TestDossier: A Dataset of Tested Values Automatically Extracted from Test Execution

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

ABSTRACT

Real-world test suites are often complex and may have thousands of test cases. In this scenario, it is not easy to spot what values are actually covered by the tests. Having access to every tested value of a test suite would provide the basis to (1) assess the quality of tested data and (2) have actionable information to improve them. In this paper, we propose TestDossier, a dataset of tested values automatically extracted from the execution of Python tests. To collect runtime data, we run an instrumented version of the tests, monitoring the test execution, and extracting argument and variable values. We monitored the test suites of 15 Python Standard Libraries to create the dataset. TestDossier contains 1,234 distinct argument/variable names and 133,169 distinct values, leading to a total of 12,9M individual values. We envision that our dataset can help developers detect rarely tested values, untested values, and variations of tested values. We also foresee that our dataset can support novel empirical studies in the context of software testing, for example, it can expose the diversity of the tested data.

CCS CONCEPTS

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

KEYWORDS

software testing, test quality, python, unittest, pytest

ACM Reference Format:
Andre Hora. 2024. TestDossier: A Dataset of Tested Values Automatically Extracted from Test Execution. In *21st International Conference on Mining Software Repositories (MSR '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643991.3644875

1 INTRODUCTION

Ideally, test suites should test both expected and unexpected behaviors to catch more bugs, protect against regressions, and ensure sustainable software evolution [1, 4, 12, 18, 19, 28]. In practice, real-world test suites are often complex and may have thousands of test cases. In this scenario, it is not easy to spot what values are actually covered by the tests. Therefore, we cannot easily detect whether expected and unexpected behaviors are properly tested [2, 3, 5, 8, 14, 22, 23, 25].

Consider a large test suite that tests multiple encoding formats (*e.g., ascii*, *utf-8*, *iso8859-1*, etc.). In this context, it is important to have an overview of *every* tested encoding format. One solution would be to inspect each test case and keep track of the tested encoding formats. This solution works for small test suites, however, it is not feasible for large ones. That is, developers cannot manually keep track of every relevant value that is tested both directly and indirectly by the test suite. Having access to the tested values of a test suite would provide the basis to (1) assess the quality of tested data and (2) have actionable information to improve them, for example, creating novel tests to cover untested values.

In this paper, we propose TestDossier, a dataset of tested values automatically extracted from the execution of Python tests. Our dataset construction has two major steps: collecting runtime data and computing tested values (Section 2). To collect runtime data, we run an instrumented version of the tests, monitoring the test execution, and extracting argument and variable values. Our dataset is exported in JSON format, in which a key is an argument/variable name and the key's value is a list with every argument/variable value during the test execution. To create the dataset, we monitored the test suites of 15 Python Standard Libraries. TestDossier contains 1,234 distinct argument/variable names and 133,169 distinct values, leading to a total of 12,9M individual values (Section 3).

We conclude by discussing multiple applications for our dataset (Section 4). We envision that TestDossier can help developers and testers detect rarely tested values, untested values, and variations of tested values. We also foresee that our dataset can support the improvement of fake data generators and novel empirical studies to expose the diversity of the tested data.

Dataset availability. Our dataset is publicly available at: https://doi.org/10.5281/zenodo.10292595.

2 DATASET CONSTRUCTION

Figure 1 presents an overview of the dataset construction, which has two major steps: (1) collecting runtime data and (2) computing tested values. Next, we detail each step.

2.1 Collecting Runtime Data

To extract the tested values of a test suite, we need to run an instrumented version of the test suite. The first step consists of running and monitoring the tests and collecting method/function call information at runtime. For this purpose, we rely on SpotFlow [17], a tool to ease runtime analysis in Python.[1] It executes and monitors a target Python program, collecting detailed information on calls and

[1] https://github.com/andrehora/spotflow

MSR '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0587-8/24/04…$15.00
https://doi.org/10.1145/3643991.3644875

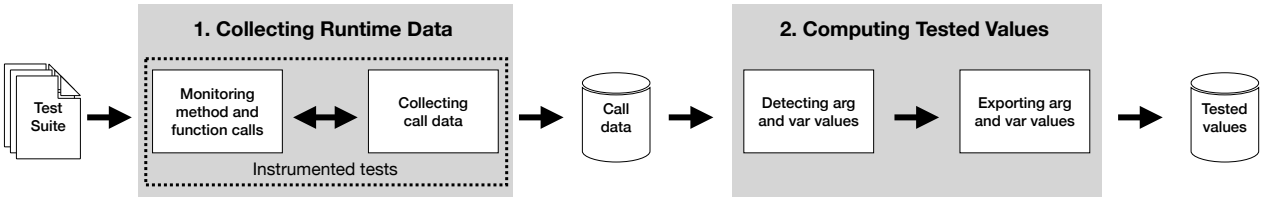

Figure 1: Overview of the dataset construction.

states. SpotFlow is implemented with the support of the standard system's trace function `sys.settrace` [26]. This function is the basis for performing runtime analysis in Python, for instance, it is used to build Coverage.py, the *de facto* coverage tool for Python [15, 16]. The trace function allows for registering a hook that gets called at every executed line of code, function call, function return, and exception. SpotFlow registers to the hook, monitors those events, and collects call data. This tool performs the inspection of live objects on the current stack frame[2] to collect the call and state objects. For this purpose, it relies on the *inspect*[3] module, which provides functions to help get information about live objects, such as modules, classes, methods, functions, and frame objects.

As output, this step exports information related to the execution of methods/functions by the test suite, named call data. The *call data* consists of multiple objects such as `MethodCall` and `CallState`, which store every call and state of a method/function with their respective argument and variable values. We store values of the basic build-in types: numeric (`int`, `float`, and `complex`), boolean (`bool`), and text sequence (`str`). We also store values of the types: sequence (`list`, `tuple`, and `range`), set (`frozenset` and `set`), and mapping (`dict`) that include only basic types. For other complex objects, we store their types (not their values) to avoid any possible issue caused by the inspection of live objects. Notice that knowing the type of complex objects is valuable because Python is dynamically typed, therefore, types are only known at runtime.

For example, consider the method `count_uppercase_words` presented in Listing 1 and the two test methods presented in Listing 2. After running and monitoring these tests, this step produces the *call data* summarized in Figure 2 (due to space limit, Figure 2 presents a simplified and didactic version of the generated data).

```
class StringParser:

    def count_uppercase_words(self, text):
        counter = 0
        for word in text.split():
            if word.isupper():
                counter += 1
        return counter
```

Listing 1: Target method (parser.py).

```
class TestStringParser(unittest.TestCase):

  def test_find_multiple_uppercase_words(self):
      p = StringParser()
      counter = p.count_uppercase_words("ABC DEF")
      self.assertEqual(counter, 2)

  def test_not_find_uppercase_word(self):
      p = StringParser()
      counter = p.count_uppercase_words("abc")
      self.assertEqual(counter, 0)
```

Listing 2: Test suite (test_parser.py).

```
MonitoredProgram:
- monitored_methods: ['count_uppercase_words']

MonitoredMethod:
- name: 'count_uppercase_words'
- class_name: 'StringParser'
- filename: 'parser.py'
- calls: [MethodCall 1, MethodCall 2]

MethodCall 1:
- caller: 'test_find_multiple_uppercase_words'
- call_state: CallState 1

CallState 1:
- arg_states: [ArgState(name='text', value='ABC DEF']
- var_states:
  - counter: VarStateHistory([0, 1, 2])
  - word: VarStateHistory(['ABC', 'DEF'])

MethodCall 2:
- caller: 'test_not_find_uppercase_word'
- call_state: CallState 2

CallState 2:
- arg_states: [ArgState(name='text', value='abc']
- var_states:
  - counter: VarStateHistory([0])
  - word: VarStateHistory(['abc'])
```

Figure 2: Example of call data (simplified version).

The `MonitoredProgram` object holds the monitored methods, which is in this case only method `count_uppercase_words`.

`MethodCall` represents a method call that happens at runtime. `CallState` holds the state of a method call, with information about argument values and local variable values. The argument of the first call is the string "ABC DEF". Note that the monitored method has two local variables: `counter` and `word`. The states of those variables over time are also recorded, for example, we can check that `counter` has the values 0, 1, and 2, while `word` has the values "ABC" and "DEF" due to the text split. In the second call, the argument is "abc". The local variable `counter` is always zero (as it is not incremented), while `word` is "abc" (as the text is not split).

[2] https://docs.python.org/3/reference/datamodel.html#frame-objects
[3] https://docs.python.org/3/library/inspect.html

2.2 Computing Tested Values

In the second step, we compute the *tested values* by processing the *call data*. Specifically, we mine values of arguments and local variables from the call data objects. We automatically inspect every `CallState` object and extract the argument values from the attribute `arg_states` and the variable values from the attribute `var_states` (as presented in Figure 2). In this process, we group argument and variable values by name.

As output, this step exports the *tested values* dataset in JSON format. In this dataset, a key is an argument/variable name and the key's value is a list with every argument/variable value during the test execution. Each argument/variable value is a tuple with (1) the value itself and (2) the frequency of the value. For example, Figure 3 presents the *tested values* computed from the *call data* presented in Figure 2. We notice three keys representing every argument/variable name: `text`, `counter`, and `word`. The key's value represents the argument/variable values, for example, `text` has two values: "ABC DEF" with frequency 1 and "abc" with frequency 1; `counter` has three values: 0 with frequency 2, 1 with frequency 1, and 2 with frequency 1; and `word` has three values: "ABC" with frequency 1, "DEF" with frequency 1, and "abc" with frequency 1.

```
{
    "text": [["ABC DEF", 1], ["abc", 1]],
    "counter": [[0, 2], [1, 1], [2, 1]],
    "word": [["ABC", 1], ["DEF", 1], ["abc", 1]]
}
```

Figure 3: Example of tested values.

3 DATASET DESCRIPTION

Our dataset has been curated from the test suites of 15 Python Standard Libraries: gzip, calendar, locale, json, ast, csv, ftplib, collections, os, tarfile, pathlib, smtplib, argparse, configparser, and email. These libraries are fundamental to building any Python application, providing features to handle text processing, file access, persistence, and networking, to name a few. Table 1 presents some statistics of the dataset. It contains 1,234 distinct argument/variable names and 133,169 distinct values, leading to a total of 12,9M individual values. On the average, the value frequency is 97.

Table 1: Dataset description.

Data	Value
Analyzed test suites	15
Argument/variable names	1,234
Argument/variable values: distinct	133,169
Argument/variable values: all	12,938,567
Value frequency: average	97
Value frequency: median	2
Value frequency: min	1
Value frequency: max	455,462
Dataset size	555MB

4 DATASET USAGE AND LIMITATIONS

Our dataset can be used to help developers and testers detect rarely tested values, untested values, and variations of tested values. We also foresee that our dataset can support the improvement of fake data generators and novel empirical studies to expose the diversity of the tested data

4.1 Usage 1: Rarely Tested Values

Our dataset can be used to detect rarely tested values. For example, Figure 4 presents an excerpt of TestDossier, detailing the tested values of `encoding`. We notice that popular encoding formats are largely tested by the test suites, such as *ascii*, *utf-8*, and *iso8859-1*. On the other hand, we can observe that less popular encoding formats are rarely tested, for example, *euc_kr* (Korean encoding), *tis620* (Thai encoding), and *iso8859-15* (a revision of *iso8859-1*). Developers and testers may rely on this data to possibly improve existing tests that cover these rarely tested values. Note that not necessarily rarely tested values must be covered by more tests. This data can be seen as an overview of the tested data and it is up to the expert to decide whether some tests should be improved or not.


```
"encoding": [["'ascii'",182133],["'utf-8'",114867],
["'iso8859-1'",90657],["None",5790],["''",551],["'q'",416],
["'us-ascii'",413],["'utf7'",118],["'b'",61],["'locale'",26],
["'utf8'",23],["'latin-1'",22],["'UTF-8'",14],["'utf_8'",13],
["'euc'",12],["'iso88591'",10],["'sjis'",9],["'iso88592'",7],
["'eucjp'",6],["'jis'",6],["'iso8859-9'",6],["'ISO8859-1'",6],
["'ISO8859-15'",5],["'big5'",5],["'JIS7'",5],["'utf-16'",4],
["'microsoftcp1251'",4],["'eucJP'",4],["'88591'",3],
["'885915'",3],["'euckr'",3],["'iso8859-15'",3],["'koi8c'",3],
["'koi8-c'",3],["'microsoft-cp1251'",3],["'iso8859-2'",3],
["'ajec'",3],["'iso-2022-jp'",3],["'iso2022jp'",3],["'jis7'",3],
["'ujis'",3],["'latin_1'",3],["'euc_jp'",3],["'shift_jis'",3],
["'SJIS'",3],["'ISO8859-9'",3],["'euctw'",2],["'iso88599e'",2],
["'cp1133'",2],["'ibmcp1133'",2],["'georgianacademy'",2],
["'mscode'",2],["'pck'",2],["'iso88595'",2],["'KOI8-C'",2],
["'CP1251'",2],["'ISO8859-2'",2],["'iso2022_jp'",2],
["'iso8859_9'",2],["'l1'",2],["'iso8859-9e'",1],
["'ibm-cp1133'",1],["'georgian-academy'",1],["'koic'",1],
["'en'",1],["'c'",1],["'437'",1],["'isiri3342'",1],
["'isciidev'",1],["'armscii8'",1],["'nunacom8'",1],
["'georgianps'",1],["'georgianrs'",1],["'mulelao1'",1],
["'cp1251'",1],["'tscii'",1],["'tscii0'",1],["'tactis'",1],
["'tis620'",1],["'tatarcyr'",1],["'tcvn'",1],["'tcvn5712'",1],
["'viscii'",1],["'viscii111'",1],["'big5hk'",1],["'gbk'",1],
["'euc_kr'",1],["'eucKR'",1],["'ISO8859_15'",1],
["'iso8859_15'",1],["'koi8_c'",1],["'microsoft_cp1251'",1],
["'iso8859_2'",1],["bytes",1]]
```


Figure 4: Tested values of `encoding`.

4.2 Usage 2: Variations of Tested Values

We envision that TestDossier can also detect variations of tested values. Table 2 summarizes some variations considering the tested values of `encoding`. We find that the encoding format *iso8859-15* is tested with five variations: *iso8859-15*, *iso8859_15*, *ISO8859-15*, *885915*, *ISO8859_15*. The encoding format *utf-8* has four tested variations: *utf-8*, *utf8*, *UTF-8*, and *utf_8*. Developers and testers can rely on this data to spot tiny tested value variations and use this information to enhance their test cases. For example, while the *iso8859-15* and *iso8859-9* have five variations, the *iso8859-2* and *iso8859-1* have only four. This may warn that some variations are missing in the latter cases.

Table 2: Examples of variations of tested values (`encoding`).

Tested Variations	#
iso8859-15, *iso8859_15*, *ISO8859-15*, *885915*, *ISO8859_15*	5
iso8859-9, *iso88599e*, *iso8859-9e*, *iso8859_9*, *ISO8859-9*	5
iso8859-2, *iso88592*, *iso8859_2*, *ISO8859-2*	4
iso8859-1, *iso88591*, *ISO8859-1*, *88591*	4
koi8-c, *KOI8-C*, *koi8c*, *koi8_c*	4
utf-8, *utf8*, *UTF-8*, *utf_8*	4
microsoftcp1251, *microsoft-cp1251*, *microsoft_cp1251*	3
iso2022jp, *iso2022_jp*, *iso-2022-jp*	3
eucJP, *euc_jp*, *eucjp*	3
euckr, *euc_kr*, *eucKR*	3
ibmcp1133, *ibm-cp1133*	2
latin-1, *latin_1*	2
sjis, *SJIS*	2

4.3 Usage 3: Untested Values

We also foresee that TestDossier can be used to detect untested values. Consider again the tested values of `encoding`. Table 3 details parts of the ISO8859 encoding family that are tested and untested. We find four tested parts: *iso8859-1*, *iso8859-2*, *iso8859-9*, and *iso8859-15*. Considering that the ISO8859 encoding family is divided into 16 parts,[4] we find that 12 parts (16 - 4) are untested, such as *iso8859-3* and *iso8859-16*. In this case, developers and testers can use this information to create novel tests covering the untested values.

Table 3: Examples of untested values (`encoding`).

Tested Values	*iso8859-1*, *iso8859-2*, *iso8859-9*, *iso8859-15*
Untested Values	*iso8859-3*, *iso8859-4*, *iso8859-5*, *iso8859-6*, *iso8859-7*, *iso8859-8*, *iso8859-10*, *iso8859-11*, *iso8859-12*, *iso8859-13*, *iso8859-14*, *iso8859-16*

4.4 Usage 4: Automated Input Generators

We envision that our dataset can be used to support the improvement of automated input generators. These tools typically generate fake data to support software testing in multiple contexts, like addresses, currencies, credit cards, and phone numbers, to name a few. For this purpose, automated input generators commonly mimic the real data format with random words. For example, consider the popular tool called Faker [10]. This tool can generate fake addresses for a certain country by specifying every possible address format, such as street names and addresses.[5] In this context, our dataset can be used as a source to discover novel data formats. Our dataset includes actual tested data about hundreds of distinct contexts, such as general contexts (*e.g.,* time, date, and filenames) and specific ones (*e.g.,* locale and language formats). For instance, we can see that a simple encoding may have dozens of possible formats, as presented in Figure 4. Those formats can be directly adopted to expand the ones provided by automated input generators.

[4]https://en.wikipedia.org/wiki/ISO/IEC_8859

[5]Example: https://github.com/joke2k/faker/blob/7c9ba46ad7ee5960f22eefd79c20266f6c9e90ca/faker/providers/address/fr_FR/__init__.py

4.5 Usage 5: Empirical Studies on Tested Data

In addition to exploring the frequency and data format of tested values, we also foresee that our dataset can support novel empirical studies in the context of software testing. For example, it can expose the diversity of the tested data. In this context, metrics can be proposed to assess the homogeneity and heterogeneity of tested data. Test suites with an over-concentration of homogeneous tested data could be seen as a potential problem.

4.6 Limitations

Our current dataset presents the tested values grouped by name. There may exist other solutions to group the values, for example, by type. This could be explored in future versions of the work. Moreover, we analyze 11 build-in types: `int`, `float`, `complex`, `bool`, `str`, `list`, `tuple`, `range`, `frozenset`, `set`, and `dict` (as detailed in Section 2.1). For other complex objects, we keep track of their types. As these objects are not necessarily serializable, we avoid any possible issue caused by the inspection of the live objects during the execution of the instrumented tests. We recall that having access to the type of complex objects is important because Python is dynamically typed, thus, types are only known at runtime. We plan to expand the monitored types in further versions of TestDossier.

5 RELATED WORK

Dynamic analysis is fundamental for multiple software engineering tasks, such as software testing [7, 11, 21]. Unfortunately, few tools have been created and made public to support developers extracting information from software execution. Rabiser *et al.* [24] found that most monitoring tools are not publicly available. In Python, we find SpotFlow [17] and DynaPyt [9]. DynaPyt is a dynamic analysis framework that offers hooks into specific kinds of runtime events [9]. In this study, we rely on SpotFlow [17] due to its extension facility to create the proposed dataset. Multiple datasets have been proposed in the context of software testing. For example, Methods2Test [27] is a large dataset of focal methods mapped to test cases extracted from Java projects. TestRoutes [20] is a test-to-code traceability dataset containing the traceability information on 220 test cases, also in Java. Jbench [13] is a dataset of data races for concurrency testing. Recently, a dataset with snapshot tests was proposed to address the lack of data on this type of test [6]. TestDossier contributes to the software testing literature with a novel dataset focused on exposing the runtime tested values.

6 CONCLUSION

This paper presented TestDossier, a dataset of tested values automatically extracted from the execution of Python tests. To create the dataset, we monitored the test suites of 15 Python Standard Libraries. TestDossier contains 1,234 distinct argument/variable names and 133,169 distinct values, leading to a total of 12,9M individual values. We envision that TestDossier can help developers and testers detect rarely tested values, untested values, and variations of tested values. We also foresee that our dataset can support the improvement of fake data generators and novel empirical studies to expose the diversity of the tested data.

ACKNOWLEDGMENT

This research is supported by CNPq, CAPES, 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] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.
[5] 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.
[6] Emily Bui and Henrique Rocha. 2023. Snapshot Testing Dataset. In *International Conference on Mining Software Repositories*. IEEE, 558–562.
[7] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.
[8] 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.
[9] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: a dynamic analysis framework for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 760–771.
[10] Faker: Faker is a Python package that generates fake data for you. November, 2023. https://github.com/joke2k/faker.
[11] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. 2021. A taxonomy for classifying runtime verification tools. *International Journal on Software Tools for Technology Transfer* 23, 2 (2021), 255–284.
[12] Michael Feathers. 2004. *Working Effectively with Legacy Code.* Prentice Hall Professional.
[13] Jian Gao, Xin Yang, Yu Jiang, Han Liu, Weiliang Ying, and Xian Zhang. 2018. Jbench: a dataset of data races for concurrency testing. In *International Conference on Mining Software Repositories*. 6–9.
[14] 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.
[15] Andre Hora. 2021. What Code Is Deliberately Excluded from Test Coverage and Why?. In *International Conference on Mining Software Repositories*. 392–402.
[16] Andre Hora. 2023. Excluding code from test coverage: practices, motivations, and impact. *Empirical Software Engineering* 28, 1 (2023), 1–33.
[17] Andre Hora. 2024. SpotFlow: Tracking Method Calls and States at Runtime. In *International Conference on Software Engineering*. 1–5.
[18] Cem Kaner, Sowmya Padmanabhan, and Douglas Hoffman. 2013. *The Domain Testing Workbook.* Context Driven Press.
[19] Vladimir Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns.* Simon and Schuster.
[20] András Kicsi, László Vidács, and Tibor Gyimóthy. 2020. Testroutes: A manually curated method level dataset for test-to-code traceability. In *International Conference on Mining Software Repositories*. 593–597.
[21] LearnableProgramming: Designing a programming system for understanding programs. January, 2022. http://worrydream.com/LearnableProgramming.
[22] 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.
[23] 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.
[24] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul Grünbacher. 2017. A comparison framework for runtime monitoring approaches. *Journal of Systems and Software* 125 (2017), 309–321.
[25] 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.
[26] sys.settrace. May, 2023. https://docs.python.org/3/library/sys.htmlsys.settrace.
[27] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. Methods2Test: A dataset of focal methods mapped to test cases. In *International Conference on Mining Software Repositories*. 299–303.
[28] Titus Winters, Hyrum Wright, and Tom Manshreck. 2020. Software Engineering at Google: Lessons Learned from Programming over Time.