

PYOHIO 2026 · JULY 26

Testing the Parts You Don't See

Pseudo-Tested Native Code in Python Libraries

Issei Hasegawa

Gregory M. Kapfhammer

Department of Computer and Information Science
Allegheny College



"look inside"

Issei Hasegawa

CS Undergrad · Allegheny College

 <https://github.com/IsseiHasegawa>

 [linkedin.com/in/issei-hasegawa](https://www.linkedin.com/in/issei-hasegawa)

RESEARCH INTERESTS



MLSys



Distributed Systems



Software Engineering

? Your Tests Pass. What Does That Prove?



147 tests passed

every test is green



100% coverage

every line was executed

Both metrics only prove that your program's code **ran**
Neither proves that the tests **checked the result**

Even in this aspirational case, the program **may not** be as **well-tested** as we think

That gap between running and checking is where **pseudo-tested** code hides!



What Is a **Pseudo-Tested** Function?

A **pseudo-tested** function runs during testing, but no test ever checks its result, suggesting that it is not adequately tested.

1. it is **covered** during testing; it actually runs.
2. even if you apply an **extreme transformation** to its body, all tests still pass.

Idea: replace its whole body with a **default return**, and every test still passes!

Test coverage still counts it as covered since the line was executed by the tests.

Covered $\neq$ Tested: coverage shows execution, not correctness!



A Pure-Python Example

```
def is_adult(user):  
    return user.age >= 18  
  
def test_is_adult():  
    is_adult(User(age=25))  
    # result is never asserted
```

Replace the body:

return True

→ **passes**

return False

→ **passes**

Both survive → **is_adult**
function is pseudo-tested

coverage: 100% ✓

actually tested? ✗

*The test does not **assert**, but a standard coverage report does not highlight this!*



Why Pseudo-Testedness Matters to You



Developers

Find the blind spots that green tests hide, and where a missing assertion belongs.



Researchers

Pseudo-tested detection has mostly focused on Java. Native code in Python is unexplored!



Educators

A vivid way to teach “covered $\neq$ tested”: 100% coverage does not mean full verification.



Students

Build the instinct to ask: what does this test actually demonstrate about correctness?



How Do You Detect Pseudo-Testedness?

Break the function body on purpose. If the tests still pass, they weren't testing it. Developers have an actionable insight!

Extreme mutation testing: replace a function's **whole body** with a default return, not just tweak one operator. First look for the functions most in need of better tests.

```
int clamp(int x, ...) {  
    if (x < lo) ...  
    // real logic  
}
```



```
int clamp(int x, ...) {  
    return 0;  
}  
// did any test notice?
```

Use C since focus is on pseudo-tested native code in Python libraries



What Are the Default Return Values?

The mutant is chosen by the function's **return type**, so it always compiles.

RETURN TYPE	MUTANT (Default return values)
void	<code>(empty body)</code>
int, bool	<code>return 0; return 1;</code>
pointer	<code>return NULL;</code>
string	<code>return " "; return "A";</code>
struct	<code>return (JSOBJ){0};</code>

Type-directed

Tree-sitter reads the return type, so every mutant still compiles. Since the focus is on the C/C++ backend of a Python project, all function mutants must compile!

Two values

Some return types need more than one default value, because a single value could accidentally match what the function really returns and the tests are checking.

void has nothing to return

The function's body is emptied, so a function that was already empty survives by definition.

When a function has more than one mutant, we call it **pseudo-tested only if all of them survive**.



Survived = Bad News, Killed = Good News



SURVIVED

The mutant lived. Tests still passed.
No one checked the result.

→ **There is a real gap in the tests!**



KILLED

A test failed on the mutant.
The behavior is actually checked.

→ **This suggests the tests are good!**

*Counterintuitive insight: a **surviving mutant** is the **bad news** because the test suite could not detect this defect if it later exists in the program!*



Some Python Libraries Are Half C/C++

Examples: **NumPy** · **Pillow** · **pandas** · **PyTorch**

- Many projects that are critical to **data science** and **machine learning** use **C / C++**
- For these projects, the **performance-critical hot loops** are written in **C / C++**
- **NumPy** package averages **1.03 billion** to **1.08 billion** downloads per month

Python layer

exercised by your tests ✓

C / C++ layer

tested?



One import, two worlds. What if your test suite lives almost entirely in the green one?

Is the deep or hidden layer of a Python library well tested? Can you trust the quality of the library? Is there an **automated technique** for checking quality of testing at C/C++ layer?

Coverage & Mutation Tools Stop at C/C++

-  `coverage.py`: tracks coverage of Python code, can't see inside the C/C++!
-  `mutmut` / `cosmic-ray`: mutation testing, but it never touches `.c` / `.cpp` files!
-  `Descartes` / `PseudoSweep`: don't run on Python at all, only Java projects!

The native code in a Python project is a **blind spot** for the test suite! It may be the hardest-to-review, most performance-critical code you ship. **But, is it well tested?**

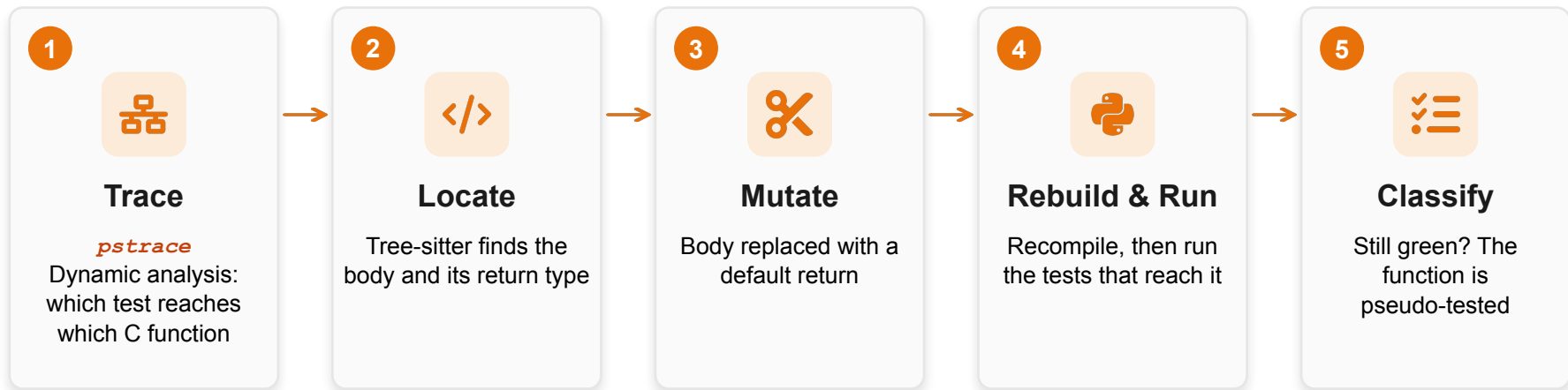
 So we built **PseudoScope** to look inside it and assess the tests!



How PseudoScope Works

Input: the library's C source and its own test suite

Output: functions that are pseudo-tested



repeat for every C function in the file



Why rebuild? Python never runs your C code directly. It runs a compiled version, so a mutant that is not recompiled changes nothing, every test passes, and every function looks pseudo-tested.



PseudoScope is Free and Open Source



Scan the QR code for our tool!
Use PseudoScope to find testing
limitations in the native code of
your Python project!

Direct reference to project GitHub:
[https://github.com/IsseiHasegawa/
PseudoScope](https://github.com/IsseiHasegawa/PseudoScope)



Running PseudoScope on ultrajson

- Illustrate how to apply PseudoScope to **ultrajson's** file called **objToJSON.c** (i.e., its object-to-JSON encoder), judged against **ultrajson's** own pytest suite.
- **ujson** is a fast JSON library designed as a drop-in replacement for Python's built-in **json** module. For efficiency reasons, its core processing is implemented in C/C++ (99.5%).

```
python -m pseudoclang \
--file src/ujson/python/objToJSON.c \
--project-root-source-dir ultrajson \
--test-command "source .venv/bin/activate && pip install -e . -q && python -m pytest -q" \
--test-runner-template "source .venv/bin/activate && pip install -e . -q && python -m pytest -q {selection}" \
--test-list-cmd "python -m pytest --collect-only -q | grep '::'" \
--pstrace-module ujson \
--pstrace-src-root src/ujson \
--pstrace-build-cmd "pip install -e ." \
--pstrace-test-cmd "python -m pytest tests/" \
--pstrace-python ultrajson/.venv/bin/python \
--output-file ultrajson-objToJSON-sweep.json
```



34 C Functions in One File of ultrajson

Discovered 34 function(s) in src/ujson/python/objToJSON.c:

- PyLongToINT64 (line 83)
- PyLongToUINT64 (line 89)
- PyLongToINTSTR (line 95)
- PyFloatToDOUBLE (line 102)
- PyStringToUTF8 (line 109)
- PyUnicodeToUTF8Raw (line 116)
- PyUnicodeToUTF8 (line 147)
- PyRawJSONToUTF8 (line 152)
- Tuple_iterNext (line 165)
- Tuple_iterEnd (line 177)
- Tuple_iterGetValue (line 181)
- List_iterNext (line 186)
- List_iterEnd (line 199)
- List_iterGetValue (line 203)
- Dict_convertKey (line 214)
- Dict_iterNext (line 244)
- Dict_iterEnd (line 262)

What you are looking at

Every C function in the encoder, with its name and line number. PseudoScope finds them by reading the source, before anything runs.

What happens to each one

The body is replaced with a default return, the extension is rebuilt, and only the tests that reach that function run again. Some return types get two default values, so functions produce two mutants.

Why it matters

This is one file in one library, and **not one of these functions is visible to coverage.py!**

Run recursively on all C/C++ files.



Mutate, Test, and Repeat

For each function: replace the body, rebuild, re-run the tests.

```
[8/34] Function: PyRawJSOToUTF8 (line 152)
Mutations: 1 (PASS: 0, FAIL: 1)
```

```
[9/34] Function: Tuple_iterNext (line 165)
Mutations: 2 (PASS: 0, FAIL: 2)
```

```
[10/34] Function: Tuple_iterEnd (line 177)
Mutations: 1 (PASS: 1, FAIL: 0)
```

```
[11/34] Function: Tuple_iterGetValue (line 181)
Mutations: 1 (PASS: 0, FAIL: 1)
```

```
[12/34] Function: List_iterNext (line 186)
Mutations: 2 (PASS: 0, FAIL: 2)
```

PASS: the mutant survived, which is the bad case. Your test suites **did not** notice the function was broken.



NOT WELL TESTED

FAIL: a test caught it, which is the good case. Your test suites **notice** the function was broken.



WELL TESTED

*Most mutants fail fast. A few, like **Tuple_iterEnd**, show PASS!*



Investigate PseudoScope's Output

FUNCTION	MUTANT	TEST RESULT
PyLongToINT64	return NULL;	FAIL (detected)
Tuple_iterNext	return 0; return 1;	FAIL (detected)
Object_getDoubleValue	return 0.0; return 1.0;	FAIL (detected)
Object_iterGetValue	return (JSOBJ){0};	FAIL (detected)
Tuple_iterEnd	(empty body)	PASS (survived)
Object_releaseObject	(empty body)	PASS (survived)

34 discovered **34** analyzed **3** survived **8.8%** pass rate

The **91.2%** detection rate suggests strong tests, while the **three survivors** provide a focused set of cases for further investigation. Now we can improve the tests!



What to Take Home About PseudoScope

“Would my tests notice if this function returned garbage?”

A passing test tells you the code **ran**. It does not tell you your tests are **effective enough**. Moreover, native code is prevalent, critical, and often overlooked.



PseudoScope asks that question for the code you never see:
the C and C++ inside the Python libraries that you depend on every day!



Conclusion and Key Contributions

The C and C++ inside your Python libraries was one place where no automated test assessment tool could easily look!

PseudoScope makes it checkable.

THE PROBLEM

Green tests and 100% coverage prove your code ran, not that results were checked.

THE BLIND SPOT

Certain Python popular and frequently adopted libraries are half C and C++, and coverage and mutation tools for Python normally **stop at that boundary**.

THE TOOL

PseudoScope carries extreme mutation testing across it: trace, mutate, rebuild, classify.

*Pseudo-tested detection existed only for Java. **Now it reaches the native code inside Python!***



Limitations and Future Work

PseudoScope effectively demonstrates the concept. We have a roadmap for what we will build next!

THE LIMIT	WHY IT EXISTS	HOW WE FIX IT
Dynamic only, no static mode	A survivor only counts if the tests ran, so the verdict has to come from a real run.	To improve performance, add a static pre-pass that skips unreachable and already-empty functions before any build.
macOS and Linux only	The tracer needs -finstrument-functions and ELF or Mach-O symbols. MSVC has no direct equivalent.	Target Windows through clang-cl and map symbols from PDB files.
Slow on large projects	Every mutant needs a rebuild and a test run, and libraries like numpy have very large test suites, making costs high.	Explore meta-mutants , cache incremental builds, and mutate in parallel , on top of the per-function test selection.

*I welcome your feedback and invite you to try the tool. **Please chat with me at PyOhio and on Discord!***

Thank You!

Project GitHub:

<https://github.com/IsseiHasegawa/PseudoScope>

Issei Hasegawa

Gregory M. Kapfhammer

