython interpreter libraries (AST [7], Email [20], and String [2]) while 18 are third-party libraries (e.g., Astroid [15]). Consistent with PyRTFuzz's evaluation [62], we use CPython v3.9.15 for all PyRTFuzz and SnakeCharmer campaigns targeting CPython core libraries.

SnakeCharmer Setup: All pre-fuzzing stages of SnakeCharmer (Figure 5) are configured to run single-threaded on one core, with the only exception being its correctness analysis (§ 4.3.3), which we configure to operate concurrently and collect feedback for up to *five* harnesses candidates at a time. To bootstrap correctness analysis, we seed SnakeCharmer with 5-11 publicly sourced seed files per library, spanning both well-formed as well as malformed (e.g., random or partially formed) inputs. Consistent with prior literature [63, 76, 81], we allocate SnakeCharmer one 24-hour harness generation campaign per library, terminating earlier once its worklist of candidate harness mutations is exhausted, and subsequently fuzz generated harnesses for five 24-hour trials each.

OSS-Fuzz-Gen Setup: While OSS-Fuzz-Gen makes many of its generated fuzzing harnesses publicly available [32], none of its Python library harnesses are released online at the time of writing. Hence, we match SnakeCharmer's setup by deploying OSS-Fuzz-Gen, with GPT-4 as its supporting model, for one harness-generation campaign of up to 24 hours per supported benchmark, followed by five 24-hour fuzzing campaigns on its generated harnesses.

PyRTFuzz Setup: As PyRTFuzz's library fuzzing and harness generation procedures are one and the same—namely, it fuzzes *as* it synthesizes harnesses—we follow prior literature [48, 76] and perform five harness-generation and fuzzing campaigns for each supported benchmark. Given that PyRTFuzz ordinarily targets *all* CPython libraries simultaneously, we configure it to fuzz one benchmarked library at a time. We seed all PyRTFuzz's harness generation campaigns with their required CPython API specifications provided by PyRTFuzz's own source code repository [62].

Data Collection and Analysis: Following the fuzzing evaluation standard set by Klees et. al [60], we perform all fuzzing campaigns for five 24-hour trials. We configure each competitor with the same corpus of seed inputs utilized during SnakeCharmer's harness generation for each library (§ 4.3.3), except for PyRTFuzz, which operates entirely without seeds. We deploy all experiments and data analysis across five 24-core Ubuntu 22.04 workstations, each with 64G RAM and an Intel i9-12900K CPU. All reported means are calculated as geometric means.

5.2 Q1: Harnessing Correctness

Below, we evaluate SnakeCharmer's accuracy in recovering libraries' intended exception semantics, as well as type information in the absence of developer-provided annotations.

5.2.1 ***Exception Handling Correctness.*** As § 3.2 details, failure to prune benign runtime exceptions risks derailing fuzzing with false-positive bug reports. To evaluate whether SnakeCharmer correctly identifies and filters *unwanted* exceptions unique to each library, we compare its exception handling behavior against the ground-truth exception handling in libraries' expert-written harnesses. For each library, we replace the `except` clauses in its expert-written harnesses with calls to SnakeCharmer's per-library exception filtering module (§ 4.4.1), and then execute both the original ground-truth and modified harnesses on an identical input corpus derived from five independent 24-hour fuzzing campaigns, flagging any divergences in the exceptions they surface to the fuzzer.

Library	Total	Surfaced	FP	FN	TP	TN	Accuracy	Precision	Recall
Astroid	3247	18	0	1327*	18	1902	0.59	1.00	0.01
AutoFlake	1208	1208	0	0	1208	0	1.00	1.00	1.00
Babel	0	0	0	0	0	0	n/a	n/a	n/a
Bleach	0	0	0	0	0	0	n/a	n/a	n/a
DateUtil	24748	6929	50	6480*	6879	11339	0.74	0.99	0.51
H5py	7	0	0	0	0	7	1.00	n/a	n/a
Hiredis-py	8496	5612	0	0	5612	2884	1.00	1.00	1.00
Jinja2	62680	10338	9846	0	492	52580	0.84	0.05	1.00
LXML	24826	132	0	0	132	24694	1.00	1.00	1.00
MsgPack	101934	5304	5304	0	0	96630	0.95	0.00	n/a
NumExpr	2042	2042	15	0	1444	584	0.99	0.99	1.00
PikePDF	2265	935	0	0	935	1330	1.00	1.00	1.00
Pillow	7077	132	132	0	0	6945	0.98	0.00	n/a
Psycopg2	120	23	23	0	0	97	0.81	0.00	n/a
PyCParser	16763	78	0	0	78	16685	1.00	1.00	1.00
PyJSON5	1975	36	36*	0	0	1939	0.98	0.00	n/a
PyYAML	31865	1989	194	0	1795	29876	0.99	0.90	1.00
SimpleJSON	39079	543	543	0	0	38526	0.99	0.00	n/a
SnakeCharmer's mean exception filtering correctness:							**93%**	**60%**	**85%**
Adjusted for incorrect ground-truth exception handling:							**97%**	**71%**	**100%**

Table 2. Total runtime exceptions triggered per library vs. the number SnakeCharmer *surfaces* to the fuzzer as probable bugs. We categorize mismatches by manually identifying false positives (FP) and negatives (FN) relative to expert-written harnesses' ground-truth exception handling. ✱ = instances where ground-truth exception handling is *incorrect* and omitted in our *adjusted* measurement; **n/a** = zero exceptions are raised.

Table 2 lists (1) the total number of exceptions encountered, (2) the subset of exceptions surfaced to the fuzzer by SnakeCharmer, and (3) SnakeCharmer's overall accuracy, precision, and recall in exception reporting. We classify the underlying causes of divergences into three core categories.

- **Interpreter-raised exceptions (~66%):** Most exception divergences arise from libraries deferring raising of Python built-in exceptions to the interpreter—a common practice for signaling input-validation errors such as malformed arguments or type errors (§ 3.2). While SnakeCharmer often identifies such expected exceptions during dynamic exception recovery (§ 4.3.4), completeness hinges on whether its seed corpus of invalid inputs *reaches* them. Unfortunately, in these cases, SnakeCharmer's seed inputs were insufficient to flag all benign exceptions prior to fuzzing.
- **Externally-raised C/C++ exceptions (~1%):** One instance where SnakeCharmer's exceptions diverge from expert-written harnesses' is the Pillow [34] library, which links external C components from FreeType [21]. Should these FreeType components encounter an error, Pillow, via its own C extension, raises a corresponding Python exception, embedding the FreeType-reported error message. However, because SnakeCharmer's native code parsing (§ 4.2.2) is limited to only *the library's own* C/C++ components—and therefore excludes external C/C++ dependencies—it fails to recognize these FreeType-raised exceptions as expected error behavior in Pillow.
- **Incorrect expert-written handling (~33%):** Through manual analysis, we find that several apparent divergences in SnakeCharmer's exception handling relative to expert-written harnesses—specifically, 36 false positives in PyJSON5, and 1,327 and 6,480 false negatives in Astroid and DateUtil, respectively—in fact reflect *valid* behavior, arising from incorrect exception handling in the ground-truth expert-written harnesses. Development artifacts confirm that the OSS-Fuzz harnesses for Astroid [8] and DateUtil [10] fail to ignore benign exceptions (tokenize.TokenError and decimal.InvalidOperation, respectively), while the PyJSON5 harness incorrectly suppresses ValueError exceptions, which developers confirmed as real bugs in response to our bug reports [39].

After adjusting for the discrepancies stemming from incorrect ground-truth exception handling in expert-written harnesses, we find that SnakeCharmer's harnesses filter exceptions with an overall **97% accuracy** relative to libraries' corrected expert-written harnesses.

5.2.2 ***Type Recovery Correctness.*** To assess SnakeCharmer's precision in inferring library type information *without* the help of type annotations (§ 4.3.4), we strip all annotations from the six benchmarks that provide them (Table 1), regenerate their harnesses, and evaluate SnakeCharmer's inferred function argument types on these annotation-stripped library versions. Consistent with prior work [79, 80], we treat the original developer-added type annotations in these six libraries as ground truth; and mark a SnakeCharmer-inferred type as correct if the ground-truth type is among the successfully resolved types for the corresponding function argument. Table 3 lists our results.

Library	Total	Resolved	Correct	Precision	Recall
Astroid	285	246	231	0.94	0.86
AutoFlake	50	50	48	0.96	1.00
Babel	279	261	232	0.89	0.93
Hiredis-py	2	2	2	1.00	1.00
Jinja2	416	328	312	0.95	0.78
PikePDF	80	68	64	0.94	0.84
Pillow	317	260	246	0.95	0.81
SnakeCharmer's mean type recovery correctness:				**95%**	**88%**

Table 3. Total per-library harnessed function arguments alongside those resolved by SnakeCharmer, and precision and recall versus manual type annotations.

Overall, we see that SnakeCharmer achieves **95% precision** in inferring correct argument types. In the remaining cases, we identify two recurring sources of error, described below:

- **Passive argument use:** Type mismatches often go undetected when functions accept arguments but perform no meaningful operations on them (e.g., constructors that merely assign constant values to attributes). In the absence of observable runtime side effects, incorrect types pass silently during correctness analysis (§ 4.3.3), potentially delaying observable failures to downstream calls.
- **Equivalent behavior:** Some mismatches arise when SnakeCharmer-inferred types behave similarly to library-expected types. For example, Pillow's `read_mk()` expects a `tuple[int,int]`, yet SnakeCharmer instead provides a `list[int]`. Because the function only accesses elements by index, this substitution does not trigger a failure, thus masking the underlying type mismatch.

In summary, although Python's flexible semantics can occasionally mask type errors, SnakeCharmer consistently infers correct types with high accuracy, demonstrating its effectiveness in Python harness generation **even in the absence of developer-provided type annotations**.

Q1: SnakeCharmer reconstructs key library semantics with accuracy comparable to expert-written harnesses, enabling high-quality harness generation even for Python libraries with poorly-defined semantics.

5.3 Q2: Library Code Coverage

To evaluate fuzzing code coverage, we fuzz SnakeCharmer's final per-library generated harnesses alongside those produced by PyRTFuzz, OSS-Fuzz-Gen, and all available OSS-Fuzz and PolyFuzz harnesses, measuring the coverage achieved by each. However, as Table 1 shows, SnakeCharmer generates dozens to hundreds of harnesses per library within 24 hours. To ensure a manageable comparison set, we follow prior work [76] and retain only the top-10 highest-coverage harnesses per library (or fewer when fewer are produced), ranked by unique code coverage over the library's seed inputs (§ 5.1); and apply the same selection strategy to OSS-Fuzz-Gen.

After completion of all fuzzing campaigns, we replay each competitor's fuzzer-generated inputs on their corresponding harnesses while measuring runtime code coverage. We use Python's Coverage.py [19] module to trace branch coverage of pure-Python components, and Atheris's [53] Clang-based tooling for basic block coverage of any C/C++ extensions. Table 4 reports SnakeCharmer's mean coverage relative to all competitors across five trials per benchmark. Following Klees et al. [60], we assess statistical significance using the Mann-Whitney U test at the $p = 0.05$ level.

As Table 4 highlights, SnakeCharmer achieves mean total coverage of **1.87**×, **1.78**×, **1.40**×, and **1.48**× that of expert-written harnesses from OSS-Fuzz and PolyFuzz, and automatically generated harnesses from OSS-Fuzz-Gen and PyRTFuzz, respectively. Unfortunately, as PyRTFuzz supports only CPython libraries, it cannot harness any of our remaining 18 third-party libraries (Table 1). Moreover, we find that OSS-Fuzz-Gen supports only 11 of our 21 benchmarks as it is limited to just those pre-integrated with its underlying OSS-Fuzz tooling [3, 75]. Comparatively, **SnakeCharmer accommodates all Python libraries**, irrespective of integration on OSS-Fuzz.

We also observe that SnakeCharmer reaches more C/C++ code than expert-written harnesses in **9 out of 10** hybrid libraries, including 2 cases where prior harnesses fail to exercise *any* C/C++ code. In further comparing each competitor's *uniquely* covered code—namely, the library code covered by that approach but not by any others—we find that SnakeCharmer achieves unique coverage of **16.89**×, **49.75**×, **2.38**×, and **3.12**× that of OSS-Fuzz, PolyFuzz, OSS-Fuzz-Gen and PyRTFuzz, respectively. Table 5 and Table 6 further present the *raw* total code coverage as well as the raw coverage specific to C/C++ components, respectively, for all competing approaches.

In examining OSS-Fuzz-Gen's frequent low coverage, we identify two main causes. First, its target exploration often *misses* key library functions, ultimately harnessing only **32%** of data-consuming functions targeted across all expert-written harnesses, compared to SnakeCharmer's **68%**. Second, we observe that OSS-Fuzz-Gen's harnesses often contain critical *semantic errors* that impede deeper

Library	SnakeCharmer/OSS-Fuzz				SnakeCharmer/PolyFuzz				SnakeCharmer/OSS-Fuzz-Gen				SnakeCharmer/PyRTFuzz			
	Total	C/C++	Unique	C/C++	Total	C/C++	Unique	C/C++	Total	C/C++	Unique	C/C++	Total	C/C++	Unique	C/C++
AST	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	**3.20 ×**	**1.68 ×**	**34.35 ×**	**12.49 ×**
Astroid	**2.09 ×**	n/a	**127.7 ×**	n/a	✗	✗	✗	✗	**2.69 ×**	n/a	**55.53 ×**	n/a	✗	✗	✗	✗
AutoFlake	**1.38 ×**	n/a	**89.36 ×**	n/a	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Babel	**1.52 ×**	n/a	**18.84 ×**	n/a	✗	✗	✗	✗	**1.26 ×**	n/a	**0.40 ×**	n/a	✗	✗	✗	✗
Bleach	**1.40 ×**	n/a	**4,742 ×**	n/a	**1.40 ×**	n/a	**862.2 ×**	n/a	**1.34 ×**	n/a	**5.75 ×**	n/a	✗	✗	✗	✗
DateUtil	**1.04 ×**	n/a	**1.93 ×**	n/a	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Email	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	**1.04 ×**	**0.88 ×**	**1.12 ×**	**0.34 ×**
H5py	**5.10 ×**	**3.66 ×**	**25.33 ×**	**16.69 ×**	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Hiredis	**1.48 ×**	**1.02 ×**	**38.33 ×**	**2.86 ×**	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Jinja2	1.01 ×	n/a	**1.28 ×**	n/a	✗	✗	✗	✗	**1.10 ×**	n/a	**3.11 ×**	n/a	✗	✗	✗	✗
LXML	**3.70 ×**	**1.93 ×**	**7.03 ×**	**2.68 ×**	✗	✗	✗	✗	**4.89 ×**	**2.79 ×**	**78.07 ×**	**1,093 ×**	✗	✗	✗	✗
MsgPack	**2.79 ×**	**1.28 ×**	**10.32 ×**	**0.09 ×**	**2.06 ×**	**1.03 ×**	**1.99 ×**	**0.02 ×**	**1.94 ×**	**0.97 ×**	**2.01 ×**	**0.14 ×**	✗	✗	✗	✗
NumExpr	**1.67 ×**	0.98 ×	**59.54 ×**	0.83 ×	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
PikePDF	**3.48 ×**	**3.48 ×**	**3.55 ×**	**3.55 ×**	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Pillow	**1.27 ×**	**2.33 ×**	**4.33 ×**	**15.91 ×**	**1.19 ×**	**2.13 ×**	**7.88 ×**	**24.10 ×**	✗	✗	✗	✗	✗	✗	✗	✗
Psycopg2	**7.56 ×**	∞	**5.55 ×**	∞	✗	✗	✗	✗	**0.40 ×**	**1.49 ×**	**0.33 ×**	**1.93 ×**	✗	✗	✗	✗
PyCParser	1.03 ×	n/a	1.19 ×	n/a	✗	✗	✗	✗	**1.93 ×**	n/a	**4.07 ×**	n/a	✗	✗	✗	✗
PyJSON5	**1.32 ×**	n/a	∞	n/a	✗	✗	✗	✗	**0.73 ×**	n/a	**0.13 ×**	n/a	✗	✗	✗	✗
PyYAML	**1.13 ×**	∞	**1.60 ×**	∞	**1.70 ×**	∞	**130.0 ×**	∞	**1.15 ×**	**0.38 ×**	**0.81 ×**	**0.23 ×**	✗	✗	✗	✗
SimpleJSON	**1.91 ×**	**1.15 ×**	**432.5 ×**	**17.50 ×**	**3.07 ×**	**2.06 ×**	**173.0 ×**	**7.00 ×**	**1.33 ×**	**1.26 ×**	**1.57 ×**	**2.06 ×**	✗	✗	✗	✗
String	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	**0.97 ×**	**0.92 ×**	**0.78 ×**	**0.46 ×**
MEAN:	**1.87 ×**	**1.74 ×**	**16.89 ×**	**3.14 ×**	**1.78 ×**	**1.65 ×**	**49.75 ×**	**1.50 ×**	**1.40 ×**	**1.14 ×**	**2.38 ×**	**2.68 ×**	**1.48 ×**	**1.11 ×**	**3.12 ×**	**1.24 ×**

Table 4. SnakeCharmer's total and uniquely exercised library code coverage relative to the union of expert-written harnesses (i.e., all harnesses from OSS-Fuzz and PolyFuzz) as well as OSS-Fuzz-Gen and PyRTFuzz. For hybrid libraries, we further report the code coverage exercised within their C/C++ components alone. **n/a** = library has no C/C++; ∞ = SnakeCharmer reaches code coverage but competitor reaches *zero*; ✗ = API unsupported by that approach. Statistically significant changes (i.e., Mann-Whitney U test $p < 0.05$) are **bold**.

Library	SnakeCharmer		OSS-Fuzz		PolyFuzz		OSS-Fuzz-Gen		PyRTFuzz	
	Total	Unique	Total	Unique	Total	Unique	Total	Unique	Total	Unique
AST	14336	10154	✗	✗	✗	✗	✗	✗	4477	296
Astroid	14271	7052	6834	55	✗	✗	5311	127	✗	✗
AutoFlake	3179	894	2296	10	✗	✗	✗	✗	✗	✗
Babel	4508	923	2960	49	✗	✗	3566	2307	✗	✗
Bleach	6905	1897	4928	0.4	4919	2	5141	330	✗	✗
DateUtil	3880	282	3744	146	✗	✗	✗	✗	✗	✗
Email	11523	4235	✗	✗	✗	✗	✗	✗	11061	3773
H5py	2300	1925	451	76	✗	✗	✗	✗	✗	✗
Hiredis	480	161	323	4	✗	✗	✗	✗	✗	✗
Jinja2	4642	519	4609	405	✗	✗	4208	167	✗	✗
LXML	7910	5949	2141	847	✗	✗	1616	76	✗	✗
MsgPack	1242	394	445	38	604	198	639	196	✗	✗
NumExpr	1958	798	1174	13	✗	✗	✗	✗	✗	✗
PikePDF	1086033	1077345	312003	303315	✗	✗	✗	✗	✗	✗
Pillow	11933	2099	9403	485	10029	266	✗	✗	✗	✗
Psycopg2	541	386	72	70	✗	✗	1337	1182	✗	✗
PyCParser	2386	338	2324	285	✗	✗	1237	83	✗	✗
PyJSON5	1889	332	1430	0	✗	✗	2592	2463	✗	✗
PyYAML	3640	598	3232	374	2138	5	3152	742	✗	✗
SimpleJSON	927	346	487	1	302	2	699	221	✗	✗
String	4330	551	✗	✗	✗	✗	✗	✗	4481	702

Table 5. SnakeCharmer's total and uniquely exercised raw library code coverage versus expert-written harnesses' (i.e., OSS-Fuzz and PolyFuzz), OSS-Fuzz-Gen, and PyRTFuzz. ✗ = API unsupported by that approach.

runtime library coverage: **37%** of OSS-Fuzz-Gen's harnesses fail due to incorrectly constructed function arguments (i.e., valid calls with invalid argument values, such as passing a `str` where

Library	SnakeCharmer		OSS-Fuzz		PolyFuzz		OSS-Fuzz-Gen		PyRTFuzz	
	Total	Unique	Total	Unique	Total	Unique	Total	Unique	Total	Unique
AST	7326	3216	✗	✗	✗	✗	✗	✗	4367	258
Email	5685	402	✗	✗	✗	✗	✗	✗	6483	1200
H5py	1641	1268	449	76	✗	✗	✗	✗	✗	✗
Hiredis	329	12	321	4	✗	✗	✗	✗	✗	✗
LXML	4054	2187	2101	816	✗	✗	1455	2	✗	✗
MsgPack	539	3	422	38	524	169	174	24	✗	✗
NumExpr	61	6	62	7	✗	✗	✗	✗	✗	✗
PikePDF	1084690	1076680	311260	303250	✗	✗	✗	✗	✗	✗
Pillow	1940	993	834	62	911	41	✗	✗	✗	✗
Psycopg2	189	129	0	0	✗	✗	127	67	✗	✗
PyYAML	271	127	0	0	0	0	706	562	✗	✗
SimpleJSON	430	14	373	1	209	2	340	7	✗	✗
String	4075	305	✗	✗	✗	✗	✗	✗	4437	668

Table 6. SnakeCharmer's total and uniquely exercised raw C/C++ code coverage versus expert-written harnesses' (i.e., OSS-Fuzz and PolyFuzz), OSS-Fuzz-Gen, and PyRTFuzz. We accordingly exclude all eight non-hybrid libraries (Table 1) as they do not contain any C/C++ code. ✗ = API unsupported by that approach.

`bytes` are expected), **10%** due to malformed function calls (i.e., invalid call signatures, including incorrect argument counts, missing required keywords, invalid keywords, or nonexistent functions), and **2%** due to unhandled expected exceptions causing premature harness termination. Nevertheless, SnakeCharmer's broader harnessing and precise exception filtering extend effective fuzzing across Python libraries where state-of-the-art approaches such as OSS-Fuzz-Gen fall short.

Although PyRTFuzz occasionally achieves higher coverage than SnakeCharmer, this coverage is often shallow: because PyRTFuzz generates and fuzzes harnesses in short bursts, it prioritizes breadth over depth, covering many functions superficially rather than thoroughly exercising their behavior. Further, PyRTFuzz targets functions in isolation, limiting its effectiveness on libraries with inter-function dependencies across multiple calls. In contrast, SnakeCharmer's data-flow-*aware* harnessing (§ 4.3.1) yields substantially deeper coverage, as seen in the AST [7] library, where it significantly outperforms PyRTFuzz through **more realistic, multi-function call sequences**.

Q2: SnakeCharmer consistently achieves higher code coverage over all competing approaches, highlighting the effectiveness of its automated harness generation across both pure and hybrid Python libraries.

5.4 Q3: Bug Discovery

Finally, we evaluate SnakeCharmer's bug discovery capabilities. As with our code coverage evaluation (§ 5.3), we conduct five 24-hour fuzzing campaigns on SnakeCharmer's top-10 highest-coverage harnesses per library (or fewer if fewer are produced), alongside all available OSS-Fuzz and PolyFuzz harnesses, OSS-Fuzz-Gen, and PyRTFuzz. After fuzzing, we deduplicate crashes using Python exception traces, as well as AddressSanitizer [1] reports for C/C++-side crashes, before triaging and reporting all newly discovered bugs to their respective Python library developers. Table 7 lists each competitor's total and uniquely discovered bugs per library.

Overall, as Table 7 shows, SnakeCharmer finds the highest number of Python library bugs during fuzzing, with **24** in total—including the most uniquely found bugs across nine libraries—surpassing expert-written harnesses, OSS-Fuzz-Gen, and PyRTFuzz, which find 8, 0, and 0 bugs, respectively. In examining competing approaches, we observe that OSS-Fuzz-Gen's harnesses are not only impeded by frequent semantic errors (§ 5.3), but also from indiscriminately catching and suppressing *all* Python exceptions during fuzzing, preventing exception-based failures from reaching the fuzzer whatsoever. Similarly, beyond lacking support for third-party libraries, PyRTFuzz's restricted, function-isolated fuzzing broadly limits its ability to surface more complex bug-triggering call

Library	SnakeCharmer		OSS-Fuzz ∪ PolyFuzz		OSS-Fuzz-Gen		PyRTFuzz	
	Total	Unique	Total	Unique	Total	Unique	Total	Unique
AST	1	1	✗	✗	✗	✗	0	0
Astroid	2	1	1	0	0	0	✗	✗
AutoFlake	2	1	2	1	✗	✗	✗	✗
Babel	2	2	0	0	0	0	✗	✗
Bleach	0	0	0	0	0	0	✗	✗
DateUtil	1	1	0	0	✗	✗	✗	✗
Email	7	7	✗	✗	✗	✗	0	0
H5py	1	1	0	0	✗	✗	✗	✗
Hiredis-py	2	0	2	0	✗	✗	✗	✗
Jinja2	1	0	1	0	0	0	✗	✗
LXML	0	0	1	1	0	0	✗	✗
MsgPack	0	0	0	0	0	0	✗	✗
NumExpr	0	0	0	0	✗	✗	✗	✗
PikePDF	1	1	0	0	✗	✗	✗	✗
Pillow	2	2	0	0	✗	✗	✗	✗
Psycopg2	0	0	0	0	0	0	✗	✗
PyCParser	0	0	0	0	0	0	✗	✗
PyJSON5	1	1	0	0	0	0	✗	✗
PyYAML	1	1	1	1	0	0	✗	✗
SimpleJSON	0	0	0	0	0	0	✗	✗
String	0	0	✗	✗	✗	✗	0	0
Total bugs found:	**24**	**19**	8	3	0	0	0	0

Table 7. Python library bugs found by SnakeCharmer, expert-written harnesses from OSS-Fuzz and PolyFuzz, OSS-Fuzz-Gen, and PyRTFuzz, alongside those uniquely found by each. ✗ = API unsupported by that approach.

sequences. Consequently, both OSS-Fuzz-Gen and PyRTFuzz fall far short of manually written harnesses, underscoring the challenges of automated harnessing for Python libraries.

Library	Error Type	Source File	OSS-Fuzz ∪ PolyFuzz	OSS-Fuzz-Gen	PyRT-Fuzz	Snake-Charmer	Status
AST	SystemError	`ast.py`	✗	✗	✗	✔	👍
Astroid	IndexError	`builder.py`	✗	✗	✗	✔	✔
	IndexError	`brain_typing.py`	✔	✗	✗	✔	✔
AutoFlake	IndexError	`autoflake.py`	✔	✗	✗	✔	✔
	TypeError	`autoflake.py`	✗	✗	✗	✔	✔
Babel	ValueError	`pofile.py`	✗	✗	✗	✔	👍
	IndexError	`pofile.py`	✗	✗	✗	✔	✔
DateUtil	TypeError	`tz.py`	✗	✗	✗	✔	
Email	TypeError	`email/utils.py`	✗	✗	✗	✔	✔
	IndexError	`_parseaddr.py`	✗	✗	✗	✔	✔
	TypeError	`email/utils.py`	✗	✗	✗	✔	✔
	UnboundLocalError	`_header_value_parser.py`	✗	✗	✗	✔	✔
	IndexError	`get_obs_local_part.py`	✗	✗	✗	✔	✔
	AttributeError	`_header_value_parser.py`	✗	✗	✗	✔	✔
	TypeError	`_header_value_parser.py`	✗	✗	✗	✔	✔
H5py	IllegalRead	`H5Faccum.c`	✗	✗	✗	✔	✔
Hiredis-py	IllegalRead	`reader.c`	✔	✗	✗	✔	✔
	SystemError	`reader.c`	✔	✗	✗	✔	✔
Jinja2	SyntaxError	`environment.py`	✔	✗	✗	✔	👍
PikePDF	UnicodeDecodeError	`_methods.py`	✗	✗	✗	✔	✔
Pillow	ZeroDivisionError	`WmfImagePlugin.py`	✗	✗	✗	✔	✔
	UnicodeDecodeError	`PpmImagePlugin.py`	✗	✗	✗	✔	✔
PyJSON5	ValueError	`lib.py`	✗	✗	✗	✔	✔
PyYAML	ValueError	`constructor.py`	✗	✗	✗	✔	
			SnakeCharmer's total bugs found:			**24**	

Table 8. All bugs found by SnakeCharmer. ✔ = newly found bugs since fixed following our reporting; 👍 = newly found bugs confirmed and awaiting fixes; ✔ = bugs previously known by developers and since fixed.

Table 8 lists all 24 bugs found by SnakeCharmer, along with their error type, origin, and current reporting status. Below, we highlight several notable findings from SnakeCharmer's bug discovery.

Bugs in CPython Core Libraries: As detailed in our evaluation setup (§ 5.1), we evaluate both SnakeCharmer and PyRTFuzz on an older CPython release (v3.9.15) supported by PyRTFuzz. On this version, SnakeCharmer finds three bugs subsequently fixed in newer CPython releases, as well as three previously *unknown* bugs in CPython's built-in Email [20] library that remained unfixed until Python 3.14—all of which have since been confirmed and patched following our reporting. Additionally, while fuzzing the third-party library Babel [16], SnakeCharmer found another unique bug in Email (a dependency of Babel), which has also been confirmed and fixed.

Bugs in Hybrid Libraries: SnakeCharmer also exposes two memory-safety bugs in Python libraries' underlying native-code components: a previously known vulnerability (CVE-2018-13867 [66]) in H5py [23], where an out-of-bounds read in its C backend HDF5 1.10.7 [24] is triggered through H5py by calling `clear()` on a `File` object; and a new segmentation fault in HiRedis-py's [25] C backend, where a crash occurs upon invoking `gets()` on a `Reader` object. Both bugs have been confirmed by developers, with the latter fixed following our reporting.

Outcomes of SnakeCharmer-found Bugs: Of SnakeCharmer's 20 newly found bugs, 18 are now confirmed, with 15 since patched by developers. Collectively, these results demonstrate SnakeCharmer's effectiveness in **uncovering both Python-level *and* native-code defects**, underscoring its value in expanding the reach of today's many Python library fuzzing efforts.

> **Q3:** SnakeCharmer enhances bug discovery across Python's vast ecosystem of pure and hybrid libraries, outperforming both expert-written harnesses as well as today's leading automated harnessing solutions.

6 Discussion

Below, we detail and weigh several fundamental limitations of SnakeCharmer's overarching approach as well as our current prototype implementation.

Supporting Other Library Entrypoints. As § 4.3 details, SnakeCharmer targets library entrypoint routines that consume `str` or `bytes` objects, or `IO` interfaces, reflecting the dominant Python harness initialization patterns that we observe in practice: of OSS-Fuzz's [75] 483 Python library fuzzing harnesses, we find that 422 (87%) pass fuzzer-generated input data into their targeted libraries via these common data types. While we see opportunities to extend SnakeCharmer's entrypoint harnessing to a wider range of input types (e.g., `list`, `dict`), such as by leveraging Atheris's `FuzzedDataProvider` API [53] to construct complex data types directly from fuzzer-generated bytestreams, we leave the requisite engineering and reevaluation to future work.

Supporting Other C/C++ Interfaces. On hybrid Python libraries, SnakeCharmer statically analyzes available C/C++ source files to recover key native-side information not visible through Python-only analysis. For functions, SnakeCharmer searches for canonical Python/C API constructs that register native functions and their calling conventions (e.g., `PyMethodDef`, `PyTypeObject`), which we observe are used in 34 of the 40 hybrid Python-C/C++ libraries in OSS-Fuzz. For exceptions, SnakeCharmer retrieves statically defined error messages in common C/C++ exception-related constructs (e.g., `PyErr_SetString()`, `PyErr_Format()`), which we observe appear in 37 of OSS-Fuzz's hybrid Python-C/C++ libraries. In future work, we plan to extend SnakeCharmer's native-code parsing to support additional Python/native interfaces such as the Common Foreign Function Interface (CFFI) [18], and to more comprehensively handle alternative exception-raising mechanisms such as dynamically constructed error messages and message-less signaling via `PyErr_SetNone()`.

Supporting Native Code Beyond C/C++. While developers are increasingly extending Python with alternative native-code languages (e.g., PyO3 [40]'s Python/Rust bridge), our survey of all hybrid Python libraries currently tested by OSS-Fuzz [75] reveals only a single case in which a language other than C/C++ is used, underscoring the continued dominance of C/C++ in practice. We anticipate that supporting additional native-code languages requires minimal language-specific tooling additions to SnakeCharmer's cross-language analysis infrastructure (§ 4.2.2), which currently targets only C/C++-based native-code interfaces. However, given the limited real-world adoption of alternative-language native extensions today, we defer these additions to future work.

Supporting Java Libraries. As vulnerabilities in native-code extensions also impact Java software, automatic harness generation has seen a surge of advancement for Java libraries (e.g., Rubick [82], DARPA AIxCC [49, 59, 84]). Comparatively, despite Python being the *second-largest* class of libraries fuzzed by Google's OSS-Fuzz initiative (262 total Python libraries versus Java's 202 [3]), there remains a relative scarcity of techniques for automating Python library harnessing beyond SnakeCharmer. While we expect that SnakeCharmer's fundamental approach is likely extensible to Java, substantial reengineering and reevaluation are required, which we defer to future work.

Supporting Faster Harness Generation. As shown in Table 1, SnakeCharmer's harness generation time ranges from 0.67 minutes (Hiredis-py [25]) to 288 minutes (PyJSON5 [38]), with a mean of 23.8 minutes per harness across our entire evaluation. Comparatively, SnakeCharmer's harness generation time remains substantially faster than that of related auto-harnessing approach Winnie [58], which sees a self-reported mean harness generation time of 3.0 *hours* per harness. While we expect further speedups are possible by bolstering multiprocessing support across SnakeCharmer's full harness synthesis pipeline, we leave these enhancements to future work.

7 Conclusion

As Python continues to serve as a foundation for much of computing today, the need for scalable, automated solutions that expand correctness and security testing for Python software is more critical than ever. By combining cross-language static analysis with deep runtime inspection, SnakeCharmer automatically synthesizes effective API harnesses that drive fuzzing deeper into both pure and hybrid Python libraries, while reliably distinguishing expected behavior from true bugs. To this end, SnakeCharmer surfaces **24** bugs across 21 real-world Python libraries, including **20** previously unknown issues, while also revealing significantly higher code coverage than both actively fuzzed expert-written harnesses and existing automatic harnessing solutions.

Looking ahead, we expect the principles behind SnakeCharmer to extend naturally beyond Python to other dynamic programming languages—and we posit that SnakeCharmer-like automatic harnessing solutions will broaden fuzzing's reach across today's many critical software ecosystems long overlooked by existing automated testing efforts.

8 Data Availability

To enable future research in Python harnessing, we release our prototype and all artifacts at the following URL: https://github.com/FuturesLab/SnakeCharmer.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under Award No. 2419798, and by the Defense Advanced Research Projects Agency (DARPA) under Award No. FA8750-24-2-0002, Subaward No. GR105409-SUB00001384. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF or DARPA.

References

[1] 2014. AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html
[2] 2019. String: Common String Operations. https://github.com/python/cpython/blob/3.9/Lib/string.py
[3] 2022. Fuzz Introspector: Introspect, Extend and Optimise Fuzzers. https://github.com/ossf/fuzz-introspector.git
[4] 2023. OSS-Fuzz Issue #42527251. https://issues.oss-fuzz.com/issues/42527251
[5] 2023. PKGUtil: Package Extension Utility. https://github.com/python/cpython/tree/3.13/Lib/pkgutil.py
[6] 2023. Syzkaller: An Unsupervised Coverage-guided Kernel Fuzzer.
[7] 2024. AST: Abstract Syntax Trees. https://github.com/python/cpython/blob/3.9/Lib/ast.py
[8] 2024. Astroid Issue #2527. https://github.com/pylint-dev/astroid/issues/2527
[9] 2024. AutoFlake: Removes Unused Imports and Unused Variables as Reported by Pyflakes. https://github.com/PyCQA/autoflake
[10] 2024. DateUtil Pull Request #1262. https://github.com/dateutil/dateutil/pull/1262
[11] 2024. ImportLib: The Implementation of Import. https://github.com/python/cpython/blob/3.13/Lib/importlib/__init__.py
[12] 2024. OSS-Fuzz-Gen: Automated Fuzz Target Generation. https://github.com/google/oss-fuzz-gen
[13] 2024. Python-DateUtil: Useful Extensions to the Standard Python Datetime Features. https://github.com/dateutil/dateutil
[14] 2025. AST: ast.literal_eval. https://docs.python.org/3/library/ast.html#ast.literal_eval
[15] 2025. Astroid: An Abstract Syntax Tree for Python with Inference Support. https://github.com/pylint-dev/astroid
[16] 2025. Babel: A Collection of Tools for Internationalizing Python Applications. https://github.com/python-babel/babel
[17] 2025. Bleach: An Allowed-list-based HTML Sanitizing Library that Escapes or Strips Markup and Attributes. https://github.com/mozilla/bleach
[18] 2025. CFFI: A Foreign Function Interface Package for Calling C Libraries from Python. https://github.com/python-cffi/cffi
[19] 2025. Coverage.py: The Code Coverage Tool for Python. https://github.com/nedbat/coveragepy
[20] 2025. Email: An Email and MIME Handling Package. https://github.com/python/cpython/tree/3.9/Lib/email
[21] 2025. FreeType: A Freely Available Software Library to Render Fonts. https://gitlab.freedesktop.org/freetype/freetype
[22] 2025. H5Py: fuzz_h5f.py. https://github.com/google/oss-fuzz/blob/master/projects/h5py/fuzz_h5f.py
[23] 2025. H5Py: HDF5 for Python. https://github.com/h5py/h5py
[24] 2025. Hierarchical Data Format, Version 5. https://github.com/HDFGroup/hdf5
[25] 2025. Hiredis-py: Python Wrapper for Hiredis. https://github.com/redis/hiredis-py
[26] 2025. Inspect: Inspect Live Objects. https://github.com/python/cpython/blob/3.13/Lib/inspect.py
[27] 2025. LXML: fuzz_xml_parse.py. https://github.com/google/oss-fuzz/blob/master/projects/lxml/fuzz_xml_parse.py
[28] 2025. LXML: Powerful and Pythonic XML Processing Library Combining LibXML2/LibXSLT with the ElementTree API. https://github.com/lxml/lxml
[29] 2025. MsgPack-Python: MessagePack Serializer Implementation for Python. https://github.com/msgpack/msgpack-python/
[30] 2025. NumExpr: Fast Numerical Array Expression Evaluator for Python, NumPy, Pandas, PyTables and More. https://github.com/pydata/numexpr
[31] 2025. NumPy: The Fundamental Package for Scientific Computing with Python. https://github.com/numpy/numpy
[32] 2025. OSS-Fuzz-Gen: Successful Benchmark Results. https://google.github.io/oss-fuzz/research/llms/target_generation/#successful-benchmark-results
[33] 2025. PikePDF: A Python Library for Reading and Writing PDF, Powered by QPDF. https://github.com/pikepdf/pikepdf/
[34] 2025. Pillow: Python Imaging Library. https://github.com/python-pillow/Pillow
[35] 2025. Psycopg2: A Very Fast and Expressive Template Engine. https://github.com/pallets/jinja
[36] 2025. Psycopg2: Python-PostgreSQL Database Adapter. https://github.com/psycopg/psycopg2
[37] 2025. PyCParser: Complete C99 Parser in Pure Python. https://github.com/eliben/pycparser
[38] 2025. PyJSON5: A Python Implementation of the JSON5 Data Format. https://github.com/dpranke/pyjson5
[39] 2025. PyJSON5: Issue #90. https://github.com/dpranke/pyjson5/issues/90
[40] 2025. PyO3: Rust Bindings for the Python Interpreter. https://github.com/PyO3/pyo3/tree/main
[41] 2025. PyPI: The Python Package Index. https://pypi.org/
[42] 2025. PyYAML: A Full-featured YAML Processing Framework for Python. https://github.com/yaml/pyyaml
[43] 2025. SimpleJSON: Simple, Fast, Extensible JSON Encoder/Decoder for Python. https://github.com/simplejson/simplejson
[44] 2025. Tree-sitter: An Incremental Parsing System for Programming Tools . https://github.com/tree-sitter/py-tree-sitter
[45] 2025. Typing: Support for Type Hints. https://github.com/python/cpython/blob/3.13/Lib/typing.py

[46] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: Fuzz Driver Generation at Scale. In *ACM Joint Meeting on Foundations of Software Engineering (FSE)*.

[47] TIOBE Software BV. 2025. TIOBE Index Python Report. https://www.tiobe.com/tiobe-index/python/

[48] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative Fuzzing for Libraries. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[49] Andrew Chin, Dongkwan Kim, Yu-Fu Fu, Fabian Fleischer, Youngjoon Kim, HyungSeok Han, Cen Zhang, Brian Junekyu Lee, Hanqing Zhao, and Taesoo Kim. 2026. OSS-CRS: Liberating AIxCC Cyber Reasoning Systems for Real-World Open-Source Security. (2026). arXiv:arXiv preprint arXiv:2603.08566

[50] Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*.

[51] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[52] Wentao Gao, Van-Thuan Pham, Dongge Liu, Oliver Chang, Toby Murray, and Benjamin I.P. Rubinstein. 2023. Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report). In *International Fuzzing Workshop (FUZZING)*.

[53] Google. 2024. Atheris: A Coverage-guided, Native Python Fuzzer. https://github.com/google/atheris

[54] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-Aware Dataflow Graphs. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.

[55] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed Selection for Successful Fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[56] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *USENIX Security Symposium (USENIX)*.

[57] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. Utopia: Automatic Generation of Fuzz Driver Using Unit Tests. In *IEEE Symposium on Security and Privacy (Oakland)*.

[58] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Network and Distributed System Security Symposium (NDSS)*.

[59] Taesoo Kim, HyungSeok Han, Soyeon Park, Dae R. Jeong, Dohyeok Kim, Dongkwan Kim, Eunsoo Kim, Jiho Kim, Joshua Wang, Kangsu Kim, Sangwoo Ji, Woosun Song, Hanqing Zhao, Andrew Chin, Gyejin Lee, Kevin Stevens, Mansour Alharthi, Yizhuo Zhai, Cen Zhang, Joonun Jang, Yeongjin Jang, Ammar Askar, Dongju Kim, Fabian Fleischer, Jeongin Cho, Junsik Kim, Kyungjoon Ko, Insu Yun, Sangdon Park, Dowoo Baik, Haein Lee, Hyeon Heo, Minjae Gwon, Minjae Lee, Minwoo Baek, Seunggi Min, Wonyoung Kim, Yonghwi Jin, Younggi Park, Yunjae Choi, Jinho Jung, Gwanhyun Lee, Junyoung Jang, Kyuheon Kim, Yeonghyeon Cha, and Youngjoon Kim. 2025. ATLANTIS: AI-driven Threat Localization, Analysis, and Triage Intelligence System. (2025).

[60] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[61] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, and Haipeng Cai. 2023. PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. In *USENIX Security Symposium (USENIX)*.

[62] 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 SIGSAC Conference on Computer and Communications Security (CCS)*.

[63] Yuwei Liu, Junquan Deng, Xiangkun Jia, Yanhao Wang, Minghua Wang, Lin Huang, Tao Wei, and Purui Su. 2025. PromeFuzz: A Knowledge-Driven Approach to Fuzzing Harness Generation with Large Language Models. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[64] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021).

[65] MITRE. 2018. CVE-2018-1000807. https://www.cve.org/CVERecord?id=CVE-2018-1000807

[66] MITRE. 2018. CVE-2018-13867. https://www.cve.org/CVERecord?id=CVE-2018-13867

[67] MITRE. 2020. CVE-2020-1747. https://www.cve.org/CVERecord?id=CVE-2020-1747

[68] MITRE. 2021. CVE-2021-27921. https://www.cve.org/CVERecord?id=CVE-2021-27921

[69] MITRE. 2022. CVE-2022-30595. https://www.cve.org/CVERecord?id=CVE-2022-30595

[70] MITRE. 2024. CVE-2024-261307. https://www.cve.org/CVERecord?id=CVE-2024-26130

[71] MITRE. 2024. CVE-2024-28219. https://www.cve.org/CVERecord?id=CVE-2024-28219

[72] MITRE. 2024. CVE-2024-34359. https://www.cve.org/CVERecord?id=CVE-2024-34359

[73] Python Software Foundation. 2025. Python/C API. https://docs.python.org/3/c-api/index.html

[74] Kosta Serebryany. 2016. Continuous Fuzzing with LibFuzzer and AddressSanitizer. In *IEEE Cybersecurity Development Conference (SecDev)*.
[75] Kostya Serebryany. 2017. OSS-Fuzz: Google's Continuous Fuzzing Service for Open Source Software. In *USENIX Security Symposium (USENIX)*.
[76] Gabriel Sherman and Stefan Nagy. 2025. No Harness, No Problem: Oracle-Guided Harnessing for Auto-Generating C API Fuzzing Harnesses. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.
[77] Fabian Trautsch and Jens Grabowski. 2017. Are There Any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.
[78] Liam Tung. 2025. *Programming Languages: Python Developers Now Outnumber Java Ones.* Technical Report. https://www.zdnet.com/article/programming-languages-python-developers-now-outnumber-java-ones/
[79] Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq Type Inference Using Static Analysis. In *International Conference on Learning Representations (ICLR)*.
[80] Jifeng Wu and Caroline Lemieux. 2024. QuAC: Quick Attribute-Centric Type Inference for Python. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024).
[81] Hanxiang Xu, Wei Ma, Ting Zhou, Yanjie Zhao, Kai Chen, Qiang Hu, Yang Liu, and Haoyu Wang. 2025. CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph. In *IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*.
[82] Cen Zhang, Yuekang Li, Hao Zhou, Xiaohan Zhang, Yaowen Zheng, Xian Zhan, Xiaofei Xie, Xiapu Luo, Xinghua Li, Yang Liu, et al. 2023. Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation. In *USENIX Security Symposium (USENIX)*.
[83] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-Source SDK Libraries. In *USENIX Security Symposium (USENIX)*.
[84] Cen Zhang, Younggi Park, Fabian Fleischer, Yu-Fu Fu, Jiho Kim, Dongkwan Kim, Youngjoon Kim, Qingxiao Xu, Andrew Chin, Ze Sheng, Hanqing Zhao, Brian J. Lee, Joshua Wang, Michael Pelican, David J. Musliner, Jeff Huang, Jon Silliman, Mikel Mcdaniel, Jefferson Casavant, Isaac Goldthwaite, Nicholas Vidovich, Matthew Lehman, and Taesoo Kim. 2026. SoK: DARPA's AI Cyber Challenge (AIxCC): Competition Design, Architectures, and Lessons Learned. (2026). arXiv:arXiv preprint arXiv:2602.07666

Received 2025-09-10; accepted 2025-12-22