ExploTest, a Python Unit Test Carving Tool

by

Randy Zhu

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2026

Abstract

The abstract is a concise and accurate summary of the research contained in the thesis. It states the problem, the methods of investigation, and the general conclusions, and should not contain tables, graphs or illustrations. It must not exceed 350 words, and should contain relevant keywords that will make your thesis more likely to be found in an electronic search. Do not put a separate list of keywords. There must be a single abstract for the entire thesis.

Table of Contents

Appendices

Chapter 1

Introduction

ExploTest is a tool that solves software engineering problems!?

1.1 Problem Statement

Unit tests are the gold standard to test programs to catch analysis or programming errors. An ideal unit test only assesses one behaviour of the program, is repeatable and requires no software engineer intervention. They often aid in the debugging process, allowing software engineers to identify bugs faster, and earlier in the product life cycle. However, the creation of unit tests is often falsely perceived to not contribute to the product's development. Creating these tests also requires effort to gather the inputs to the test, and requires the software engineer to create test oracles (assertions in JUnit style testing frameworks) to model correct behaviour. As programs evolve, these oracles may shift, and inputs to the test may also change.

1.2 Scope of Research

We intend to evaluate 4 research questions enumerated below:

RQ1: To what extent does the unit test carving tool we created generate viable unit tests when applied to popular, open-source Python repositories?

RQ2: What software families and attributes are associated with the differing viabilities of unit tests created by the tool?

RQ3: What is the amount of speedup offered by the unit tests created by the tool compared to their system test runs?

RQ4: What modes of failure occur when the tool fails to create any kind of unit test?

RQ5: What is the procedure to use ExploTest to catch defects in software?

Chapter 2

Background

2.1 Unit Test Carving

Unit test carving, originally called unit test factoring, was introduced by Saff et al. Test carving is described to be “a technique for automatically creating fast, focused unit tests from slow system-wide tests; each new unit test exercises only a subset of the functionality exercised by the system tests” [15, 16]. Test carving techniques typically involve record-and-replay (also called carve-and-replay [4, 5]), recording function executions, serializing the inputs and outputs of the function, then invoking the function again with said captured inputs.

Elbaum et al. [4] coined the term unit test carving, and they abstractly break down unit test carving for a given method m during program execution. Both the program state before the first instruction of m, called s_{pre} and the state after the final instruction of m, called s_{post} are captured. A pair of states before and after the execution of m: (s_{pre}, s_{post}) constitute a unit test for m. As the codebase of a program evolves, m may change over time.

The execution of a carved unit test must first restore s_{pre}, execute m, then compare the state after the execution (s_m) to s_{post}. In common software engineering parlance, the execution of a carved unit test is as follows:

- Restoring $s_{pre} \rightarrow$ test setup; typically found in the fixtures or the `@BeforeEach` part of a unit test.
- Executing $m \rightarrow$ running the function-under-test
- Comparing s_m to $s_{post} \rightarrow$ the test oracle; typically assertions like `assertEquals`.

There are two approaches in capturing s_{pre}, explains Elbaum et al.: state-based and action-based.

State-based capture involves saving s_{pre}, typically serializing s_{pre} to disk, then deserializing it to restore s_{pre}. ExploTest implements this in its pickle-mode. This comes with certain challenges which we will describe later in this thesis. The majority (TODO: CHECK IF THIS IS ACTUALLY TRUE!) of unit testing tools use state-based capture.

Action-based capture involves saving the instructions before s_{pre}, and later replaying those instructions to re-create s_{pre}. This resolves many of the challenges of state-based capture, but is harder to do in practice. ExploTest has a limited version of action-based capture in its argument reconstruction mode.

2.1.1 Challenges in State-based Capture

Evolution and Serialization

Serialization involves storing the state of the program to disk [2]. In turn, when we want to restore the state of the program, we have to load that state from disk. This state is entirely unaware of changes to the source code, which means any changes to the structure of the object in the source code at a later time is not reflected in the serialized format. In turn, this makes the unit tests more brittle to breaking from code evolution.

Serialization Opaqueness

Unlike the code to create objects in source code, reading objects from disk does not at all explain the process that got the object to that serialized state. This makes it much more confusing for maintainers of source code that utilizes serialized state in unit tests to change the unit tests as code evolves, worsening the evolution issue. MicroTestCarver by Deljouyi and Zaidman [3] tries to remedy this issue in their work by serializing to XML, a human-editable file format using the XStream library, however, there is no similar library that produces any kind of human-readable serialization output in Python.

Non-serializable Objects

There are objects that cannot be serialized, which typically have ill-defined representations after deserialization. Currently, the most robust Python serialization library which we elected to use, dill [12] do not support `generator`, `traceback`, and code frame objects. Code frame objects cannot be captured by state-based capture, as the creation of a code frame object entails the creation of native code objects which cannot be done without some action-based capture. Similarly, network sockets and active network connections do not have a meaningful value when deserialized, as the server they connect has likely since closed the connection. Finally, objects which explicitly reject serialization by disabling `__reduce__` cannot be serialized. This family of objects necessitates the use of action-based capture.

2.1.2 Challenges in Action-based Capture

The Object Creation Problem

The Object Creation Problem, defined by Bach et al. [1], is the problem of finding the correct sequences of method calls to construct an object. In Python, creating is objects is not always as simple as calling the `__init__` constructor, instead, objects rely on hierarchies and overridden `__new__`, or other classes like in the factory design pattern.

Arbitrary Object Modification

Explained by Politz et al. [14], the semantics of Python allow for arbitrary modification to object attributes at any time in the programs. The shape of an object, which we define to be the number and names of fields of an object can change at any point in a Python program.

While this is a problem in unmanaged languages like C++, and the authors of [1] discuss this issue, they consider it out of scope of the object creation problem, as it is unidiomatic in C++, and often leads to undefined behaviour. Unfortunately, this technique of arbitrary modification of objects at any point is often used by Python programmers.

This presents challenges to action-based capture, as existing action-based capture unit test carving tools like ARTISAN [6] assume that the shape of an object does not change, and can assume the object

2.2 Prior Work in Test Carving for Python

2.2.1 Pythoscope

Pythoscope, by Kwiatkowski and Clements-Tiberio [9, 10, 13] is an open source Python 2 unit test generation tool, created around 2008.

2.2.2 Auger

[11]

2.2.3 Fuzzing Book Carver

[19]

2.2.4 Genthat

2.3 How ExploTest Performs Unit Test Carving of Python Programs

Originally created by Zihao Huang under the supervision of Dr. Caroline Lemieux, ExploTest is a unit test carving tool used to created unit tests from exploratory test runs. As defined by Kaner [7], exploratory testing is an unscripted technique where the software designer verifies the program's correctness using their own intuition and domain knowledge to uncover defects and create test oracles. Exploratory testing exercise the whole program, and we aim to capture unit tests for specific methods.

Chapter 3

Evaluation

3.1 Evaluation Dataset

3.1.1 Criteria

Since unit tests are used to catch defects in software, it's natural to use software defects to evaluate ExploTest. Therefore, we collected 22 bugs across 12 repositories.

In selecting bugs, we had a few criteria inspired by Pythoscope's [8] creator:

1. Since Python is a popular and general purpose programming language, we want bugs from various types of Python software.
2. Due to design limitations associated with ExploTest, the program with the bug must support Python 3.13.7.
3. Each bug must be easy to reproduce with an associated system test.
4. Each bug must have an associated function that causes the bug — i.e., the bug must be detectable by unit testing a function.
5. The root cause of the bug must be from the source code itself, not due to platform issues (e.g., a bug in the interpreter or operating system).
6. The bug cannot be so severe that the program does not execute or cause other processes to crash.

3.1.2 Dataset

We evaluate ExploTest on 22 individual bugs across 12 repositories:

Table 3.1: Repositories used in the evaluation dataset

Project name	Category	Description
`gunicorn`	Web framework	HTTP server.
`click`	UI library	Library for creating command-line interfaces.
`seaborn`	Data visualization	Data visualization library.
`Pygments`	Source code analysis	Tool for syntax highlighting source code.
`pylint`	Source code analysis	Linter for Python code.
`IPython`	Meta tooling	Enhanced interactive Python REPL.
`httpie`	Command-line utility	Command-line HTTP client and API testing tool.
`Tornado`	Web framework	Web framework.
`black`	Source code analysis	Python source code formatter.
`sanic`	Web framework	Web framework.
`PySnooper`	Meta tooling	Debugging tool that traces function inputs, outputs, and state.
`youtube-dl`	Command-line utility	Command-line tool for downloading videos from online streaming services (e.g., YouTube).

Bugs were selected from the popular BugsInPy bug repository [17] and from selected repositories in QuAC [18], a Python type inference tool.

Each bug consists of three things: the buggy commit, the fixed commit, and a bug report containing a system-test procedure on how to reproduce the bug.

Details about each bug can be found in the appendix, section B.1.

3.2 Environment

The evaluation takes place on an x86_64 computer running Ubuntu 24.04.3. All code is run using Python 3.13.7 in a virtual environment for each bug.

3.3 Answering the RQs

3.3.1 ExploTest Outcomes

The unit tests that ExploTest produces can be of different qualities, or it may not produce a unit test at all due to technical limitations. As discussed in the introduction, unit tests have to exercise the function-under-test (FUT) and then assess its return value with an oracle which in Python is an assertion.

If ExploTest does produce a unit test, we assess 5 different qualities of the unit tests.

Test Qualities

- **Function called**: Does the tool correctly setup and call the required function?
- **Output correct**: When the function-under-test is called by the generated unit test, does it have the same output as the function during the system test?
- **Oracle created**: Are any assertions generated in the test?
- **Passes good code**: Do the assertions generated pass when run on the function-under-test in the correct version of the codebase?
- **Fails on bad code**: Do the assertions generated fail when run on the function-under-test in the buggy version of the codebase?

3.3.2 Flakiness

A unit test is considered flaky when it exhibits both passing and failing results on the same version of the source code without any changes to the code itself. We run the test twice as a rudimentary test for flakiness.

Table C.1 in Appendix **??** outlines the per-bug evaluation results.

3.3.3 Preliminary Evaluation

Preparation

For each bug, we will create a new Python virtual environment using the `venv` module.

We then install the package and its dependencies as instructed by the documentation of the package, along with ExploTest. For seaborn, since the bugs appear as plots, we install `jupyter` and `notebook` to work with plots interactively.

Finding carve target

ExploTest uses an `@explore` decorator to mark functions to carve in the source code. By placing the decorator, the function is transformed to first perform a static analysis of the function's source code (e.g., to get the function name and parameter/package information).

To evaluate RQ1, we can split the dataset into two by bug origin: bugs from BugsInPy and bugs from QuAC. Bugs from BugsInPy come with buggy functions already known, while the causes of bugs from QuAC are indeterminate. While the bugs from QuAC may contain fixes in pull requests that identify the functions that introduce the bug, we refrain from looking at them to help answer RQ5 with a cleaner slate.

Locating carve targets for bugs from BugsInPy

The data provided by BugsInPy already identify the individual lines that cause the bug to occur.

For us, the first thing to do is to place the `@explore` at the nearest function to the lines that cause the bug to occur. However, these functions might not be good candidates for carving for two reasons: (1) parameters might not be serializable, as discussed in chapter 2, or (2), the return value of the function will not allow us to generate an oracle that detects the bug.

Oftentimes, human inspection of the source code allows us to determine the presence of non-serializable objects. We can infer these properties from variable names, for example, variables named `frame` suggest the presence of frame objects which contain pointers to native code that cannot be serialized.

A similar manual source code analysis works to identify return values not conducive to good test oracles. We can inspect the source code for freestanding `return`s that indicate a function that is used for its side effects only.

When we identify a method that is a bad candidate for test carving, we walk up the call graph of the function (i.e., look at the caller and the usage of our target function), and place an `@explore` once we determined that the objects are serializable, or keep looking up.

There are cases where we cannot find any suitable candidates, in which case we consider the unit test to not be carvable with our tool.

Locating carve targets for bugs from QuAC

Answering RQ1

1. Run the non-buggy version of the code and function and observe its output.
2. Use the procedure outlined above to place the `@explore` to mark the function-under-test for carving.
3. Generate the test on the non-buggy version of the code using the system test case:

   Did the tool crash? Were exceptions or errors emitted and what kind?
4. If a non-empty file was produced, inspect the file and run the test using a debugger to determine whether the FUT was reached. Observe the output of the function from the generated the unit test against the output of the system test. If they are the same, we can consider the FUT to be exercised.
5. When running the test, we can observe whether the test passes on the non-buggy code. If it passes, we move and apply the buggy commits. If the test fails on the source it was generated on, we note it down and skip the next step.
6. We then run the test, generated on the non-buggy code on the buggy version of the codebase. The test should not pass, and if it does, we consider the oracle defective.

This procedure should allow us to bucket each created test by the table, and facilitate discussion of failure cases.

Answering RQ2

For RQ2, we define a functional unit test as one that correctly exercises the function under test; it need not create an oracle for the test.

We group individual bugs by their software family, and calculate the ratio of tests that are functional and total number of bugs. A discussion of the results will follow.

Answering RQ3

We time the time it takes to execute the system test with GNU `time` version 1.9 on our environment mentioned in section 3.2. We calculate and discuss three metrics:

- The bug detection speedup, $\frac{T^B_{\text{sys}}}{T^B_{\text{unit}}}$, which is the speedup multiplier offered by the unit test to catch buggy code.

  It is the ratio of the time it takes for a system test on buggy code to run (T^B_{sys}) and the time it takes for a unit test on buggy code (T^B_{unit}).
- The regression verification speedup, $\frac{T^P_{\text{sys}}}{T^P_{\text{unit}}}$, which is the speedup multiplier offered by the unit tests to validate code changes on a correct codebase.

  It is the ratio of the time it takes for a system test on correct code to run (T^P_{sys}) and the time it takes for a unit test on correct code (T^P_{unit}).
- Carving time, which is the difference between the system test running with ExploTest and without ExploTest.

  It is the ratio of the time to run the program while carving and without carving.

Answering RQ4

When carving tests, the most common cause of failure are non-serializable objects. We'll discuss the root cause of these objects and their frequencies. Another failure mode is a defective oracle that either fails on correct code, or oracles that pass on incorrect code.

Answering RQ5

For the assessment of this RQ to remain valid, we purposely discard the bugs found from BugsInPy, since the author was already exposed to the root causes of these bugs. Instead, the author will try to debug and root cause analyze the bugs from QuAC, using ExploTest, and determine if there is a procedure to using ExploTest to find, and potentially patch bugs. A discussion of the technique will follow.

Chapter 4

Results

4.1 Answers to RQs

4.1.1 RQ1: Test quality

In RQ1, we posed: *To what extent does the unit test carving tool we created generate viable unit tests when applied to popular, open-source Python repositories?*

Table C.1 shows ExploTest is able to generate at least 1 test for 16 curated bugs out of 22 bugs from the software repositories we choose. We consider these runs successful. 72% of ExploTest runs were able to create some kind of syntactically valid unit test. The 6 unsuccessful runs are examined further in RQ4.

Out of the 16 runs that produced a unit test, all were able to correctly set up the arguments to the function and successfully called the function. 72% of ExploTest runs create unit tests that when run, reach the function-under-test.

Manual analysis of the returned object's state showed that 14 out of 16 of the functions-under-test exercised by the unit test had the same output as the function-under-test being exercised by the system test. The 2 cases of incorrect function object were due to the custom serialization protocol a parameter used; ExploTest has no control over the customized serialization code that implementors use. 64% of ExploTest runs create unit tests that reach and correctly call the function-under-test the same as a system test would.

AutoAssert was able to create assertions for 12 out of 14 of the tests that correctly call the functions-under-test. 55% of ExploTest runs create unit tests that correctly exercise the function-under-test and have an oracle.

11 out of 12 tests with assertions when run on correct code pass, and only 2 tests fail when run on incorrect code. Only 9% of the tests generated by ExploTest actually expose the bugs they intend to detect. Our approach to generating assertions is likely too naive or conservative. Further discussion on improving the quality of assertions will follow in section 5.

RQ1 Answer: **64% of ExploTest runs create unit tests that correctly exercise the behaviour of their carving targets. Only 9% of runs create tests that expose buggy behaviour. Test quality degrades sharply at the assertion generation phase: only 2 of 14 correctly-excerising tests produce assertions that expose the target bug.**

4.1.2 RQ2: Unit test viability by software attributes and family

RQ2 poses the question: *What software families and attributes are associated with the differing viabilities of unit tests created by the tool?*

Differing viabilities are classified as follows:

- Defective — the function was either not called, or called but produced a result that differs from the system test. There are 8 runs in this category.
- Exercises function — the function was called by the unit test, and its output was the same as the system test. There are 14 runs in this category.
- Uncovers bug — the unit test is able to exercise the function, and can uncover the bug in the buggy version of the repository with its oracle. This category is a subset of the "exercises function" category. There are 2 runs of the test in this category.

Tests which can uncover bugs

First, we discuss the excellent. What attributes of the function-under-test and codebase contribute are associated with the tool creating a test that falls into the "uncovers bug" category?

Observe the test generated when run on the `update_headers` procedure from `httpie` in A.1. The test is able to detect the bug since in the buggy version the check for `value is None` is not present and causes the reference to `value.decode` to throw a `NameError`.

The other test for in the "uncovers bug" category for the `sanic` web framework in A.2 has similar attributes to the first. The function-under-test, `url_for`'s only external data dependencies are on the `re` regular expression library and the `urlencode`, `urldecode` functions from `urllib` library which do not require any previous state to be set up for them.

We note a few things from the two tests that uncover bugs:

- The function-under-test has limited external dependencies for it to function.
- The function-under-test's external dependencies do not require their states to be set; they are stateless, or the default state is sufficient.
- Every type used in the function-under-test is serializable. The `request_headers` parameter is a dictionary of strings and bytes, which are both serializable.
- Even with weak test oracles like `assert return_value is None` can catch bugs, since crash oracles exist.

Tests which can exercise their target function-under-tests

We'll discuss common patterns in ExploTest runs that successfully create a unit test, but cannot detect the bug.

Looking at our 14 runs in this category, we can make a few observations:

- Target function-under-tests have serializable parameters. This is unsurprising, considering the tool uses record-and-replay techniques to carve its tests.
- Carve candidates have clearly defined entry-points.

Defective tests

Defect: No test generated

Defect: Test generated, FUT not correctly exercised

Viability by software family

We aggregate the ratio of successful tests by software category in Table D.1. The Data visualization, Command line utility, and Web framework categories are discarded since there is not enough data to draw a meaningful analysis.

RQ Answer

4.1.3 RQ3: *What is the amount of speedup offered by the unit tests created by the tool compared to their system test runs?*

4.1.4 RQ4: *What modes of failure occur when the tool fails to create any kind of unit test?*

4.1.5 RQ5: *What is the procedure to use ExploTest to catch defects in software?*

Chapter 5

Discussion and Future Work

Bibliography

[1] Thomas Bach, Ralf Pannemans, and Artur Andrzejak. 2020. Determining method-call sequences for object creation in C++. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 108–119.

[2] ISO C++ Standards Committee. [n. d.]. Serialization and Unserialization. `https://isocpp.org/wiki/faq/serialization`

[3] Amirhossein Deljouyi and Andy Zaidman. 2023. Generating understandable unit tests through end-to-end test scenario carving. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 107–118.

[4] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 253–264.

[5] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Matthew Jorde. 2008. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering* 35, 1 (2008), 29–45.

[6] Alessio Gambi, Hemant Gouni, Daniel Berreiter, Vsevolod Tymofyeyev, and Mattia Fazzini. 2023. Action-based test carving for android apps. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 107–116.

[7] Cem Kaner. 2008. A tutorial in exploratory testing. *QUEST* 6 (2008).

[8] Michal Kwiatkowski. 2011. *Capture and analysis of side effects in a running Python program for the purpose of unit test generation*. Master's thesis. Gdańsk University of Technology.

[9] Michal Kwiatkowski and Brian Clements-Tiberio. [n. d.]. `http://pythoscope.wikidot.com/start`

[10] Micha l Kwiatkowski. 2011. Capture and analysis of side effects in a running Python program for the purpose of unit test generation. (2011).

[11] Chris Laffra, John Mather, Adam Johnson, and Jonas Eschle. [n. d.]. Laffra/Auger: Automated Unittest Generation for Python. `https://chrislaffra.blogspot.com/2016/12/auger-automatic-unit-test-generation.html`

[12] Mike McKerns. [n. d.]. Dill package documentation. `https://dill.readthedocs.io/en/latest/index.html`

[13] Michal Mkwiatkowski and Brian Clements-Tiberio. [n. d.]. Mkwiatkowski/Pythoscope: Unit test generator for python. `https://github.com/mkwiatkowski/pythoscope`

[14] Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The full monty. *ACM SIGPLAN Notices* 48, 10 (2013), 217–232.

[15] David Saff, Shay Artzi, Jeff H Perkins, and Michael D Ernst. 2005. Automatic test factoring for Java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 114–123.

[16] David Saff and Michael D Ernst. 2004. Mock object creation for test factoring. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 49–51.

[17] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1556–1560.

[18] Jifeng Wu and Caroline Lemieux. 2024. QuAC: Quick Attribute-Centric Type Inference for Python. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 2040–2069.

[19] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2023. Carving Unit Tests. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. `https://www.fuzzingbook.org/html/Carver.html` Retrieved 2023-11-11 18:18:05+01:00.

Appendix A

Case studies of bugs and generated tests used in evaluation

A.1 Tests and functions-under-test in the "uncovers bug" category

These tests are able to fully expose bugs in their functions-under-test. They have fully-functional oracles, and can correctly set up and call the function-under-test.

Some renaming and elision of unimportant data has been done for the sake of readability.

A.1.1 httpie, bug #3

Generated unit test

```
import os
import dill
import pytest
from httpie import sessions


@pytest.fixture(autouse=True)
def mock_setup():
    import os

    os.environ["RUNNING_GENERATED_TEST"] = "true"


@pytest.fixture
def generate_self():
    self = {
        "headers": {},
        "cookies": {
            "AEC": {
                "expires": 1784594863,
                "path": "/",
                "secure": True,
                "value": "AaJma5sHy...",
            },
            "NID": {
                "expires": 1784851905,
                "path": "/",
                "secure": False,
                "value": "528=Udfh...",
            },
            "__Secure-STRP": {
                "expires": 1769043163,
                "path": "/",
                "secure": True,
                "value": "AD6DogubTSu...",
            },
        },
        "auth": {"password": None, "type": None, "username": None},
        "__meta__": {
            "about": "HTTPie session file",
            "help": "https://github.com/jkbrzt/httpie#sessions",
            "httpie": "1.0.0-dev",
        },
    }
    return self


@pytest.fixture
def generate_request_headers():
    with open(
        "...",
        "rb",
    ) as f:
        request_headers = dill.loads(f.read())
    return request_headers


def test_Session_update_headers(generate_self,
                                generate_request_headers):
    self = generate_self
    request_headers = generate_request_headers
    return_value = sessions.Session.update_headers(self, request_headers)
    assert return_value is None
```

Function-under-test

```
def update_headers(self, request_headers):
    """
    Update the session headers with the request ones while ignoring
    certain name prefixes.

    :type request_headers: dict

    """
    for name, value in request_headers.items():

        # this None-check is not present in the buggy version,
        # causing the crash, which consequently causes the generated
        # test to fail.
        if value is None:
            continue  # Ignore explicitely unset headers

        value = value.decode('utf8')
        if name == 'User-Agent' and value.startswith('HTTPie/'):
            continue

        for prefix in SESSION_IGNORED_HEADER_PREFIXES:
            if name.lower().startswith(prefix.lower()):
                break
        else:
            self['headers'][name] = value
```

A.1.2 sanic, bug #3

Generated unit test

```
@pytest.fixture
def gen_self():
    # note: serialization path has been elided
    with open('...', 'rb') as f:
        self = dill.loads(f.read())
    return self

@pytest.fixture
def gen_view_name():
    view_name = 'bp_app.external'
    return view_name

@pytest.fixture
def gen_kwargs():
    kwargs = {'_external': True}
    return kwargs

@pytest.fixture
def gen_saved_return_value():
    saved_return_value = 'http://bp.example.com/external'
    return saved_return_value

def test_Sanic_url_for(gen_self, gen_view_name,
                       gen_kwargs, gen_saved_return_value):

    self = gen_self
    view_name = gen_view_name
    kwargs = gen_kwargs
    saved_return_value = gen_saved_return_value
    return_value = app.Sanic.url_for(self, view_name, **kwargs)
    assert return_value == saved_return_value
```

Function-under-test

```
def url_for(self, view_name: str, **kwargs):
    kw: Dict[str, str] = {}

    uri, route = self.router.find_route_by_view_name(view_name, **kw)

    host = uri.find("/")
    host, uri = uri[:host], uri[host:]

    out = uri

    matched_params = re.findall(self.router.parameter_pattern, uri)

    kwargs.pop("_method", None)
    anchor = kwargs.pop("_anchor", "")

    external = kwargs.pop("_external", False)
    scheme = kwargs.pop("_scheme", "")
    if scheme and not external:
        raise ValueError("When specifying _scheme, _external must be True")

    scheme = "http"

    query_string = urlencode(kwargs, doseq=True) if kwargs else ""
    out = urlunparse((scheme, netloc, out, "", query_string, anchor))
    return out
```

A.2 Tests and functions-under-test in the "exercises function" category

A.3 Tests and functions-under test in the "defective" category

Appendix B

Links to bugs we evaluated

Table B.1: Bug dataset with commit and issue information

Repo	ID	Buggy Commit	Fixed Commit	Issue ID
black	12	8b340e	b53cb9	468
black	13	0a340e	883689	125
black	15	8a8c58	df2ae3	335
httpie	3	8c33e5	589887	412
PySnooper	2	81868c	814abc	144
PySnooper	3	6e3d79	15555e	2
sanic	1	e7001b	449731	1847
sanic	2	ba9b43	801595	1754
sanic	3	91f6ab	861e87	1788
tornado	10	ecd896	5931d9	2010
tornado	14	81ee31	1d02ed	1469
tornado	15	fdfaf3	ecb3ea	3170*
tornado	4	a8420f	db5290	2521
youtube-dl	12	435e38	e118a8	18961
click	3071	00fadb	1d038f	3071
gunicorn	3144	547f85	70a1e4	3144
ipython	14930	557b8b	1e9a2e	14930
pylint	10373	f798a4	98942b	10373
pylint	10402	f798a4	98942b	10402
seaborn	3542	f2abeb	d00a27	3542
seaborn	3553	f2abeb	d00a27	3553
pygments	2382	77a939	6c187a	2382

Appendix C

Bug results

Table columns

- **Repo**: The repository the bug originates from.
- **ID**: For bugs from BugsInPy, the BugsInPy ID that the bug is associated with. For bugs from QuAC, this is the GitHub issue number that the bug is associated with.
- **# of tests**: The number of unit tests generated by the tool. The tool can generate > 1 test when the FUT is exercised more than once by the system test, or if recursive calls occur. 0 tests are generated when the tool fails to generate any test at all.
- **Function called**: Does the tool correctly setup the required
- **Output correct**: Whether the output of the function-under-test exercised by the unit test is equivalent to the system test.
- **Oracle created**: Whether assertions were created to determine the software functions correctly.
- **Passes good code**: Whether the assertions pass when running on the correct code (i.e., the code the test was generated on).
- **Fails on bad code**: Whether the assertions fail when running on the buggy code (i.e., they detect the bug).

Table C.1: Per-bug evaluation results

Repo	**ID**	**# of tests**	**Function called**	**Output correct**	**Oracle created**	**Passes on good code**	**Fails on bad code**
black	12	490	Yes	Yes	No	N/A	N/A
black	13	3	Yes	Yes	Yes	Yes	No
black	15	1	Yes	Yes	Yes	Yes	No
httpie	3	1	Yes	Yes	Yes	Yes	Yes
PySnooper	2	4	Yes	Yes	Yes	Yes	No
PySnooper	3	1	Yes	Yes	Yes	Yes	No
sanic	1	6	Yes	Yes	Yes	Yes	No
sanic	2	0	N/A	N/A	N/A	N/A	N/A
sanic	3	3	Yes	Yes	Yes	Yes	Yes
tornado	10	0	N/A	N/A	N/A	N/A	N/A
tornado	14	2	Yes	Yes	No	N/A	N/A
tornado	15	1	Yes	Yes	Yes	Yes	No
tornado	4	2	Yes	Yes	Yes	Yes	No
youtube-dl	12	0	N/A	N/A	N/A	N/A	N/A
click	3071	4 P / 4 ARR	Yes	No	No	Yes	No
gunicorn	3144	0	N/A	N/A	N/A	N/A	N/A
ipython	14930	0	N/A	N/A	N/A	N/A	N/A
pylint	10373	1	Yes	Yes	Yes	Yes	No
pylint	10402	2	Yes	Yes	Yes	Yes	No
seaborn	3542	1	Yes	No	No	Yes	No
seaborn	3553	0	N/A	N/A	N/A	N/A	N/A
pygments	2382	1	Yes	Yes	Yes	Yes	No

Appendix D

Percentage of tests created by software category

Table D.1: Percentage of tests created by software category

Category	**# of bugs in category**	**# of valid tests created**	**Success rate**
~~Command-line utility~~	~~2~~	~~1~~	~~50%~~
~~Dataviz~~	~~2~~	~~1~~	~~50%~~
Meta tooling	5	4	80%
Source Code Analysis	4	4	100%
~~UI library~~	~~1~~	~~1~~	~~100%~~
Web framework	8	5	62.5%
Grand Total	**22**		