Monitoring the Execution of 14K Tests: Methods Tend to Have One Path That Is Significantly More Executed

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

ABSTRACT

The literature has provided evidence that developers are likely to test some behaviors of the program and avoid other ones. Despite this observation, we still lack empirical evidence from real-world systems. In this paper, we propose to automatically identify the tested paths of a method as a way to detect the method's behaviors. Then, we provide an empirical study to assess the tested paths quantitatively. We monitor the execution of 14,177 tests from 25 real-world Python systems and assess 11,425 tested paths from 2,357 methods. Overall, our empirical study shows that one tested path is prevalent and receives most of the calls, while others are significantly less executed. We find that the most frequently executed tested path of a method has 4x more calls than the second one. Based on these findings, we discuss practical implications for practitioners and researchers and future research directions.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Runtime environments**.

KEYWORDS

software testing, dynamic analysis, runtime monitoring, python

ACM Reference Format:
Andre Hora. 2024. Monitoring the Execution of 14K Tests: Methods Tend to Have One Path That Is Significantly More Executed. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663792

1 INTRODUCTION

Ideally, developers should test multiple behaviors (*e.g.,* expected and unexpected ones) of the program to catch more bugs and protect against regressions [1, 2, 16, 17, 20, 23, 26]. In practice, it is well-known that developers are more likely to test some behaviors than others [2–4, 6, 7, 10, 18, 21, 24, 25]. This happens because some behaviors of the program are often simpler to test. Another factor is that developers may lack test expertise, naively focusing on only testing the "happy cases" (aka, confirmation bias) [7].

Although the literature has provided evidence that some behaviors (*e.g.,* the expected ones) are more likely to be tested than others (*e.g.,* the unexpected ones), existing research is mostly restricted to controlled experiments, like case studies with students and developers [2, 6, 7, 24, 25]. For example, in an experiment conducted with students, they were found to naively test the "happy cases" [7]. In an experiment with developers, Teasley *et al.* [25] found evidence of using a positive test strategy, which was partially mitigated by increasing the expertise of the developers. While those results are insightful and bring to light relevant findings, they are not supported by real-world settings. We still lack empirical evidence extracted from real-world software systems and their test suites.

To illustrate a real-world scenario, consider the `email`[1] library provided by the Python Standard Library. Figure 1 presents method `email.message.Message.get`,[2] which returns the value of a named header field. Notice this method has `for` and `if` blocks, which may lead to three behaviors at runtime: (i) entering in both the `for` and `if` blocks, (ii) entering in the `for` block and not in the `if` block, and (iii) not entering in the `for` block. At this point, it is unclear what behaviors are the most and least frequently tested by developers.

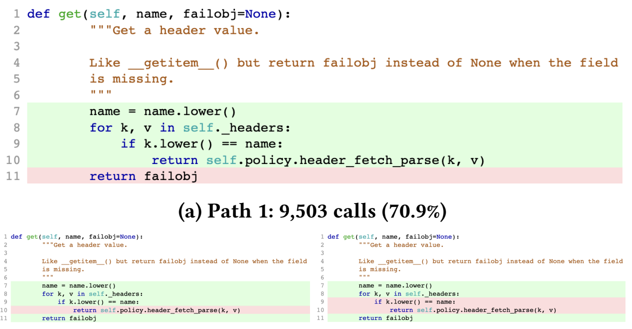
```python
def get(self, name, failobj=None):
    """Get a header value.

    Like __getitem__() but return failobj instead of None when the field
    is missing.
    """
    name = name.lower()
    for k, v in self._headers:
        if k.lower() == name:
            return self.policy.header_fetch_parse(k, v)
    return failobj
```

Figure 1: Method `email.message.Message.get` provided by the Python Standard Library.

After monitoring the test suite of the email library,[3] we find that this method is executed 13,396 times. These executions lead to three tested paths, as detailed in Figure 2. What is interesting is the large discrepancy between the execution frequency of different paths. Path 1 concentrates most of the calls (70.9%), Path 2 receives 24.7% of the calls, and Path 3 receives only 4.4%. This brings to light one important question: are tested paths of real-world software systems likely to concentrate calls (as in the presented example) or do calls tend to be more distributed among the tested paths?

[1] https://docs.python.org/3/library/email.html
[2] https://github.com/python/cpython/blob/5cd9c6b1fca549741828288febf9d5c13293847d/Lib/email/message.py#L494-L504
[3] https://github.com/python/cpython/tree/5cd9c6b1fca549741828288febf9d5c13293847d/Lib/test/test_email



```python
def get(self, name, failobj=None):
        """Get a header value.

        Like __getitem__() but return failobj instead of None when the field
        is missing.
        """
        name = name.lower()
        for k, v in self._headers:
            if k.lower() == name:
                return self.policy.header_fetch_parse(k, v)
        return failobj
```

(a) Path 1: 9,503 calls (70.9%)

```python
def get(self, name, failobj=None):
        """Get a header value.

        Like __getitem__() but return failobj instead of None when the field
        is missing.
        """
        name = name.lower()
        for k, v in self._headers:
            if k.lower() == name:
                return self.policy.header_fetch_parse(k, v)
        return failobj
```

(b) Path 2: 3,305 calls (24.7%)

```python
def get(self, name, failobj=None):
        """Get a header value.

        Like __getitem__() but return failobj instead of None when the field
        is missing.
        """
        name = name.lower()
        for k, v in self._headers:
            if k.lower() == name:
                return self.policy.header_fetch_parse(k, v)
        return failobj
```

(c) Path 3: 588 calls (4.4%)

Figure 2: Tested paths of `email.message.Message.get` (green: executed lines; red: not executed lines).

To the best of our knowledge, no study aims to gauge the execution frequency of different tested paths of a method. This information could provide insights for developers to improve existing test suites and support the creation of novel testing tools to better understand test suites. Moreover, it may reveal novel empirical data for researchers to quantify the difference between the execution frequency of distinct paths in real-world software systems.

In this paper, we propose an empirical study to assess the tested paths quantitatively. A tested path represents a set of input values that will make the method behave in the same way, that is, execute the same lines of code. We monitor the execution of 14,177 tests from 25 real-world Python systems, assessing 11,425 tested paths from 2,357 methods. We seek to answer two research questions:

- *RQ1: What is the frequency of the most tested paths?* We find that in methods with multiple tested paths, one tested path tends to receive most of the calls. Overall, the most tested path of a method has 4x more calls than the second one.
- *RQ2: What is the frequency of the least tested paths?* We find that the most tested path of a method has 6.5x more calls than the top 3+.

Overall, our empirical study shows that one tested path (*i.e.,* one method behavior) is prevalent and receives most of the calls, while others are significantly less executed. Based on our findings, we discuss practical implications for practitioners and researchers.

Contributions. The contributions of this paper are twofold. First, we provide an empirical study to quantitatively assess the tested paths of a method (Sections 4). Second, we propose practical implications for practitioners and researchers (Section 5).

2 DETECTING TESTED PATHS

This section describes a technique to identify and rank the tested paths of a method. First, we execute an instrumented version of the tests, collecting the executed lines of code at method-level (Section 2.1). Next, we detect the tested paths (Section 2.2) and compute the ranking of tested paths (Section 2.3). Finally, Section 2.4 briefly presents the tool that implements the proposed technique.

2.1 Collecting Executed Lines of Code

First, we need to run the test suite and collect relevant information about the system execution. For this purpose, we execute an instrumented version of the test suite that monitors the tests and collect data from the execution trace. Specifically, for each call of method m, we collect its *executed lines of code.* After running all tests, each executed method m has a set of executed lines of code.

2.2 Detecting the Tested Paths

A *tested path* represents a set of input values that will make the method behave in the same way, that is, execute the same lines of code. We define a *tested path* of an executed method m as a sorted set of executed lines of code of m. For example, method `email.message.Message.get` presented in Figure 2 has three tested paths.

For each executed method m, we compute its tested paths based on its set of executed lines of code. A tested path tp_m of method m is formed by a triple: (1) unique lines of code, (2) path frequency, and (3) path ratio. The *unique lines of code* is computed as follows: for each call of m, we collect the executed lines of code as a sorted set. We use a sorted set to avoid line duplication due to the execution of loops, thus, lines that are executed multiple times (due to loops) are counted once. The rationale is that a loop that iterates on the same lines one or multiple times are equivalent; without this constraint, a method with a loop could potentially mislead to multiple yet equivalent paths. The *path frequency* and *path ratio* are simply the absolute and relative execution frequencies of the tested path. As a result, we have a set of methods, each with at least one tested path.

2.3 Ranking the Tested Paths

Lastly, for each method m with one or more tested paths tp_m, we sort their paths in descending order of path frequency, creating a *ranking of tested paths.* Thus, the most executed paths are top-ranked, while the least executed ones are bottom-ranked.

2.4 Implementation Notes

We developed PathSpotter [14], a tool implements the proposed technique to detect and rank tested paths.[4] To extract the tested paths, we need to run an instrumented version of the test suite, monitoring the execution and collecting call information. PathSpotter relies on SpotFlow [15], a tool that perform runtime analysis in Python to collect the target information, such as executed lines of code at method level.[5] In short, this tool is implemented with the support of the standard system's trace function `sys.settrace` [22], which is the basis for performing runtime analysis in Python [12, 13].

3 STUDY DESIGN

3.1 Selecting Software Systems

We aim to study relevant and real-world software systems. Thus, we select Python systems that are largely adopted. Table 1 presents the 25 selected systems, which have 5,371 executed methods and 14,177 test methods. To have a larger diversity of projects, we select two types of systems: (1) real-world systems and (2) Python libraries.

[4] https://github.com/andrehora/pathspotter
[5] https://github.com/andrehora/spotflow

Table 1: Selected systems.

Name	Short Description	Methods	Tests
DateUtil	date and time library	241	2,029
TheFuck	console error corrector	550	1,887
Pylint	static code analyzer	1,537	1,822
Rich	rich text library	589	758
Requests	HTTP library	174	578
Flask	web framework	284	478
Cookiecutter	template handler	66	322
Six	compatibility library	32	199
BentoML	machine learning platform	319	169
Jupyter Client	Jupyter protocol client APIs	138	87
argparse	command-line interfaces	126	1,685
email	email message manager	381	1,666
tarfile	tar reading and writing	89	496
pathlib	OO filesystem paths	97	449
configparser	configuration file parser	82	341
os	operating system interfaces	41	316
logging	logging facility	215	208
csv	CSV reading and writing	15	113
collections	datatype container	112	111
imaplib	IMAP4 protocol client	47	103
ftplib	FTP protocol client	51	94
smtplib	SMTP protocol client	43	82
calendar	calendar helpers	63	72
gzip	gzip reading and writing	32	61
difflib	diff library	47	51
Total		5,372	14,177

Real-world systems. The 10 selected systems presented at the top of Table 1 have thousands of GitHub stars, millions of clients, and, in some cases, billions of downloads, which highlights their relevance. *Python libraries.* We also analyze the 15 libraries presented at the bottom of Table 1, which belong to the Python Standard Library and are hosted on the CPython repository. These libraries are fundamental to building virtually every Python application.

3.2 Selecting Methods

Table 1 shows that the systems have 5,372 executed methods. Among those methods, we find that 175 are generators, *i.e.,* methods that contain `yield` expressions. As this feature is absent in most programming languages, we filtered out generators. Moreover, we are interested in methods with multiple tested paths. We find 2,840 methods with only one tested path, which are also filtered out.

Finally, we selected 2,357 methods (*i.e.,* 5,372 − 175 − 2,840). The 2,357 selected methods have a total of 11,425 tested paths and are executed 13.3 million times by the tests. The median of tested paths per method is 3 (the first quartile is 2 and the third quartile is 5), while the median of calls per method is 107 (the first quartile is 16 and the third quartile is 956). Our dataset is publicly available at: https://github.com/andrehora/tested_paths_dataset.

3.3 Research Questions

3.3.1 RQ1: Frequency of the most tested paths. We explore the tested paths that are most executed. We compare the path ratio distribution of the top 1 and the top 2 tested paths in three configurations: (i) all selected methods, (ii) methods with two or three tested paths, and (iii) methods with four or more tested paths.

3.3.2 RQ2: Frequency of the least tested paths. While RQ1 focuses on the top-ranked tested paths, RQ2 focuses on the bottom-ranked ones. We compare the tested paths that are most executed (*i.e.,* the top 1) against the ones with fewer calls (*i.e.,* the top 3+). The top 3+ tested paths represent the top 3 and the other paths combined.

3.3.3 Rationales for RQ1 and RQ2. Prior studies have presented evidence that one behavior is likely to be more tested than the others [2, 6, 7, 24, 25]. However, we lack quantitative evidence from real-world software systems. We aim to fill this gap in the literature.

4 STUDY RESULTS

4.1 RQ1: Frequency of the most tested paths

Overall results: Table 2 presents the path ratio median for the top 1 and the top 2 tested paths in the selected methods. We notice that the top 1 tested paths are fairly more called than the top 2, independently of the number of paths. The difference between all distributions is statistically significant in all comparisons for the Mann-Whitney test (MWT), with a large effect for the Cohen test.[6]

On the median, the top 1 tested paths receive 72% of the calls (the first quartile is 53% and the third quartile is 92%), while the top 2 tested paths receive only 17% of the calls (the first quartile is 6% and the third quartile is 31%).

Table 2: Top 1 vs. top 2 path ratios.

Number of paths	Path ratio (median) Top 1 (%)	Path ratio (median) Top 2 (%)	MWT p-value	Cohen effect size
2	83	17	<0.01	large
3	73	18	<0.01	large
4	66	20	<0.01	large
5	67	17	<0.01	large
6	65	18	<0.01	large
7	60	18	<0.01	large
8	64	15.5	<0.01*	large
9	53	18	<0.01*	large
10+	53	17	<0.01	large
all (≥ 2)	72	17	<0.01	large

> **Finding 1**: Overall, one tested path tends to receive most of the calls. The most tested path of a method has 4x more calls than the second one.

Methods with two or three tested paths: Next, we focus only on the methods with two or three tested paths. In methods with two tested paths , the top 1 tested paths receive 83% of the calls, while the top 2 tested paths receive 17%, on the median. This represents close to 5x more calls in the top 1 than in the top 2. Similarly, in methods with three tested paths, the top 1 is also dominant (73% of the calls), the top 2 is less frequently called (18%), and the top 3 is rarely called (4%).

> **Finding 2**: In methods with two tested paths, one path tends receive close to 5x more calls than the second one. In methods with three tested paths, one path receives most of the calls (73%), while the third path is rarely called (4%).

Methods with four or more tested paths: Finally, we analyzed methods with four or more tested paths. Table 2 shows that the

[6]In methods with 8 and 9 tested paths, the Mann-Whitney test cannot be computed with confidence due to the data size in both RQ1 and RQ2.

path ratio difference is also large for the methods with four or more tested paths. For example, in methods with four tested paths, the top 1 tested paths receive 66% of the calls, while the top 2 tested paths have 20%, on the median. Interestingly, even in highly complex methods with 10 or more tested paths, one path tends to be dominant. In this case, the top 1 tested paths have 53% of the calls, while the top 2 tested paths have 17%.

Finding 3: Even methods with four or more tested paths have one path that receives the majority of the calls. In this case, the top 1 ranges from 53% to 67%, while the top 2 is between 15.5% and 20%.

4.2 RQ2: Frequency of the least tested paths

Table 3 details the path ratio median for the top 1 tested paths and the top 3+ tested paths (*i.e.,* tested paths with fewer calls). Overall, the top 1 tested paths are largely more called than the top 3+, independently of the number of paths. The difference between both distributions is statistically significant for the Mann-Whitney test (MWT), with a large effect for the Cohen effect size.

Table 3: Top 1 vs. top 3+ path ratios.

Number of paths	Path ratio (median) Top 1 (%)	Path ratio (median) Top 3+ (%)	MWT p-value	Cohen effect size
3	73	4	<0.01	large
4	66	7	<0.01	large
5	67	11	<0.01	large
6	65	12.5	<0.01	large
7	60	19	<0.01	large
8	64	15.5	<0.01*	large
9	53	19	<0.01*	large
10+	53	24	<0.01	large
all ($\geq$ 3)	65	10	<0.01	large

Notice that when incrementing the number of tested paths (from 3 to 10+), the top 3+ path ratio tends to increase (from 4% to 24%), while the top 1 path ratio tends to decrease (from 73% to 53%). This happens because the more tested paths a method has, the more it needs to share the calls with the other paths. Nevertheless, the prevalence of the top 1 is proportionally higher. Consider the methods with 10+ tested paths: the top 1 receives 53% of the calls, while all the other tested paths combined receive only 24%.

Finding 4: The top 3+ tested paths receive a minority of the calls, ranging from 4% to 24%. Overall, the most tested path of a method has 6.5x more calls than the top 3+.

5 DISCUSSION AND IMPLICATIONS

Novel evidence from real-world systems that some paths in a method are executed more frequently than others. To the best of our knowledge, this study is the first to present evidence from real-world systems that one tested path is prevalent and receives most of the calls, while other paths are less frequently called. The most called path of a method has 4x more calls than the second most called (RQ1) and 6.5x more calls than the top 3+ (RQ2). This disparity happens for all analyzed projects, indicating that the developers across multiple large projects have made this testing choice. Further analysis is needed to better reason about this disparity. Thus, our study complements current findings [2–4, 6, 7, 10, 18, 21, 24, 25] by analyzing real-world projects and confirming that one method behavior tends to receive most of the calls from test suites.

Novel tools to support the comprehension of test suites. The discrepancy between the execution frequency of different paths may provide the basis for the development of novel testing tools. New tools can explore the execution frequency of the tested paths to support comprehension of test suites, *e.g.,* presenting the paths of the SUT that are more/less executed with heatmap-like visualizations. These tools can be used by testers to potentially improve test suites, for example, tested paths that are rarely executed are candidates to be analyzed and possibly further tested.

Exploring the characteristics of frequently executed paths to improve automated test generation. Modern test generation tools [8, 9, 19] that are guided by coverage commonly try to cover each line/branch/path only once. Therefore, learning the characteristics of frequently executed paths and changing automated test generators such that they generate multiple test cases for such paths can be a potential research direction to improve current test generation tools.

6 THREATS TO VALIDITY

Deselected tests. While monitoring the test suites, we deselected 149 test methods that failed due to the instrumentation. However, those cases were rare, representing less than 1% of the analyzed tests.

Generalization of the results. We assessed real-world Python systems, which are among the most popular in the Python ecosystem. Despite these observations, our findings – as usual in empirical software engineering – may not be directly generalized to other projects or other programming languages.

7 RELATED WORK

The literature shows that developers are more likely to test some behaviors of the program than others [2–4, 6, 7, 10, 18, 21, 24, 25]. For example, students were found to naively test the "happy cases", writing basic test cases covering the expected behavior [7], while experienced developers are more prone to test unexpected/unhappy cases [2, 18, 21, 25]. The "happy path testing" is also considered a test smell that should be avoided [3, 5, 10, 11]. Our study complements the literature by analyzing real-world projects and confirming that one method path tends to receive most of the calls from test suites.

8 CONCLUSION

In this paper, we presented an empirical study to assess the tested paths quantitatively. We monitored the execution of 14,177 tests from 25 real-world Python systems and assessed 11,425 tested paths from 2,357 methods. Overall, we found that one tested path is prevalent and receives most of the calls, while others are significantly less executed. As future work, we plan to extend our empirical study to understand better other characteristics of the tested paths, like the diversity of inputs, outputs, and exceptions.

ACKNOWLEDGMENT

This research is supported by CAPES, CNPq, and FAPEMIG.

REFERENCES

[1] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide*. Simon and Schuster.

[2] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2021. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4925–4946.

[3] Gina R Bai, Justin Smith, and Kathryn T Stolee. 2021. How students unit test: Perceptions, practices, and pitfalls. In *ACM Conference on Innovation and Technology in Computer Science Education*. 248–254.

[4] Lex Bijlsma, Niels Doorn, Harrie Passier, Harold Pootjes, and Sylvia Stuurman. 2021. How do students test software units?. In *International Conference on Software Engineering: Software Engineering Education and Training*. IEEE, 189–198.

[5] David Bowes, Tracy Hall, Jean Petric, Thomas Shippey, and Burak Turhan. 2017. How good are my tests?. In *Workshop on Emerging Trends in Software Metrics*. IEEE, 9–14.

[6] Adnan Causevic, Rakesh Shukla, Sasikumar Punnekkat, and Daniel Sundmark. 2013. Effects of negative testing on TDD: An industrial experiment. In *International Conference on Agile Processes in Software Engineering and Extreme Programming*. Springer, 91–105.

[7] Stephen H Edwards and Zalia Shams. 2014. Do student programmers all tend to write the same software tests?. In *Conference on Innovation & technology in Computer Science Education*. 171–176.

[8] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*. 416–419.

[9] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology* 24, 2 (2014), 1–42.

[10] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software* 138 (2018), 52–81.

[11] Vahid Garousi, Baris Kucuk, and Michael Felderer. 2018. What we know about smells in software test code. *IEEE Software* 36, 3 (2018), 61–73.

[12] Andre Hora. 2021. What Code Is Deliberately Excluded from Test Coverage and Why?. In *International Conference on Mining Software Repositories*. 392–402.

[13] Andre Hora. 2023. Excluding code from test coverage: practices, motivations, and impact. *Empirical Software Engineering* 28, 1 (2023), 1–33.

[14] Andre Hora. 2024. PathSpotter: Exploring Tested Paths to Discover Missing Tests. In *International Conference on the Foundations of Software Engineering*. 1–5.

[15] Andre Hora. 2024. SpotFlow: Tracking Method Calls and States at Runtime. In *International Conference on Software Engineering*. 1–5.

[16] Cem Kaner, Sowmya Padmanabhan, and Douglas Hoffman. 2013. *The Domain Testing Workbook*. Context Driven Press.

[17] Vladimir Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster.

[18] Laura Marie Leventhal, Barbee M Teasley, Diane S Rohlman, and Keith Instone. 1993. Positive test bias in software testing among professionals: A review. In *International Conference on Human-Computer Interaction*. Springer, 210–218.

[19] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *International Conference on Software Engineering: Companion Proceedings*. 168–172.

[20] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[21] Rahul Mohanani, Iflaah Salman, Burak Turhan, Pilar Rodríguez, and Paul Ralph. 2018. Cognitive biases in software engineering: a systematic mapping study. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1318–1339.

[22] Python sys.settrace. November, 2023. https://docs.python.org/3/library/sys.html#sys.settrace.

[23] Stuart C Reid. 1997. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *International Software Metrics Symposium*. IEEE, 64–73.

[24] Iflaah Salman, Burak Turhan, and Sira Vegas. 2019. A controlled experiment on time pressure and confirmation bias in functional software testing. *Empirical Software Engineering* 24, 4 (2019), 1727–1761.

[25] Barbee E Teasley, Laura Marie Leventhal, Clifford R Mynatt, and Diane S Rohlman. 1994. Why software testing is sometimes ineffective: Two applied studies of positive test strategy. *Journal of Applied Psychology* 79, 1 (1994), 142.

[26] Titus Winters, Hyrum Wright, and Tom Manshreck. 2020. Software Engineering at Google: Lessons Learned from Programming over Time.

Received 28-JAN-2024; accepted 2024-04-09