An Empirical Evaluation of Property-Based Testing in Python

SAVITHA RAVI, University of California San Diego, USA
MICHAEL COBLENZ, University of California San Diego, USA

Property-based testing (PBT) is a testing methodology with origins in the functional programming community. In recent years, PBT libraries have been developed for non-functional languages, including Python. However, to date, there is little evidence regarding how effective property-based tests are at finding bugs, and whether some kinds of property-based tests might be more effective than others. To gather this evidence, we conducted a corpus study of 426 Python programs that use Hypothesis, Python's most popular library for PBT. We developed formal definitions for 12 categories of property-based test and implemented an intraprocedural static analysis that categorizes tests. Then, we evaluated the efficacy of test suites of 40 projects using mutation testing, and found that on average, each property-based test finds about 50 times as many mutations as the average unit test. We also identified the categories with the tests most effective at finding mutations, finding that tests that look for exceptions, that test inclusion in collections, and that check types are over 19 times more effective at finding mutations than other kinds of property-based tests. Finally, we conducted a parameter sweep study to assess the strength of property-based tests as a function of the number of random inputs generated, finding that 76% of mutations found were found within the first 20 inputs.

CCS Concepts: • **Software and its engineering** → **Correctness**; **Software notations and tools**.

Additional Key Words and Phrases: Property-based testing, empirical evaluation of testing, taxonomy of property-based tests, mutation testing

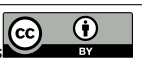
ACM Reference Format:
Savitha Ravi and Michael Coblenz. 2025. An Empirical Evaluation of Property-Based Testing in Python. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 412 (October 2025), 32 pages. https://doi.org/10.1145/3764068

1 Introduction

Property-based testing (PBT) [12] enables programmers to define classes of tests by specifying properties that should hold under given assumptions. For example, a test of the factorial function might say that if the input is a positive integer, the output is also a positive integer. Then, through random generation of test inputs, PBT allows developers to test hundreds of examples without manually selecting hundreds of inputs. Popularized in Haskell [8], property-based testing began in functional languages; other languages, such as Java [24] now have their own libraries for PBT. Python, currently the most popular programming language [43], has its own property-based testing library, Hypothesis [33]. Property-based testing is a particularly promising approach for applying techniques from the field of programming languages, since it enables structured exploration of relevant input spaces [31] and enables developers to express formal properties that are not expressible in their language's type system. It can also aid in formal verification efforts by helping developers avoid attempting to prove properties that can be shown to be false [4, 32, 39].

The goal of tests is to find bugs; if a thorough test suite fails to find bugs, the developer may be assured that at least the software handles important cases correctly. If we are to recommend that developers use property-based tests, it should be because they can find more bugs (or more

Authors' Contact Information: Savitha Ravi, University of California San Diego, La Jolla, California, USA, s2ravi@ucsd.edu; Michael Coblenz, University of California San Diego, La Jolla, California, USA, mcoblenz@ucsd.edu.



ACM 2475-1421/2025/10-ART412
https://doi.org/10.1145/3764068

important bugs) in less time than if they wrote unit tests instead. However, it is not obvious whether this is the case. On one hand, property-based tests check the code against large classes of inputs, rather than requiring developers to exhaustively identify all interesting cases. On the other hand, property-based tests are more complex to write than unit tests because, rather than only reasoning about *one* input/output pair, the author of the test must specify properties that should hold for a family of inputs. Also, often, the properties expressed are only *partial* correctness properties, since testing whether the output of the function under test *equals* some known-correct value requires a (usually unavailable) oracle. In contrast, unit tests are usually simple to write, do not require abstract reasoning, and can compare to specific outputs that the developer can manually check.

Unfortunately, knowledge of how effective PBT is at finding bugs is limited. Observational studies have examined the use of property-based testing in Python and the library Hypothesis, looking at how developers use PBT outside a pure functional language [9, 44]. Although these studies shed light on the challenges of using PBT techniques, they do not yet provide quantitative evidence regarding whether property-based tests actually find more bugs than unit tests. Further, while some work identifies some informal categories of property-based tests that may be effective, little is known about what kinds of properties developers should choose to maximize the bug-finding power of PBT.

In this work, we seek to understand the relationship between testing method (unit test vs. PBT) and tests' ability to find bugs. We are particularly interested in what kinds of properties are used in existing property-based tests and to what extent different kinds of properties are more effective for finding bugs. Thus, we focus on three research questions:

RQ1: What kinds of properties do Python developers check in property-based tests?

RQ2: How effective are these kinds of properties at catching bugs, and how do PBTs overall compare in effectiveness to unit tests?

RQ3: How does the bug-finding efficacy of property-based tests relate to the number of inputs used when executing them?

To answer these questions in a setting that maximizes external validity, we collected a corpus of 426 Python programs that import the Hypothesis library. We identified and formalized 12 categories of property-based tests we found in the corpus, defined in subsection 4.2. We implemented a static analysis to find instances of these categories, which accounted for 94% of all property-based tests in our corpus. Since mutation testing has been found to be a good approximation of real-world bugs [25], we used mutation testing to assess the bug-finding power of each category of test. After testing 40 of the projects in our corpus, we found that the property-based tests outperform unit tests in their ability to kill mutants by a factor of about 50, even when controlling for the test suite's code coverage percentage. We also found that *exception raising* tests (tests that check that a particular exception is triggered by an operation) were the most successful at catching mutations. Finally, we assessed how PBT's bug-finding efficacy relates to the number of inputs used, finding that 55% of mutations are found with a single input, and 76% within the first 20.

Our work focuses on the benefits of PBTs, leaving for future work the important question of comparing the human effort needed to write and maintain a PBT to that required for a unit test. Unit tests are much more popular in our corpus than PBTs: 98.4% of all tests were unit tests, so overall, unit tests found more bugs than PBTs, and we expect that future software systems will continue to incorporate unit tests. We also suspect that the bug-finding power of PBTs may not arise from the random generation of many test inputs: our parameter sweep experiments showed that over half of the mutations found by a PBT were found with a single input. One possible explanation is that the average PBT is better written than the average unit test, suggesting that thinking in terms of properties may help developers write better tests overall. Another possibility is that developers

who choose to write PBTs are often more skilled at writing tests than those who prefer to write unit tests.

This paper's key contributions are:

(1) Formal definitions for 12 categories of property-based tests and a property-based test classification tool, showing that 94% of property-based tests in the corpus fall into one of the categories.
(2) A categorization of property-based tests in a corpus of 426 Python programs.
(3) An evaluation of effectiveness for 40 test suites containing property-based tests, comparing unit tests to PBTs and comparing between PBT categories, showing that on average, each property-based test kills about 50 times as many mutants as the average unit test.
(4) A parameter sweep experiment of the time cost and fault-revealing capacity of PBT, showing that property-based tests found 76% of mutations within the first 20 inputs.

Although we cannot infer causation from our corpus study, which did not involve random assignment, this is the first quantitative evidence of which we are aware that on a per-test basis, many categories of property-based tests may be more effective at finding bugs than unit tests.

2 Property-Based Testing

In property-based testing, developers specify all the possible valid test inputs and a particular property that is supposed to hold of the tested code. When the test is run, a *generator* provides randomly generated instances of the test input based on a specification, and the property is checked. The Haskell library QuickCheck [8] popularized property-based testing, and PBT libraries exist in many languages, including Python. Hypothesis [33] is a popular PBT library for Python programs. In a survey of 25,000 Python developers in 2023, 5% reported using Hypothesis, placing it as the 6th most used testing framework in the survey [23]. Hypothesis has functions for generating basic inputs such as strings, integers, and collections, and it provides a way for users to create custom generators using these basic functions. Figure 1 shows a Hypothesis test function. To write a property-based test using Hypothesis, a user specifies the inputs to be generated with the `@given` decorator along with a generating function, called a strategy in Hypothesis. In the example, the `integers()` strategy generates integers between 0 and 10. Assert statements check properties: in this example, the assertion checks the property that the generated argument is less than 20.

```
1 @given(g=integers(min_value=0, max_value=10))
2 def test_function(g):
3     assert 20 - g > 0
```

Fig. 1. An example Hypothesis test function

Property-based testing is often used in conjunction with other testing approaches. In Python, PBTs are often written alongside unit tests and can be part of continuous integration (CI) pipelines [34]. Goldstein et al. [15] showed that PBT is used alongside example-based unit tests and fuzzing. In particular, they found that PBT is used more often than unit tests to test complex code, suggesting that property-based tests are written at a similar point in the development process as unit tests. In different cases, property-based tests can be more or less complicated to write than example-based unit tests, but generally they provide greater confidence to the developer [15].

Fuzzing is another approach that uses randomized inputs. Whereas PBT uses programmer-specified properties, fuzzers detect predefined incorrect behavior, such as crashes and security bugs [27]. This makes fuzzers complementary to PBT approaches, since fuzzers can be applied to check

standard properties without substantial programmer intervention, but programmers can use PBT can check domain-specific properties at any level of abstraction.

The recommended amount of time to run a fuzzer is 24 hours, making it infeasible to run after each code change [28]. Thus, fuzzing is usually not part of continuous integration (CI) pipelines. In contrast, property-based tests run on a set number of inputs. In the case of Hypothesis, this is 100. Hypothesis also has configurations for running tests in CI pipelines [34] and integrates into common Python test runners such as Pytest [10, 30]. PBT inputs are constructed using a generator function while the inputs produced by most fuzzers are mutated from a set of seed inputs [27]. Some testing techniques blur the lines between PBT and fuzzing such as JQF [37] which generates structured random inputs via coverage-guided fuzzing to check user-written properties.

Users can have difficulty coming up with properties to test [16], so guides for how to develop properties can be valuable. *How To Specify It!* [20] is one such guide for writing properties for pure functions. It recommends properties relating to function postconditions and the preservation of equivalence. Others [18, 35, 45] have also identified useful properties to test that are not explicitly for pure languages. Some examples of these properties include checking that a value is between a specific maximum and minimum and that some part of the code raises an exception. In addition to suggesting different kinds of properties, *How to Specify It!* contains a small-scale assessment of the bug-finding capabilities of the properties it identified and found that postcondition checks are the most effective. However, outside of a pure functional paradigm, there is little data on which kinds of property-based tests find the most bugs.

3 Method

Our methodology consisted of three steps. First, we collected Python projects that use the Hypothesis library for property-based testing into a corpus of 426 projects. Second, to address RQ1, we iteratively developed classifiers for property-based test assertions. We used informal descriptions of suggested properties as a basis and then identified new patterns after looking through the unclassified tests and developed detectors for these patterns. We classified the property-based tests in our corpus. Then, to assess our categorization, we classified the property-based tests in an additional dataset of 50 projects not included in the original corpus to assess our categorization. Next to address RQ2, we employed mutation testing to assess the effectiveness of the test suites containing PBTs. Finally, to address RQ3, we ran a parameter sweep experiment by executing PBTs with up to 500 inputs.

3.1 Collecting Python Projects

We collected two sets of Python-based projects: we used the first to *develop* our set of categories, and the second to *evaluate* our set of categories. To develop the categories, we used the Boa language [11] to collect a corpus of GitHub projects that used the Hypothesis library. Boa is a domain-specific language and infrastructure used to analyze large-scale software repositories. The Boa infrastructure maintains its own datasets that work with the Boa tools. It is hosted on a server that includes snapshots of GitHub projects, enabling efficient retrieval of code from GitHub with a given search query. We used the February 2022 Python snapshot, which was the most recent available as of October 2024. Our data query, which filtered the dataset to projects that use the Hypothesis library, resulted in 426 Python projects, or 0.42% of projects in the overall dataset. We include the list of these projects and the script used to collect them in the artifact. On average, the projects in our corpus contained 29,972 lines of Python code, with the median project containing 5,042 lines of Python code. The corpus contains 7,125 property-based tests in total. We include a histogram of the project sizes in Figure 2.

To evaluate our categorization, we collected a fresh set of programs that were newer than the most recent Boa snapshot. We used the GitHub API to search for Python projects that import

the Hypothesis library and were not in the previous dataset, and we found 50 projects using this method. We include the script used to collect these projects in the artifact [2].

3.2 Writing the Test Classifiers

To start creating a categorization of properties used in PBTs, we looked at property-based testing advice directed at novices. These guides suggested different kinds of properties to check in property-based tests [18, 35], and some guides are included in the Hypothesis library's official documentation [45]. We focused on the properties that could be detected structurally in order to automate the classification process, identifying five properties in this way. In contrast, some of the suggested properties were not amenable to objective classification. One such example, "hard to prove, easy to verify," is more subjective, and suggests testing algorithms whose implementations are difficult to prove correct, but whose results are easy to verify [45].

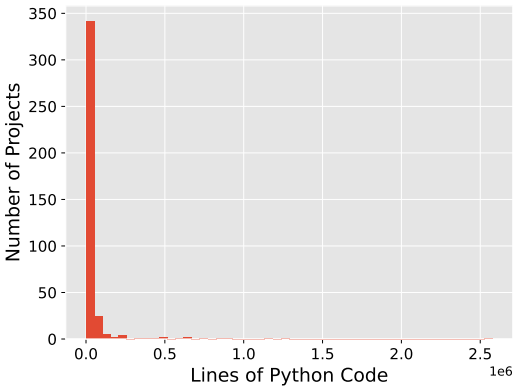


Fig. 2. Sizes of projects in corpus by lines of Python

Each test classification function (or *detector*) consists of an intraprocedural dataflow analysis. For each dataflow analysis, the data source is the input values randomly generated by Hypothesis (the *generated arguments*), and the data sinks are the `assert` statements of the test function. Dataflows occur in the following statements: assignment, function call, if statements, and loops. Expressions within these statements are considered *influenced* by an argument generated by Hypothesis if there is some sequence of dataflows from that argument to the expression.

We defined a property-based test assertion as one that involves an expression that has been influenced by an argument generated by Hypothesis. A PBT *function* is a test function that contains one or more property-based test assertions, and a test function that does not contain any is called a *unit test*. For example, to test an addition function, a possible PBT is `add(a,b) = add(b,a)`, while a possible unit test is `add(2,4) = 6`. We assumed that all function calls introduce data flows from function arguments to returned values as well as between function arguments, due to the possibility of mutation. This choice allowed our analysis to more completely capture possible dataflows. However, this assumption could have resulted in a greater number assertions being considered property-based test assertions since it introduces more dataflows. Test functions can include multiple assertions, and each assertion can have multiple kinds of property checks via boolean operations. Thus, we categorized each assertion separately, with some assertions being classified under more than one category.

The definitions inspired by the informal property recommendations accounted for five of the 12 property categories we defined and 48% of the tests in our corpus. To obtain a more comprehensive analysis, we added new property type definitions upon inspection of the uncategorized assertions: constant equality, generated-expression and constant non-equality, type checking, inclusion, and constant inclusion. We also incorporated more granular categories based on the structural characteristics of the tests, including whether comparisons included constant values. We split a category into two when there was a potential difference in test power between the two subcategories. This classification process resulted in 12 kinds of properties, listed in Table 1 and defined in subsection 4.2, and categorized 94% of all assertions in our corpus. We include the code for the detectors in the artifact [2].

3.3 Mutation Testing

To address RQ2, assessing the effectiveness of test categories, we used mutation testing to assess the bug-finding power of each project's test suite. Mutation testing or mutation analysis is a method of evaluating a test suite's effectiveness. In a mutation test, small changes are made to a program's source code, called mutations. The mutations are generated by mutation operators, which are syntactic transformation rules. Mutation operators often include swapping arithmetic operators (`+` to `-`), flipping logical operators (`and` to `or`), and changing literals. The test suite is then run on the mutated program, and if any test fails, then the mutation is considered *caught* or *killed*. If no test fails, then the test suite did not identify the bug. The number of mutations caught is the test suite's *mutation score*, and a higher score is indicative of a more robust test suite. Previous work has found that the mutations generated by mutation testing tools mimic real-world bugs and that mutation detection is positively correlated with real fault detection [25]. Thus, the more mutations a test suite is able to detect, the more effective it is. Later work [7, 38] identified a more nuanced relationship between test suite effectiveness, test suite size, and bug detection, so we report the sizes of each project's test suite. An earlier approach was to look at a test suite's code coverage, but Inozemtseva and Holmes [22] found that coverage is not strongly correlated with test suite effectiveness, thus we did not rely on test coverage to determine the effectiveness of property-based tests. Nonetheless, we conduct statistical tests controlling for test coverage when comparing property-based tests to unit tests by adding test coverage as an additional regression variable.

We used the tool MUTMUT [19] to create mutations in Python programs and run the project's test suite on those mutants. In order to keep a record of the tests that failed, we created a fork of the MUTMUT project [1]. The MUTMUT tool adds mutations to source code by editing specific nodes of the AST, including numbers, strings, lambdas, function arguments, operators, and decorators according to certain rules. For each possible mutation in the source code, MUTMUT runs the project's test suite on the mutated code and collects the names of the failed tests. By default, Hypothesis runs the property-based tests with 100 randomized inputs. To account for flaky tests, we ran a baseline of the test suite with no mutations 10 times and checked for failures.

We ran the mutation tests on a computing cluster [6]. To reduce computation time, we randomly selected a subset of 100 mutations. We used Pytest, the most commonly used Python testing library [23], to run the test suites as a majority of projects were equipped to be run with this library. For each mutation, we collected all the test functions that failed, indicating that those functions had found the bug produced by the mutation. If the function was in our corpus of property-based tests, then we noted which categories of property-based tests the function contained. From the original corpus of 426 Python projects using Hypothesis, we successfully ran mutation tests on 40 projects.

Mutation testing is computationally intensive even for small projects and fast-running test suites [3], so we limited the number of projects tested in this manner. We attempted to run the projects in order of name, and the first 40 projects that were able to run were included in our mutation testing experiment. We determined if a project was able to run by first installing the project dependencies and ensuring that all were able to be installed and none had incompatible versions. After this, we ran the test suites of the projects without any mutations. For projects with a few failing tests, we removed the failing tests and ran the rest of the test suite in our mutation trials. If more than 25% of the tests were failing without any mutations, then we did not select the project for our experiment. Skipping many failing tests would interfere with the validity of the mutation testing since the tests that failed the baseline may have also caught mutations. We include `git diff` files as part of the artifact as documentation of the changes we made [2].

We then analyzed the tests that caught mutations and identified them with their classification from the detectors. We ran three nominal logistic tests on our data using JMP statistical software

[21]. One test compared the ability of unit tests to catch mutations to that of property-based tests, another checked the effects of code coverage on the previous result, and the final test evaluated mutation catching ability within property-based test categories. To conduct the parameter sweep experiments, we tested the same projects on 100 randomly selected mutations and ran the property-based tests with 500 inputs. We recorded at which input, if any, the PBT caught the mutation.

3.4 Threats to Validity

Study Design. We conducted a corpus study of property-based testing usage in Python. As a result, we do not determine causation with respect to why certain test categories performed better in mutation testing. For example, it is possible that better programmers are more likely to write better tests and also that better programmers are more likely to choose to write property-based tests. However, we consider all of the tests in each corpus, and it is likely that the best programmers write both kinds of tests. It is also possible that property-based tests are more appropriate for some kinds of code than others, but we posit that even if property-based tests are only applied for some kinds of code, any benefits of those tests relative to unit tests will be reaped there.

The results from our study may be primarily applicable to property-based testing in the Python language. Property-based testing is most commonly found in functional languages, which may encourage users to specify different kinds of properties, so the percentages of tests found for each category may not hold in languages like Haskell and OCaml.

Assessing the practical considerations of the tradeoffs between writing unit tests and PBTs requires additional results not included in this paper. Potential avenues of research could include the amount of time needed to write a PBT, or how often PBTs need to be updated as code changes. Our work does not measure the amount of human effort needed to write and maintain a property-based test in comparison to a unit test, which we did not assess. Thus our study can only speak to the potential *bug-finding* power of PBTs in comparison to unit tests.

Test Classification. We made the assumption that all expressions used as arguments and returned by functions were subject to dataflows, meaning that the remaining arguments and return values could have been influenced by the generated expressions used as arguments to the function. This choice could have resulted in more assertions being considered property-based tests or a different breakdown of the categorizations since we may have overestimated the expressions influenced by a generated argument. However, the definitions we have produced would likely not have been affected by this change, as they are structural in nature.

Mutation Testing. Since mutation testing relies on a fully passing test suite, we skipped tests that failed with the unmutated source code. Generally, these tests failed due to errors in the existing source code, or because they required dependencies that could not be obtained or installed in the system used. We have included `git diff` files for each project that was changed as part of the artifact. As a result, some tests that may have caught certain mutations were skipped, making the test suite appear less robust. In addition, we ran a random sample of 100 mutations for each project instead of testing all possible mutations generated by MUTMUT due to mutation testing being computationally expensive [46]. This could have resulted in some tests not finding any mutations because they covered parts of the code that were not mutated. We note that when controlling for overall code coverage in the project, property-based tests remain about 50 times more effective than unit tests.

Our mutation testing used the Pytest Python testing library [30] to run the test suites, as a majority of projects were equipped to be run with this library. When tracking if tests passed or failed, Pytest records the test function, and not the specific assertion in its final report. Our test categories are on a per-assertion basis, with some assertions spanning multiple categories, so our

mutation testing results do not record which specific assertion failed. This is a limitation of our ability to discern which categories of property-based tests are the most effective. However, we have found that the average number of different test categories a function has is 1.65, so on average test functions do not contain assertions of multiple different test categories. We also report the correlations between category types in the appendix and note that the average correlation between two different categories of test is 0.059. Since our dataset was created in 2022, many projects in the corpus are not actively maintained and their test suites had many errors and failures. This, along with time and resource constraints, limited the number of projects we were able to test. In addition, testing practices may have changed since 2022, which could limit the external validity of our results. We also simulated the parameter sweep experiment by recording the first input at which a PBT caught a mutation. Due to the randomness of property-based testing, it may not always be the case that the n^{th} input will find the bug.

4 Test Category Definitions

Property-based tests written using the Hypothesis library have two components: a test function definition written by the user and one or more arguments generated by Hypothesis to the test function. These are parameters with arbitary values that the test function may reference. Each test function must also contain one or more assertions that are checked when the function is run.

Example. The `@given(...)` decorator is used to specify which argument should be generated. Figure 3 shows a test function `test_func` with parameters `a` and `g`. The `@given(g=integers())` decorator specifies that the parameter `g` will be an arbitrary integer. Injecting the arbitrary integer value into the argument turns `g` into a Hypothesis-generated argument. The `@given()` decorator does not affect the value of `a`, so it remains an un-generated test argument. `test_func` contains a single test: the sequence starting at the first statement of the function in line 3 and ending at line 5, an `assert` statement. A dataflow analysis on these sequence of statements will categorize this test.

Definitions. Let f be a test function, $\bar{g} = \{g_1, ..., g_k\}$ be the arguments generated by Hypothesis, and $\bar{s} = s_1, ..., s_n$ be the sequence of statements in the body of f. Since f is a test function, there exists some index $a \in \{1, ..., n\}$ such that s_a is an assertion statement. We call a an *assertion index*. For analysis purposes, since test functions can contain more than one assertion statement, we treat each assertion and the sequence of statements from the beginning of the function to that assertion as a separate test. Formally, given an assertion index a, we define a *test* as $\bar{s}_{\leq a}$, where $\bar{s}_{\leq a}$ is the sequence of statements from s_1 to s_a. We also denote the statements from s_1 to $s_{(a-1)}$ as $\bar{s}_{<a}$.

Example. In Figure 3, $\bar{g}$ is {g}, and $\bar{s}$ is the sequence (`b = g + a`, `c = h(b + 2)`, `assert a == c - g - 2`). There is one assertion in f, s_3, which is `assert a == c - g - 2`, at index 3, so we have one test, $\bar{s}_{\leq 3} = \bar{s}$ in f.

```
1 @given(g=integers())
2 def test_func(a, g):
3     b = g + a
4     c = h(b + 2)
5     assert a == c - g - 2
```

Fig. 3. An example Hypothesis test function.

We base our analysis on expressions that could be used as subexpressions in an arbitrary `assert` statement. For example, the expressions `a + b` and `f(c, d)` are arithmetic and function call subexpressions that can be used in the assertion `assert a + b == f(c, d)`. Thus, we define *expressions* to be instances of this subset of Python expressions. We provide the complete list of such expression types in the appendix.

Let Exprs denote the set of such expressions in $\bar{s}$ and fix a to be the index of an arbitrary assertion statement in $\bar{s}$. To conduct our analysis, we wish to record all the expressions used in the statements *before* the assertion statement s_a to determine the influences on the subexpressions of s_a. To that end, we define $\text{Exprs}_{<a}$ to be the set of expressions $e \in \text{Exprs}$ such that e is a subexpression

in some statement in $\bar{s}_{<a}$. In Figure 3, $\text{Exprs}_{<a}$ is the set { a, g, g + a, b, h(b + 2), b + 2, c }. These represent all the expressions that could introduce dataflows to the expressions in the `assert` statement. If an assertion s_a is not subject to any dataflows, then we call the test $s_{\leq a}$ a *unit test*.

4.1 Dataflow Analysis Definition

In this section, we specify the relevant definitions for our intraprocedural dataflow analysis. Our analysis is designed to show how the values of the generated arguments influence the outcome of the test. To capture all possible dataflows, we conservatively defined dataflows and added flows between collection elements and function arguments.

Example. Our analysis conveys how g influences the outcome of the assertion `assert a == c - g - 2` in Figure 3. First, g is used in the assignment of b. Then, b is used in the assignment of c, so the influences on c are { b, g }. c and g are both used in the `assert` statement, so the outcome of the assertion is dependent on g, c, and by transitivity, b.

In our analysis, we associate each variable in the test with a set of influences represented by a set of variables. For each variable x, we construct the set of influencing variables, $G(x)$. As notation, we use $G(e)$ to denote the related variables to an expression $e \in \text{Exprs}_{<a}$, i.e. $(e, G(e)) = \bigvee_{x \in \text{idents}(e)}(x, G(x))$, where $\text{idents}(e)$ is the set of variables in expression e.

4.1.1 Constructing $G(e)$. First, we construct $G(e)$ for the test function arguments. For each argument $g_i \in \bar{g}$, $G(g_i) = \{g_i\}$, and for any other argument b, $G(b) = \{\}$. Different expressions can change an variable's influences, so the value of $G(e)$ is dependent on the type of e.

We list some of the judgements for $G(e)$ where e is an expression Figure 4, and the rest in the appendix in Figure 26. For expressions where there are multiple possible configurations, such as `and`, `or`, `not` in boolean expressions, we provide a representative example. We adopted conventions from the Python language reference [13].

$\boxed{G(e)}$

$$\frac{n \in \text{idents}(\text{Exprs}_{<a}) \qquad i : \texttt{int}}{G(n[i]) = G(n)}\text{Subscript} \qquad \frac{e_1, ..., e_k \in \text{Exprs}_{<a}}{G(f(e_1, ..., e_k)) = \bigcup_{k=1}^{m} G(e_k)}\text{Call}$$

$$\frac{n, n' \in \text{idents}(\text{Exprs}_{<a}) \qquad G(n) \neq \emptyset}{G(n') = \{n\} \cup G(n)}\text{GenIdentifier}$$

$$\frac{n, n' \in \text{idents}(\text{Exprs}_{<a}) \qquad G(n) = \emptyset}{G(n') = G(n)}\text{NonGenIdentifier} \qquad \frac{}{G(\texttt{atom}) = \{\}}\text{Literal}$$

$$\frac{e_1, ..., e_m \in \text{Exprs}_{<a}}{G(Collection(e_1, ..., e_m)) = \bigcup_{k=1}^{m} G(e_k)}\text{Collection} \qquad \frac{e_1, e_2 \in \text{Exprs}_{<a}}{G(e_1 + e_2) = G(e_1) \cup G(e_2)}\text{ArithmeticExpr}$$

$$\frac{n \in \text{idents}(\text{Exprs}_{<a}) \qquad x \in \texttt{attr}(n)}{G(n.x) = G(n)}\text{Attribute} \qquad \frac{e_1, e_2 \in \text{Exprs}_{<a}}{G(e_1 < e_2) = G(e_1) \cup G(e_2)}\text{Comparison}$$

Fig. 4. A partial definition of G(e), which computes the variables from which an expression is derived. A full definition of G(e) is provided in the appendix.

When e is a subscript or slice from a collection, i.e. $e = n[i]$ where n is a collection, $G(e)$ is dependent on $G(n)$, not just the value of G for the i^{th} entry of n, since Python allows collections to be rearranged. We use $G(n)$ rather than n to capture all variables from which n may have been derived rather than n itself.

For example, in Figure 5, the value of b changes depending on the value of g, so we consider g an influence on b. Thus, $G(l[0]) = G(l)$. When e is a function call, we consider it influenced by all the arguments to the function. So, for arguments $\bar{e} = e_1, ..., e_k$, $G(\bar{e})$ is the union of $G(e_i)$.

```
1 @given(g=integers())
2 def test_example2(g):
3     l = [2, g, 3]
4     l.sort()
5     b = l[0]
6     assert(...)
```

Fig. 5. An example of list mutation. We assume that generators influencing any index may influence all indices.

All the possible combinations of variables and their influences, our *dataflow facts*, make up the join semi-lattice that we use to define our analysis. One dataflow fact reflects all the variables whose values may influence the value of a given variable. A set of dataflow facts, $D_{<a}$, is specific to a test $\bar{s}_{\leq a}$.

For the example in Figure 3, our test is $\bar{s}_{\leq 3}$, so our dataflow facts $D_{<3}$ are (g, {g}), (b, {g}), (c, {b, g}). Expressions in an assertion statement s_a are not used to determine $D_{<a}$. Thus, each dataflow fact shows the relationship between an variable x used in $\text{Exprs}_{<a}$ and the arguments $\bar{g}$ generated by Hypothesis.

Let $\text{idents}(\text{Exprs}_{<a})$ be the set of variables in $\text{Exprs}_{<a}$. In Figure 3, $\text{idents}(\text{Exprs}_{<a})$ would be { g,a,b,c }. Our set of dataflow facts, $D_{<a}$, is the set $\{(x, G) : x \in \text{idents}(\text{Exprs}_{<a}), G \subseteq \text{idents}(\text{Exprs}_{<a})\}$, where x is an variable and G is as defined previously. We use D as an abbreviation for $D_{<a}$ when the test being referred to is clear.

Our dataflow analysis uses a semi-lattice on D with partial order $\preccurlyeq$ and join operator $\vee$. We define the partial order $\preccurlyeq$ on D as set inclusion, i.e., $S_1 \preccurlyeq S_2 \stackrel{\text{def}}{=} S_1 \subseteq S_2$, and the join operator $\vee$ as

$$\{(x_i, G_i) : i = 1, ..., m\} \vee \{(y_j, H_j) : j = 1, ..., n\} = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} \mathbb{G}_{i,j}$$

$$\text{where } \mathbb{G}_{i,j} = \begin{cases} \{(x_i, G_i \cup H_j)\}, & \text{if } x_i = y_j \\ \{(x_i, G_i), (y_j, H_j)\}, & \text{otherwise} \end{cases}$$

Example. In Figure 3, the set of variables is $\text{idents}(\text{Exprs}_{<3}) = \{\text{g}, \text{b}, \text{c}\}$, and the dataflow facts for this test are given by $D_{<3} = \{\{(g, \{g\}), (b, \{g\}), (c, \{b, g\})\}$.

4.1.2 Computing Dataflow Facts. Now, we give a procedure for computing $D_{<a}$ on a sequence of statements $\bar{s}_{\leq a}$.

Example. The test function in Figure 3 has two assignment statements and an assertion statement. At the beginning of the function, our only dataflow facts are those for the generated arguments, in this case, (g, {g}). After the first assignment statement, `b = g + a`, we add to our set of dataflow facts the tuple (b, {g}), reflecting flow from g to b. After the second assignment statement, we add the tuple (c, {g, b}). No dataflows occur in an assertion statement, so the final set of dataflow facts are {(g, {g}), (b, {g}), (c, {g, b})}.

The inference rules in Figure 6 show how to inductively construct dataflow facts for a sequence of statements $\bar{s}; s$, where ; is the sequencing operation. The figure shows a subset of the rules, with the rest located in the appendix in Figure 27. If s contains branches, then the value of $G(e)$ for an expression e will be propagated to all branches. In information flow control literature, this is called an *implicit flow*, since e influences which branch is taken [36]. Thus, for a branch $\bar{s}'$ of s, we write

$G(e) \cdot D_{s'}$ for $\{(x, G(x) \cup G(e)) : x \in \text{idents}(\text{Exprs}_{\bar{s}'})\}$ to add the influence of e to all variables x in $\bar{s}'$. We include implicit flows in our analysis to account for all possible dataflows.

$\boxed{D(\bar{s}; s_k)}$

$$\frac{(e_1, G(e_1)) \in D(\bar{s})}{D(\bar{s}; e_1 = e_2) = (D(\bar{s}) \backslash (e_1, G(e_1))) \cup (e_1, G(e_2))}\text{Assign} \qquad \frac{}{D(\bar{s}; \texttt{assert } e) = D(\bar{s})}\text{Assert}$$

$$\frac{(e, G(e)) \subset D(\bar{s})}{D(\bar{s}; \texttt{if } e \texttt{ then } \bar{s}_1 \texttt{ else } \bar{s}_2) = D(\bar{s}) \cup G(e) \cdot D(s_1) \cup G(e) \cdot D(\bar{s}_2)}\text{IfGen}$$

$$\frac{(e, G(e)) \not\subset D(\bar{s})}{D(\bar{s}; \texttt{if } e \texttt{ then } \bar{s}_1 \texttt{ else } \bar{s}_2) = D(\bar{s}) \cup D(\bar{s}_1) \cup D(\bar{s}_2)}\text{IfNonGen}$$

Fig. 6. Example rules for statements.

Assigning to an expression e_1 replaces $G(e_1)$ with the G constructed from the right-hand side expression. The only expressions that can be assigned to are subscripts, slices, and attributes, so $(e_1, G(e_1)) = (x, G(x))$ for some identifier x. The inference rules for conditionals are split into two judgements, based on if the test e has a nonempty $G(e)$. If $G(e) \neq \emptyset$, then e influences the expressions in the two branches, so the dataflow fact for e must be propagated to sub-statements s_1 and s_2. Otherwise, $D(\bar{s}_1)$ and $D(\bar{s}_2)$ can remain unchanged.

4.2 Test Category Definitions

In this section, we provide descriptions and definitions of each category of test using the formalization developed in the subsections above. We provide examples from the corpus to illustrate each pattern, which have been edited for clarity. We introduce some terminology, *generated expressions*, which refer to expressions that have been influenced by some argument generated by Hypothesis, and *constants*, which denote all other expressions in the test. Formally, e is a generated expression if $G(e) \neq \emptyset$, and e is a constant if $G(e) = \emptyset$. If a generated argument $g \in \bar{g}$ is in $G(e)$, then we say e is *derived from* g. We consider an assertion a property-based test if it involves one or more generated expressions or arguments. Many categories are split on the basis of whether they involve generated expressions or constants, as we consider assertions that compare to constants as checking an invariant with respect to the generated arguments. The diagrams in Figure 7 are visual representations of the categories we found. Each of the blocks represent an expression, and the arrows represent (transitive) influences on an expression. The blue lines show the operation of the assertion: equality, inequality, non-equality, inclusion, type check, and exception. The examples in this section come from the projects in our corpus.

The categories we identified are disjoint: the constant and generated-expression versions of categories cannot overlap since no expression is both a constant and a generated-expression. Roundtrip, and partial roundtrip, tests and commutative path tests differ according to the overlap of the influences on the left-hand- and right-hand-sides.

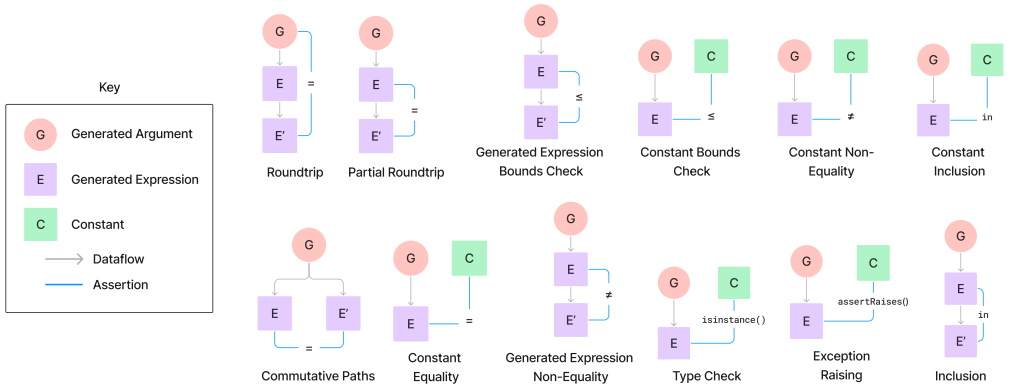


Fig. 7. Diagrams of test category structures

4.2.1 Tests of Equality. These pertain to assertions of the form `assert` e_L `==` e_R.

Roundtrip. In a roundtrip test, one or more transformations is applied to a generated argument, $g_i \in \bar{g}$. Then, an assertion checks equality between the original generated argument and an expression derived from that argument. In Figure 8, `idx` is an argument generated by Hypothesis, and `index` is derived from `idx`. An assertion is a roundtrip test if one of the following holds

(1) $e_L \in \bar{g}$ and $e_L \in G(e_R)$
(2) $e_R \in \bar{g}$ and $e_R \in G(e_L)$

```
@given(ndindices)
def test_ndindex(idx):
    index = ndindex(idx)
    assert index == idx
    ...
```

Fig. 8. Example of a roundtrip test.

Partial Roundtrip. A partial roundtrip test differs from a classic roundtrip test by checking equality between a generated expression and one derived from it. They are roundtrip tests that do not check for equality with a generated argument. Formally, a partial roundtrip test contains an assertion where one of the following holds:

(1) $e_L, e_R \notin \bar{g}$ and $G(e_L) \subset G(e_R)$
(2) $e_L, e_R \notin \bar{g}$ and $G(e_R) \subset G(e_L)$

```
@given(data=st.data())
@schema_strategy_params
def test_canonicalises_to_equivalent_fixpoint(schema_strategy, data):
    """Check that an object drawn from an arbitrary schema is valid."""
    schema = data.draw(schema_strategy, label="schema")
    cc = canonicalish(schema)
    assert cc == canonicalish(cc)
```

Fig. 9. Example of a partial roundtrip test.

In Figure 9, $G(\texttt{cc}) = \{\texttt{data}, \texttt{schema}\}$ and $G(\texttt{canonicalish(cc)}) = \{\texttt{data}, \texttt{schema}, \texttt{cc}\}$, so $G(\texttt{cc}) \subset G(\texttt{canonicalish(cc)})$ Informally, `data` is the value generated by Hypothesis, and it is used to produce `schema`. `schema` is then used to produce `cc`, which is then compared to the result

of `canonicalish(cc)`, creating a roundtrip between `cc` and `canonicalish(cc)`. Since `cc` is not a generated expression, the assertion is a partial roundtrip test.

Although this kind of test is not a pattern described in other literature, we include this as a separate category from roundtrip tests. One possible partial roundtrip test could involve checking equality between one field of an object. In Figure 10, the assertion checks the equality of `g.a` and `b`, which is derived from `g.a`. A roundtrip version would ensure that the other fields of `g` are as expected, making it a stronger check.

```
@given(g=obj_w_int_field())
def test_partial_roundtrip(g):
    a = g.a + 1
    b = a - 1
    assert g.a == b
```

Fig. 10. Partial roundtrip test of an object field.

```
@given(max_len=integers(min_value=4, max_value=MAX_ELEMENT_LEN))
def test_partial_from_element(element, id_, max_len):
    packets = to_packets(max_len, element, id_, element.serialize())
    partial = PartialElement.from_element(element, id_, max_len)
    ...
    assert partial.part_total == len(packets)

```

Fig. 11. Example of a commutative paths test.

Commutative Paths. A commutative paths test checks for equality across dataflow branches. The two expressions being compared are influenced by some shared generated argument, but one value cannot be derived using the other. By definition, we have a commutative paths assertion if

(1) $\exists g \in \bar{g}$ such that $g \in G(e_L) \cap G(e_R)$
(2) $\exists x \in (G(e_L) \backslash \bar{g}) \wedge x \notin (G(e_R) \backslash \bar{g})$
(3) $\exists x \in (G(e_R) \backslash \bar{g}) \wedge x \notin (G(e_L) \backslash \bar{g})$

In Figure 11, we have one argument generated by Hypothesis `max_len`. Then, `max_len` is used to produce `packets` and `partial` independently of each other. The assertion checks if `partial.part_total == len(packets)`, so in the example, $G(e_L)$ = {`max_len, element,id_,partial`} and $G(e_R)$ = {`max_len, element, id_, packets`}.

Constant Equality. A constant equality assertion checks equality between an expression that has been influenced by a generated argument and an expression that has not been influenced by such an argument.

Formally, a test is in the constant equality category if either $G(e_L) = \emptyset$ and $G(e_R) \neq \emptyset$, or $G(e_R) = \emptyset$ and $G(e_L) \neq \emptyset$. In the example in Figure 12, `a % d` is checked if it is equal to zero, and we have $G($`a % d`$) = \{$`a, b, d`$\}$ and $G(\emptyset) = \{\}$.

4.2.2 Tests of Inequality. Tests of inequality and non-equality enforce bounds on the possible values an expression can take. These assertions are of the form `assert` e_L R e_R, where R is one of `<,>,<=,>=,!=`.

Generated-Expression and Constant Bounds Checking. (e.g., `assert` e_L `<` e_R) These two kinds of tests check that an expression stays within the specified bounds. An assertion checks *generated expression bounds* if generated values influence *both* sides, i.e., $G(e_L) \neq \emptyset$ and $G(e_R) \neq \emptyset$. A *constant inequality bounds check* only uses generated values on *one* side of the comparison: either $G(e_L) = \emptyset$ and $G(e_R) \neq \emptyset$ or $G(e_R) = \emptyset$ and $G(e_L) \neq \emptyset$. Both kinds of test use one of the following

```
@given(st.integers(min_value=1, max_value=1000), st.integers(min_value=1,
    max_value=1000))
def test_extended_euclid(a, b):
    x, y, d = extended_euclid(a, b)
    assert a % d == 0

```

Fig. 12. Example of a constant equality test.

relation operators: `<`, `>`, `<=`, `>=`. We consider these as two separate categories because using a generated expression as a boundary shows a relationship between different transformations while a constant bound expresses an invariant.

```
@given(st_bounds())
def test_bounds_generation(bounds):
    assert len(bounds) % 2 == 0
    n = len(bounds) // 2
    for d in range(n):
        assert bounds[d + n] >= bounds[d]
```

Fig. 13. Example of a generated expressions bounds check test.

```
@given(dtm=strategy_dtm(),matrix_type=st.integers(min_value=0, max_value=1),
       proportions=st.integers(min_value=0, max_value=2))
def test_codoc_frequencies(dtm, matrix_type, proportions):
    ...
    cooc = bow.bow_stats.codoc_frequencies(dtm, proportions=proportions)
    ...
    if proportions > 0:
        assert np.all(cooc <= 1)
```

Fig. 14. Example of a constant bounds check test.

In Figure 13, `bounds[d + n]` and `bounds[d]` are compared to check if the latter half of `bounds` has entries larger than its corresponding entry the first half. The constant bounds check in Figure 14 determines if all entries in the array `cooc` are less than or equal to 1.

Constant Non-Equality and Generated Expression Non-Equality. (`assert` e_L `!=` e_R) Some tests check that a value derived from a generated argument is not equal to a particular value. If that value is also derived from a generated argument, then we consider it a separate category from non-equality being checked with a constant value. Formally, a test checks *constant non-equality* if

(1) $G(e_L) = \emptyset$ and $G(e_R) \neq \emptyset$, or (2) $G(e_R) = \emptyset$ and $G(e_L) \neq \emptyset$

and asserts that $e_L \neq e_R$. For a *generated expression non-equality* assertion A, $G(e_L) \neq \emptyset$, $G(e_R) \neq \emptyset$, and A asserts that $e_L \neq e_R$. The assertion in Figure 15 checks that `reduced.start` is not `None`, which is a relatively weak test since `reduced.start` can have almost any other value. From the comment, we can see that this test is meant to check if the postconditions of the `Slice.reduce()` method are met. In Figure 16, the assertion checks that casting unequal instances of a type to integers

results in unequal integers. While this is also a weaker test since it does not say anything about the properties of `int(x)` with respect to `x`, it encodes that the mapping from `schroedintegers`, a type created by the project, to `int` is injective.

```
@given(slices(), one_of(integers(0, 100), shapes), reduce_kwargs)
def test_slice_reduce_no_shape_hypothesis(s, shape, kwargs):
    ...
    try:
        S = Slice(s)
    except ValueError: # pragma: no cover
        assume(False)
    ...
    # Check the conditions stated by the Slice.reduce() docstring
    reduced = S.reduce(**kwargs)
    assert reduced.start != None
```

Fig. 15. Example of a constant non-equality test.

```
@given(schroedintegers, schroedintegers)
def test_distinct_integers_give_distinct_values(x, y):
    assume(x != y)
    assert int(x) != int(y)
```

Fig. 16. Example of a generated expression non-equality test.

4.2.3 Other Tests. This group encompasses any test categories that do not directly fit into tests of equality and inequality. Instead, they check other properties such as inclusion in a collection, the type of a variable, and if an exception is raised.

Inclusion and Constant Inclusion. (e.g., `assert` e_1 `in` e_2) Inclusion tests check whether a collection contains a particular item. We again split this kind of test based on whether it checks that a *constant* or a *generated expression* is in some collection. An inclusion check assertion is an assertion with $G(e_1) \neq \emptyset$, and a constant inclusion check assertion is of the form `assert` e_1 `in` e_2 where $G(e_1) = \emptyset$ and $G(e_2) \neq \emptyset$.

In Figure 17, both `-estimate0` and `(estimate1, estimate2, estimate3)` are derived from the generated argument `args`, so this assertion is an inclusion test. Figure 18 is a constant inclusion test: `ast` is influenced by generated argument `expr`, and the assertion checks that the literal `'children'` is in a field of `ast`. As in other tests with constants, constant inclusion tests can serve as an invariant check. An inclusion test involving two generated expressions can convey stronger properties: the test in Figure 17 shows that for any valid input, `-estimate0` is always one of `estimate1,estimate2,estimate3`.

Type checking. (e.g., `assert isinstance(`e_1`,` e_2`))` Some property-based tests check that a variable is of a certain type since Python's dynamic type system allows functions to return values of different types. These tests can check that a variable is of a given type or that the types of two variables match. There are two built-in methods for checking a variable's type, `type()` and `isinstance()`. Both methods are allowed in a type checking assertion, which can be of the form `assert isinstance(`e_1`,` e_2`)` where $G(e_1) \neq \emptyset$, or `assert type(`e_1`) == type(`e_2`)`, where either

```
@given(gufunc(
       "(n),(),(),()->()", dtype=[np.float_, np.float_, np.int_, np.float_],
           elements=[floats_, probs, counts, probs]
       )
)
def test_min_quantile_CI_to_max(args):
    X, q, m, alpha = args
    epsilon = 1e-8 # Small allowance for numerics
    estimate0, LB0, UB0 = qt.min_quantile_CI(X, q, m, alpha)
    estimate1, LB1, UB1 = qt.max_quantile_CI(-X, (1.0 - q) - epsilon, m, alpha)
    estimate2, LB2, UB2 = qt.max_quantile_CI(-X, 1.0 - q, m, alpha)
    estimate3, LB3, UB3 = qt.max_quantile_CI(-X, (1.0 - q) + epsilon, m, alpha)
    ...
    else:
        assert -estimate0 in (estimate1, estimate2, estimate3)
```

Fig. 17. Example of an inclusion test

```
@given(st.text())
def test_parser_api_from_str(expr):
    ...
    try:
        ast = Parser().parse(expr)
    ...
    assert 'children' in ast.parsed
```

Fig. 18. Example of a constant inclusion test

```
@given(name=symbols)
def test_symbol_is_parsed(
    decode_simple, name):
    parsed =
        decode_simple(f"Symbol({name!r})")
    assert isinstance(parsed, Symbol)
```

Fig. 19. Example of a type checking test

$G(e_1) \neq \emptyset$ or $G(e_2) \neq \emptyset$. The example in Figure 19 checks that the parsed output of the generated argument `name` is of type `Symbol`.

Exception Raising. (e.g., `assertRaises(exception, f, args, kw_args)`) Some tests are meant to trigger errors, so exception raising tests check that a specific exception was raised when an error was encountered. We defined these tests using Python's built-in `unittest` library function `assertRaises`, as these were the most commonly seen assertions that checked for exceptions. Formally, we have an exception raising assertion if it is of the form `assertRaises(exception, f, args, kw_args)` where $G(\texttt{f}) \neq \emptyset, G(\texttt{args}) \neq \emptyset$, and $G(\texttt{kw\_args}) \neq \emptyset$. The first argument, `exception`, is the exception name, while the latter three specify the called function and its arguments. The example in Figure 20 shows an assertion that raises a `RuntimeError` when the function `SQLiteConnection` is called with `engine_str`, an expression derived from generated argument `s`.

```
@given(text(alphabet="/ "))
def test_invalid_ending_name_connect(self, s):
    engine_str = "sqlite:///{}".format(
        os.path.join(self.tmp_dir.name,s))
    self.assertRaises(RuntimeError, SQLiteConnection, engine_str)
```

Fig. 20. Example of an exception raising test.

5 Results

Here, we present our results from the test classification to address RQ1: determining the kinds of properties Python developers check in property-based tests. On average, a project from our corpus contains 29,972 lines of Python code, and there are 7,125 property-based tests identified in our corpus. We adopted the intraprocedural forward dataflow analysis described in section 4 and defined property types based on the flow patterns. We found that 94.4% of all property-based assertions in our corpus were categorized. The most common category of test was constant equality (41.25%), followed by commutative paths (23.40%), and roundtrip (6.99%). Running the detection tool on the test set found 93 property-based tests and categorized with 94.62 % of tests. In this test set, the most common categories were constant equality (38.71%), roundtrip (29.03%), and commutative (13.98%). The percentages found in both data sets for each category are listed in Table 1.

Table 1. Categorization of Tests
Note: the percentages add up to more than 100% since some assertions can include more than one category of test, such as `assert a == 3 or a in [4,5,6]`. There were 426 projects in the corpus and 50 in the test set.

Category Type	Category	Corpus Percent	Test Set Percent
Test of Equality	Roundtrip	6.99%	29.03%
	Partial Roundtrip	7.03%	0%
	Commutative Paths	23.40%	13.98%
	Constant Equality	41.25%	38.71%
Test of Inequality	Generated-Expression Bounds Checking	4.08%	4.09%
	Constant Bounds Checking	6.05%	1.08%
	Generated-Expression Non-Equality	1.26%	1.08%
	Constant Non-Equality	0.84%	1.08%
Other	Typechecking	7.21%	3.22%
	Inclusion	2.71%	6.45%
	Constant Inclusion	1.79%	0%
	Exception Raising	0.28%	0%

Since we categorized individual assertions, some test functions contained multiple different kinds of tests. The most commonly paired test categories were constant bounds checking and constant inclusion with a correlation coefficient of 0.63, and roundtrip and partial roundtrip with a correlation coefficient of 0.33. The complete list of correlations are found in the appendix.

Our tool was able to categorize around 95% of the tests in both the corpus and the test set, but the percentages of tests in each category were significantly different. We conducted a Fisher's exact test and obtained a p-value of 0.0005, indicating that there is an association between the categorization of a test and whether it came from the corpus or the test set. The categories of tests programmers write may be particular to the programs that they are trying to test. Certain categories of test may be more prevalent in some projects over others, leading to the differences in the number of tests in each category.

RQ 1 Result: The most common PBTs in our corpora include PBTs checking equality with constants (41%, 39% in test set), commutative paths (24%, 14%), and roundtrips (7%, 29%).

5.1 Mutation Testing Results

To address RQ2, we used mutation testing to assess the effectiveness of the collected projects' test suites. To ensure that any test failures were due to mutations rather than pre-existing bugs, we checked projects in alphabetical order by project name and removed any that had many failing tests. For projects that had few failing tests (<25%), we disabled the failing tests. This resulted in a set of 40 Python projects from our corpus. Mutation testing is a computationally expensive method [3], so we chose to use 100 randomly sampled mutations to reduce computation costs, which is a limitation of our method. For each mutation, we collected all the test functions that failed, indicating that those functions had found the bug that the mutation had introduced. If the function included any property-based tests, then we recorded which categories of property-based tests were used in that function.

Dataset information. The 40 Python projects tested contained 350 property-based tests and 21,222 unit tests. Descriptions of these projects and information on the number of tests in each project is provided in Table 2. We classified projects based on their project descriptions and READMEs, and found eight project types: applications (4 projects), cryptography-related (5), language tooling (5), machine learning-related (3), networking related (5), scientific programs (6), utilities (8) and visualization tools (4). Project sizes ranged from 531 to 55,189 lines of Python code. The majority of projects had fewer than 10,000 lines of code, as seen in Figure 22.

Of the property-based tests, 137 caught mutations (39.14%), and 1394 unit tests (6.6%) caught mutations. The average number of mutations caught per project was 40.5, and the maximum number of mutations caught by a project's test suite was 96. Over all projects, the test suites detected 1,610 mutations out of 4,000. 1,365 of the mutations were found by unit tests (84.7%) and 467 (34.2%) were found by PBTs. Around six mutations per project were caught by both PBTs and unit tests, or 15% of all mutations found.

Statistical Tests. After obtaining our data, we conducted nominal logistic regression tests on our data with a response variable of whether a mutation was caught by a failed test. We conducted two of these tests for our data, one assessing whether property-based tests or unit tests were more effective at catching mutations, and another to determine if there is a link between category of PBTs and the number of mutations caught. To assess the effect of code coverage on our results, we conducted an additional nominal logistic regression test by adding code coverage as a predictor. We found that property-based tests were more effective than unit tests ($p < .0001$), and that certain property-based test categories performed better. In particular, the exception raising, inclusion, and type checking categories performed the best. However, each mutation is the result of a single change, such as replacing + with −, so the fraction of tests that a mutation would affect is likely small. Unsurprisingly, the R^2 values for our models were also small, ranging between 0 and 0.02.

Unit test vs. property-based tests. We had a significant result for the relationship between the number of mutations caught and whether a test was a property-based test or a unit test ($\chi^2(1, N = 1916905) = 5015.334$, $p < .0001$). Property-based tests were more likely to catch mutations than unit tests. The contingency table in Table 3 shows the breakdown of percentages of the data for the times a mutation was caught by a property-based test or a unit test. Unit tests had an odds ratio of 0.019, indicating that unit tests are around $\frac{1}{50}^{\text{th}}$ as effective as property-based tests at catching mutations. The effect size, calculated as Cramér's V, is 0.05 for a likelihood ratio chi-squared statistic. Since the effect size is less than 0.2, this is a weak association.

The two tests categorized as unit tests that killed the most mutations were, in fact, instances of property-based tests that did not fit into our definition of a PBT. These tests included no assertion statements because their assertion statements were located in functions that they invoked, so they were not considered PBTs by our tool. Of the remaining unit tests, the ones that were most effective

Table 2. Number of tests in test suite and mutations caught by project
Note: ML = Machine Learning, Lang. Tooling = Language Tooling, Vis. Tools = Visualization Tools, LOC = Lines of Python Code, Mut. Found = Number of Mutations Found

Project Name	Type	LOC	Total Tests	Percent PBT	Mut. Found by Unit Tests	Mut. Found by PBT	Total Mut. Found
buku	Applications	6610	1169	0.09%	20	0	20
Harmonbot	Applications	34882	107	37%	1	1	2
habitipy	Applications	2116	15	6.7%	12	10	16
hyperdome	Applications	5650	15	53%	8	7	9
anonlink	Cryptography	5649	9058	0.19%	21	51	63
argon2-cffi	Cryptography	2111	75	2.7%	55	8	55
clkhash	Cryptography	5718	141	8.5%	57	4	58
cryptoconditions	Cryptography	6353	425	1.2%	57	3	60
eth-keys	Cryptography	2625	62	15%	27	17	34
elm-ops-tooling	Lang. Tooling	1518	6	17%	27	13	34
hissp	Lang. Tooling	3139	53	9.4%	8	0	8
attrs	Lang. Tooling	18978	1330	2.0%	45	9	47
datatyping	Lang. Tooling	990	38	45%	15	50	52
dataclasses-json	Lang. Tooling	5142	328	0.30%	47	0	47
chocolate	ML	6117	128	2.3%	25	18	28
dsntnn	ML	927	36	2.8%	94	0	94
fklearn	ML	16115	172	0.58%	42	0	42
aiosmtplib	Networking	8235	358	2.2%	34	1	34
framewirc	Networking	2062	136	0.74%	37	8	37
h2	Networking	19048	1434	1.1%	37	8	38
hpack	Networking	7792	499	0.80%	9	29	32
dvrip	Networking	3960	125	54%	18	6	20
BIGSI	Science	3493	28	3.6%	3	0	3
cdflib	Science	10988	60	5%	31	5	32
forgi	Science	42094	561	0.71%	19	0	19
Ciw	Science	16740	330	6.1%	75	1	76
ExoData	Science	5935	182	0.55%	47	11	53
datacube-core	Science	55189	941	0.43%	0	0	0
fuzzywuzzy	Utilities	1675	71	2.8%	27	69	71
cyberpandas	Utilities	2484	105	0.95%	26	0	26
dpath-python	Utilities	2556	79	19%	52	13	53
hyperlink	Utilities	5554	115	7.0%	51	43	68
cloudformation-cli-python-plugin	Utilities	6884	159	5.7%	45	8	53
dc_stat_think	Utilities	3934	26	61.5%	22	32	41
chalice	Utilities	48486	1419	0.21%	62	0	62
chopsticks	Utilities	3852	38	2.6%	96	2	96
barva	Vis. Tools	531	2	50%	2	1	3
corrscope	Vis. Tools	14426	310	1.3%	11	1	11
geovoronoi	Vis. Tools	2140	43	7.0%	36	42	48
glue	Vis. Tools	49937	1393	0.07%	64	0	64

at finding mutations were explicitly written as integration tests, as reported by a comment in the test file. The next most effective unit test was a roundtrip test, which checked that a particular string, when split and put back together resulted in the same string. Such a test, if combined with an input generator, would be considered a property-based test.

Effects of code coverage. Since our projects varied in the percentage of code their test suites covered, we conducted a nominal logistic regression to compare the effects of a test being a unit test or a PBT and with respect to a project's code coverage percentage. We obtained statistically significant result ($\chi^2(2, N = 1916905) = 5479.835, p < .0001$) with an effect size of 0.05 and property-based tests having an odds ratio of 51.91, showing that a PBT is 52 times as likely to catch a mutation than a unit test. Our regression equation is $P(killed) = \exp((-2.34)+(1.97)(p)+(-0.02)(c))$, where p is either 1 (if the test is a PBT) or 0, and c is the project's code coverage percentage. This shows that a project's code coverage percentage has little effect on a test's ability to catch a mutation.

Categories of property-based tests. We obtained statistically significant results for all categories except generated-expression bounds checking, generated-expression non-equality, and constant inclusion. This indicates that there is a relationship between a test being in one of the other eight categories and the likelihood of catching a mutation. The category with the highest odds ratio is exception raising, at 112.76, followed by inclusion (36.42), and type checking (19.44). The odds ratio represents how much more likely a test of a particular category will catch a mutation than a test that is not a part of that category. Thus, exception raising tests are 113 times more likely to catch mutations than any other kind of test, including unit tests. Logistic regression can have greater bias when working with rare events, which may have influenced the results since there are few exception raising tests [26]. The effect sizes for these top three categories are 0.023, 0.017, and 0.011, respectively. Since they are all less than 0.2, the effects are small, so there is a weak association between test category and likelihood of catching a mutation. The chi-squared statistic, odds ratio, and effect size for each category are listed in Table 4.

Table 3. Contingency table for Unit Test vs. Property-Based Test
Note: Each test function is run on every mutation. For example, in the top right cell, 0.42% of all mutation tests involved a PBT. In the top left cell, 0.37% of *all* mutation tests resulted in a PBT function not catching a mutation, or 88% of those involving PBTs.

Test Type	Percent of mutation tests where a function did not catch a mutation	Percent of mutation tests where a function caught a mutation	Σ
PBT	0.37%	0.05%	0.42%
Unit	99.34%	0.24%	99.58%
Σ	99.71%	0.29%	100%

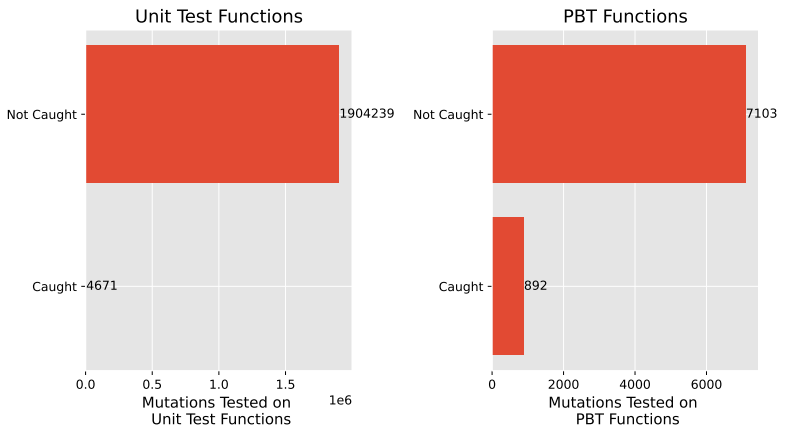


Fig. 21. Distribution of mutations tested on unit tests and PBT functions.

Table 4. Mutation testing statistics within PBT categories.
Note: GE = Generated-Expression

Category Type	Category	χ^2 (df=1)	Odds Ratio	Effect Size (Cramér's V)	p-value
Test of Equality	Roundtrip	359.458	13.02	0.014	**<0.0001**
	Partial Roundtrip	714.900	13.55	0.019	**<0.0001**
	Commutative Paths	1506.723	14.60	0.028	**<0.0001**
	Constant Equality	366.958	10.34	0.014	**<0.0001**
Test of Inequality	GE Bounds Checking	3.464	0	0.0013	.0627
	Constant Bounds Checking	164.745	12.46	0.0093	**<0.0001**
	GE Non-Equality	0.949	0	0.00070	0.3299
	Constant Non-Equality	21.298	6.12	0.0033	**<0.0001**
Other	Type Checking	234.785	19.44	0.011	**<0.0001**
	Inclusion	559.445	36.42	0.017	**<0.0001**
	Constant Inclusion	0.017	1.14	0.00009	0.8969
	Exception	1002.539	112.76	0.023	**<0.0001**

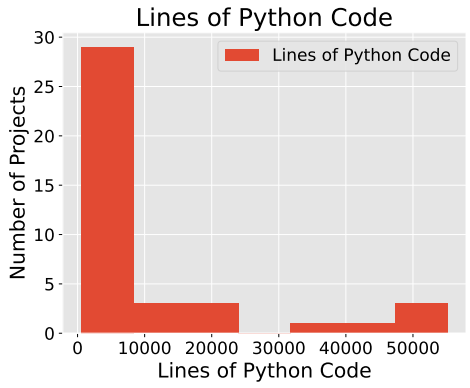


Fig. 22. Histogram showing the total lines of code in the tested projects

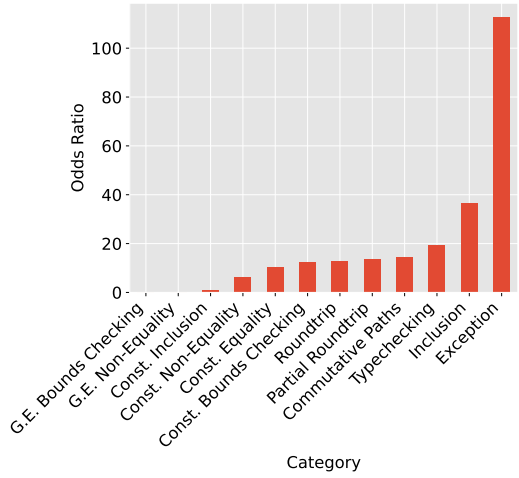


Fig. 23. Odds ratios for PBT categories.

RQ 2 Result: Unit tests comprised 98.4% of our sample and killed 84.7% of the total mutations found, while PBTs (comprising 1.6%) killed 15.3% of mutations. On a per-test basis, property-based tests were more effective at catching mutations than unit tests. Exception raising, inclusion, and type checking were the most effective of the property-based test categories.

5.2 Parameter Sweep Tests

To answer RQ3, we conducted an additional round of mutation testing to record how many inputs are needed for a property-based test to find a mutation.

We tested 100 mutations per project, but increased the default number of randomized inputs to 500 instead of Hypothesis's default 100. Two of the projects, `forgi` and `chopsticks`, had tests that consistently timed out on more than 100 inputs, so we do not include the projects in this experiment. Since a mutation can be found by multiple tests, we report the minimum input number needed to find a mutation. Figure 24 shows that the majority of mutations are found within 20 inputs to a PBT, and the number of mutations found starts to slowly level off after 100 inputs. In particular, 55% of mutations were found with a single input, and 76% within 20. Hypothesis's default number of inputs is 100, and 86% of mutations were found in this range. We also calculated the average amount of time taken to run tests of each category on 100 inputs and report them in Table 5.

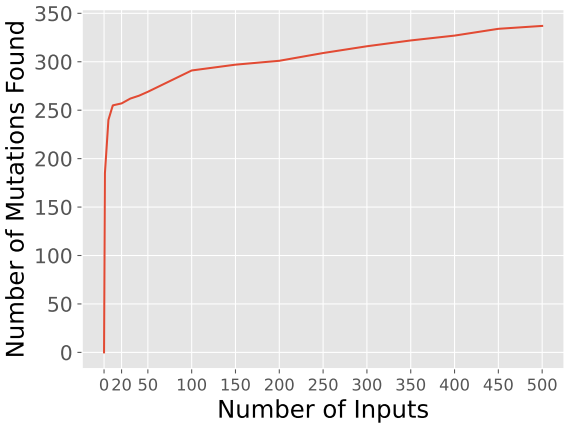


Fig. 24. Number of Inputs Needed to Find Mutations

Table 5. The average amount of time it takes to run a PBT function with 100 inputs containing a test of a specific category. Note: GE = Generated Expression

Category	Average Time (s)
Type Checking	2.519
Constant Equality	1.085
GE Bounds Checking	0.620
GE Non-Equality	0.519
Constant Non-Equality	0.511
Constant Bounds Checking	0.391
Inclusion	0.387
Commutative Paths	0.372
Roundtrip	0.283
Partial Roundtrip	0.262
Constant Inclusion	0.127
Exception Raising	0.083

RQ 3 Result: Over half of the mutations (55%) were found with a single input to a PBT. 76% of mutations were found within the first 20 inputs to a PBT function, 86% within the first 100 inputs, and 96% within the first 350 inputs.

6 Discussion

Prioritizing development time on writing the most effective tests is important for development efficiency; in addition to the time required to write tests, developers must also maintain tests so they continue to work as the code is updated. Writing only the most effective tests is also important for reducing test execution time and energy cost, and since property-based tests can generate thousands of inputs, they are more computationally expensive than unit tests. By primarily using property types that are the most effective at catching bugs, developers can reduce the burden of property-based tests on their development pipelines.

Among property-based tests, we found that exception raising, inclusion, and type checking tests are the most effective, with exception raising tests being 113 times more likely to catch mutations. Yet, these tests only make up 10% in total of the tests categorized in our corpus. This suggests that there may be a significant opportunity to help developers focus their test creation work on tests that are more likely to catch more bugs in cases where multiple kinds of properties are relevant to the code that needs to be tested. While our work does not assess the amount of human effort needed to write a PBT compared to a unit test, the large potential benefits of PBTs in bug-finding may motivate future work in this area.

In *How to Specify It!*, Hughes conducts a small-scale study of which properties find the most bugs, and concluded that tests that check a function's postconditions are the most effective [20]. Thus, encouraging the use of property-based tests with properties based on function preconditions and postconditions may help users write better tests. This is consistent with our finding that exception raising tests, which check that a particular exception is raised when executing a function with certain arguments, are particularly effective. These tests are in sense a test of preconditions, since often exceptions indicate that a function's preconditions have been violated. However, using exception raising as a *property* requires specifying the generator to only produce invalid inputs. For such a test to be effective, it would need to produce a wide variety of invalid inputs, and ideally, use the negation of the function's precondition to constrain the generator. This could present an opportunity for developers of generators, which could leverage informal specifications to create potentially-precondition-violating inputs. Likewise, an analysis could identify relevant properties by considering which exceptions code could raise.

Inclusion tests were the next most effective category of test. These tests check that a generated expression is in some collection, constraining the possible values the generated expression can take. This is akin to stating that a generated value can take on one of n possible values, where n is the size of the collection. These may be particularly useful to test whether data structure properties are invariant under various operations or as a more flexible kind of equality check that is applicable to a broad set of inputs. For example, the maximum element of a list must itself be an element of the list, so one can apply a generic list generator to create test cases.

Type checking tests were the third-most effective at finding mutations in Python programs. Since mutations mimic real-world bugs [25], the effectiveness of type checking tests could indicate that adding type annotations to Python programs can help reduce the amount of bugs in Python programs. Constant equality tests were the most common form of PBT in our corpus, but they are one of the least effective kinds of property-based tests. Using constants in a property-based test is one method of checking invariants: if a property involving a constant does not fail, then it is true for all valid program inputs. However, such tests were not particularly successful at catching mutations. This may be because the kinds of invariants represented by constant equality tests may be too weak to detect subtle bugs in programs.

The parameter sweep experiments showed that of the mutations found by property-based tests, 55% of them were found with one input, making the PBT effectively act like a unit test. What,

then, is the benefit of a property-based test over unit tests? Goldstein et al. [15] find that the increased *confidence* PBTs provide developers may play a part. Framing correctness in terms of properties instead of specific examples may help programmers gain a better understanding of their code. One of the most effective unit tests was one that checked a specific instance of a roundtrip property holds, which is an example-based test checking a particular property. Tools that help developers determine properties of their code may be useful regardless of whether they choose to use property-based testing or not.

7 Related Work

Studies of property-based testing practices in Python. Corgozinho et al. analyzed a sample of 86 property-based tests written using Hypothesis to find the most common properties checked in property-based testing and commonly used features of Hypothesis [9]. To categorize properties, they used a set of ten property definitions recommended either by a blog post cited in the official Hypothesis documentation [45] or a paper by one of the developers of Hypothesis [18]. Through manual analysis, Corgozinho et al. categorized tests using some of the more subjective properties such as test oracles and metamorphic properties, which compare how variations in input affect variations of output. After manually analyzing their collected tests, they found that 70.9% of property-based tests were either roundtrip tests (37.2%) or test oracles (33.7%). The authors also highlighted some other patterns, such as type checking and exception raising, which we also found in our corpus. In contrast to the 37.2% roundtrip tests found by Corgozinho et al, only 13.84% of tests in our study were considered some form of roundtrip test. In addition, our work evaluates the efficacy of PBTs written in Python and provides formal definitions for our categories, making our classification reproducible.

Looking at 28 projects, Wauters and De Rooter [44] investigate how users write property-based tests in Hypothesis for machine learning projects. They look at what kind of machine learning code PBT is used to test and the kinds of generators used. The authors found that property-based testing focused on conventional, non-machine learning specific code, and 20 out of 28 projects used custom complex generators for input data. Unlike our approach, theirs does not compare efficacy of property-based tests to that of unit tests.

In their work, Goldstein et al. recruited seven professional developers with experience using Hypothesis for an interview study [16]. Through thematic analysis, they identified five challenges developers face when using property-based testing, some of which include difficulties with determining properties to test, writing random generators, and integrating property-based tests into programmer workflows. In particular, the authors suggested that the choice of properties written by some of the participants may inadvertently undermine the power of their test. Consistent with this hypothesis, we found that different classes of property-based tests have different rates of finding bugs. Building off this previous study, the same authors [15] spoke to experienced users of property-based testing in OCaml and found that they primarily used PBT in "high-leverage" situations. Our work identifies categories of effective property-based tests, which can be used to identify potentially high-leverage situations where PBT will be useful for finding bugs.

Tools assessing the effectiveness of property-based tests. Etna [42] is a platform for comparing different property-based testing frameworks and evaluating the effectiveness of the generators written with these libraries. It adds manually constructed mutations to the generator's code and checks if the property in question will fail. At the time of publication, Etna supports PBT frameworks in Haskell and Coq. Tyche [17] is a tool to help developers understand the effectiveness of their property-based test. The tool focuses on representing the sample of inputs used when testing in terms of distribution and code coverage. Both tools assess the effectiveness of custom generators to test the properties used in the test, however, they do not evaluate the effectiveness of the test at

finding bugs in programs. Our study complements these tools by providing data on how property choice affects test suite effectiveness.

Studies assessing the effectiveness of test suites. The three most popular methods for assessing the effectiveness of test suites are code coverage [5, 14, 29, 41], mutation testing [5, 29, 40], and assessing tests on programs with known bugs [41]. Inozemtseva and Holmes found that code coverage and test suite effectiveness are not highly correlated when test suite size is controlled for [22]. Thus, we chose not to use test coverage as our metric for effectiveness. We did conduct a statistical test to determine the effectiveness of property-based tests compared to unit tests when code coverage is controlled, and we found that property-based tests are still around 50 times more effective than unit tests. To maximize external validity, we conducted our experiment on a variety of Python programs from existing GitHub projects. Thus, we did not conduct our experiment on Python programs with known bugs.

8 Future Work and Conclusion

It could be valuable to improve on our categorization of tests. One method could be to use a more nuanced definition of dataflow (perhaps an interprocedural analysis) that uses a lightweight analysis of called functions to assess whether the arguments are mutated in the function bodies. Another is to incorporate non-structural patterns that span multiple assertions. For instance, invariants can be checked by running one assertion, exercising the unit under test, then running the assertion again to ensure that it has the same outcome.

Property-based tests written in Python may differ from those written in functional or statically typed languages like Haskell and Rust; it could be valuable to study these and assess whether the results are consistent with ours. Another approach would be to categorize tests based on the kinds of generators used, as PBT libraries provide basic constructs but allow users to create their own custom generators for their tests. Wauters and De Rooter [44] investigated whether property-based tests in machine learning projects use custom generators, but there has not been a study of how custom generators are used in other kinds of Python programs.

Our study shows specific property classes that may have particularly high benefit, and that property-based tests are individually more effective at finding bugs than individual unit tests, but further work is required to assess the *cost* of writing various kinds of tests; our analysis is limited to only the benefits. Finally, we demonstrated that a majority of mutations are found with the first few inputs to a property-based test. This finding could motivate further work studying how the number of inputs relates to bugs found in the real world.

In conclusion, we identified 12 categories of property-based tests in Python, and found that property-based tests are 50 times more effective at finding bugs than unit tests. Among property-based tests, those in the exception raising, inclusion, and type checking categories are over 14 times more effective at catching bugs than other kinds of property-based tests. Our parameter sweep experiment, which showed that fewer than 20 inputs are needed to find 76% of bugs, can inform developers in trading bug detection power off for test completion speed. These findings motivate broader adoption of property-based tests and may help researchers build better tooling for property-based testing in Python and other languages.

9 Data Availability Statement

We provide all code and data used in our results in our artifact [1, 2]. Our data collection has three parts: (1) a script that finds projects importing the Hypothesis library using the 2022 Boa dataset; (2) a script that retrieves project files from GitHub at the current commit along with the commit hashes of the versions we used; and (3) a script that builds the test set of projects using the GitHub API, for which we provide both the dataset and code. We also include the results of mutation testing.

A Appendix

A.1 Valid Expression Types

(1) identifier
(2) literal (string, number, etc.)
(3) collection (list, set, or dictionary)
(4) comprehension (used with a collection)
(5) attribute (e.g., `a.x`)
(6) subscript or slice (e.g., `a[0]` or `a[1:3]`)
(7) function call
(8) arithmetic expressions
(9) boolean expressions
(10) comparisons
(11) membership test operation expressions
(12) bitwise operations expressions
(13) conditional expressions

Fig. 25. Valid expression types for assertion subexpressions

A.2 Definition of G(e)

$\boxed{G(e)}$

$$\frac{n \in \text{idents}(\text{Exprs}_{<a}) \qquad i : \texttt{int}}{G(n[i]) = G(n)}\text{Subscript} \qquad \frac{e_1, ..., e_k \in \text{Exprs}_{<a}}{G(f(e_1, ..., e_k)) = \bigcup_{k=1}^{m} G(e_k)}\text{Call}$$

$$\frac{n, n' \in \text{idents}(\text{Exprs}_{<a}) \qquad G(n) \neq \emptyset}{G(n') = \{n\} \cup G(n)}\text{GenIdentifier}$$

$$\frac{n, n' \in \text{idents}(\text{Exprs}_{<a}) \qquad G(n) = \emptyset}{G(n') = G(n)}\text{NonGenIdentifier} \qquad \frac{}{G(\texttt{atom}) = \{\}}\text{Literal}$$

$$\frac{f : \texttt{function} \qquad e' \in \text{Exprs}_{<a}}{G([f(x)\ \texttt{for}\ x\ \texttt{in}\ e']) = G(e')}\text{Comprehension}$$

$$\frac{e_1, ..., e_m \in \text{Exprs}_{<a}}{G(Collection(e_1, ..., e_m)) = \bigcup_{k=1}^{m} G(e_k)}\text{Collection}$$

$$\frac{n \in \text{idents}(\text{Exprs}_{<a}) \qquad x \in \texttt{attr}(n)}{G(n.x) = G(n)}\text{Attribute}$$

$$\frac{e_1, e_2 \in \text{Exprs}_{<a}}{G(e_1 + e_2) = G(e_1) \cup G(e_2)}\text{ArithmeticExpr} \qquad \frac{e_1, e_2 \in \text{Exprs}_{<a}}{G(e_1\ \texttt{or}\ e_2) = G(e_1) \cup G(e_2)}\text{BoolExpr}$$

$$\frac{e_1, e_2 \in \text{Exprs}_{<a}}{G(e_1 < e_2) = G(e_1) \cup G(e_2)}\text{Comparison} \qquad \frac{e_1, e_2 \in \text{Exprs}_{<a}}{G(e_1\ \texttt{in}\ e_2) = G(e_1) \cup G(e_2)}\text{MembershipTest}$$

$$\frac{e_1, e_2 \in \text{Exprs}_{<a}}{G(e_1\ \texttt{\&\&}\ e_2) = G(e_1) \cup G(e_2)}\text{BitwiseExpr}$$

$$\frac{e_1, e_2 \in \text{Exprs}_{<a}}{G(e_1\ \texttt{if}\ e_2\ \texttt{else}\ e_3) = G(e_1) \cup G(e_2) \cup G(e_3)}\text{CondExpr}$$

Fig. 26. The definition of G(e), which computes the variables from which an expression is derived.

$\boxed{D(\bar{s}; s_k)}$

$$\frac{(e_1, G(e_1)) \in D(\bar{s})}{D(\bar{s}; e_1 = e_2) = (D(\bar{s}) \backslash (e_1, G(e_1))) \cup (e_1, G(e_2))}\text{Assign}$$

$$\frac{(e, G(e)) \subset D(\bar{s})}{D(\bar{s}; \texttt{if } e \texttt{ then } \bar{s}_1 \texttt{ else } \bar{s}_2) = D(\bar{s}) \cup G(e) \cdot D(s_1) \cup G(e) \cdot D(\bar{s}_2)}\text{IfGen}$$

$$\frac{(e, G(e)) \not\subset D(\bar{s})}{D(\bar{s}; \texttt{if } e \texttt{ then } \bar{s}_1 \texttt{ else } \bar{s}_2) = D(\bar{s}) \cup D(\bar{s}_1) \cup D(\bar{s}_2)}\text{IfNonGen}$$

$$\frac{}{D(\bar{s}; \texttt{assert } e) = D(\bar{s})}\text{Assert} \qquad \frac{(e, G(e)) \subset D(\bar{s})}{D(\bar{s}; \texttt{while } e : \bar{s}') = D(s) \cup G(e) \cdot D(\bar{s}')}\text{WhileGen}$$

$$\frac{(e, G(e)) \not\subset D(\bar{s})}{D(\bar{s}; \texttt{while } e : \bar{s}') = D(\bar{s}) \cup D(\bar{s}')}\text{WhileNonGen}$$

$$\frac{(e_2, G(e_2)) \subset D(\bar{s}))}{D(\bar{s}; \texttt{for } e_1 \texttt{ in } e_2 : \bar{s}') = D(\bar{s}) \cup G(e_2) \cdot D(\bar{s}')}\text{ForGen}$$

$$\frac{(e_2, G(e_2)) \not\subset D(\bar{s}))}{D(\bar{s}; \texttt{for } e_1 \texttt{ in } e_2 : \bar{s}') = D(\bar{s}) \cup D(\bar{s}')}\text{ForNonGen}$$

Fig. 27. Rules for statements.

A.3 Project Descriptions

Table 6. Description and categorization of tested projects

Project Name	Description	Type	Lines
buku	Mini web browser for the terminal	Applications	6610
Harmonbot	Discord and Twitch bot	Applications	34882
habitipy	CLI for a habit tracking app	Applications	2116
hyperdome	Anonymous messaging application	Applications	5650
anonlink	Anonymous linkage w/ cryptographic linkage keys	Cryptography	5649
argon2-cffi	Password hashing utility	Cryptography	2111
clkhash	Cryptographic linkage key hashing library	Cryptography	5718
cryptoconditions	Library for cryptographic conditions and fulfillments	Cryptography	6353
eth-keys	API for Ethereum key operations	Cryptography	2625
elm-ops-tooling	Tooling for the Elm language	Language Tooling	1518
hissp	Lisp implementation compiling to Python	Language Tooling	3139
attrs	Utility for writing Python classes	Language Tooling	18978
datatyping	Typechecking for Python data structures	Language Tooling	990
dataclasses-json	API for encoding and decoding Python dataclasses to and from JSON	Language Tooling	5142
chocolate	Hyperparameter optimization framework	Machine Learning	6117
dsntnn	PyTorch implementation of a differentiable spatial to numerical layer of a neural network	Machine Learning	927
fklearn	Functional machine learning library	Machine Learning	16115
aiosmtplib	Asynchronous SMTP client	Networking	8235
framewirc	IRC message format framwork build with ansynchronous I/O	Networking	2062
h2	HTTP/2 protocol implementation	Networking	19048
hpack	HTTP2/ header encoding	Networking	7792
dvrip	Library and tools for the DVRIP communication protocol	Networking	3960
BIGSI	Searching utility for genomic data	Science	3493
cdflib	CDF file format reader and writer	Science	10988
forgi	Library for manipulating RNA secondary structures	Science	42094
Ciw	Simulation library for open queueing networks	Science	16740
ExoData	Exoplanet catalog interface and analysis tool	Science	5935
datacube-core	Analysis environment for satellite data	Science	55189
fuzzywuzzy	Fuzzy string matching library	Utilities	1675
cyberpandas	IP and MAC address datatype for Pandas	Utilities	2484
dpath-python	Library for accessing and searching dictionaries like a filesystem	Utilities	2556
hyperlink	Immutable URL implementation	Utilities	5554
cloudformation-cli-python-plugin	Python code generation tool for the Cloudformation Provider Development Toolkit	Utilities	6884
dc_stat_think	Utility functions used in a statistics course	Utilities	3934
chalice	Serverless microframework for AWS	Utilities	48486
chopsticks	Orchestration Library	Utilities	3852
barva	Audio visualizer	Visualization Tools	531
corrscope	Oscilloscope renderer for WAV files	Visualization Tools	14426
geovoronoi	Library for creating and plotting Voronoi diagrams with geographic data	Visualization Tools	2140
glue	Linked scientific dataset visualizer	Visualization Tools	49937

A.4 Correlations

Table 7. Correlations between property-based test categories

	Roundtrip	Partial Roundtrip	Comm. Paths	Const. Equality	G.E. Bounds	Const. Bounds	G.E. N.E.	Const. N.E.	Type Checking	Inclusion	Const. Inclusion	Exception Raising
Roundtrip	1.0	**0.33**	0.11	0.01	0.03	0.07	0.05	0.06	0.05	0.04	0.07	0.03
Partial Roundtrip	**0.33**	1.0	0.03	0.0	0.05	0.05	0.12	0.0	0.01	0.04	0.05	0.02
Comm. Paths	0.11	0.03	1.0	0.08	0.01	0.04	0.0	0.01	0.2	0.01	0.06	0.04
Constant Equality	0.01	0.0	0.08	1.0	0.01	0.12	0.05	0.02	**0.28**	0.04	0.12	0.04
G.E. Bounds	0.03	0.05	0.01	0.01	1.0	0.09	0.12	0.03	0.16	0.02	0.02	0.02
Constant Bounds	0.07	0.05	0.04	0.12	0.09	1.0	0.03	0.01	0.12	0.0	**0.63**	0.02
G.E. N.E.	0.05	0.12	0.0	0.05	0.12	0.03	1.0	0.02	0.02	0.01	0.03	0.01
Constant N.E.	0.06	0.0	0.01	0.02	0.03	0.01	0.02	1.0	0.03	0.02	0.01	0.01
Type Checking	0.05	0.01	0.2	**0.28**	0.16	0.12	0.02	0.03	1.0	0.02	0.06	0.01
Inclusion	0.04	0.04	0.01	0.04	0.02	0.0	0.01	0.02	0.02	1.0	0.02	0.01
Constant Inclusion	0.07	0.05	0.06	0.12	0.02	**0.63**	0.03	0.01	0.06	0.02	1.0	0.02
Exception Raising	0.03	0.02	0.04	0.04	0.02	0.02	0.01	0.01	0.01	0.01	0.02	1.0

References

[1] 2024. *Code for mutation testing tool.* https://doi.org/10.5281/zenodo.16912461

[2] 2024. *Paper artifact.* https://doi.org/10.5281/zenodo.16921991

[3] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036

[4] Stefan Berghofer and Tobias Nipkow. 2004. Random Testing in Isabelle/HOL. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference (SEFM '04)*. IEEE Computer Society, USA, 230–239.

[5] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.

[6] San Diego Supercomputer Center. 2022. Triton Shared Computing Cluster. University of California, San Diego. Service. https://doi.org/10.57873/T34W2R

[7] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 237–249.

[8] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[9] Arthur Lisboa Corgozinho, Marco Tulio Valente, and Henrique Rocha. 2023. How Developers Implement Property-Based Tests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 380–384. https://doi.org/10.1109/ICSME58846.2023.00049

[10] Hypothesis Developers. [n. d.]. *Hypothesis.* https://hypothesis.works/

[11] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 422–431.

[12] George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes* 22, 4 (1997), 74–80.

[13] The Python Language Foundation. 2024. The Python Language Reference: Python Full Grammar Specification, version 3.13.0. https://docs.python.org/3/reference/grammar.html

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

[15] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 187, 13 pages. https://doi.org/10.1145/3597503.3639581

[16] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. 2022. Some Problems with Properties. In *Proc. Workshop on the Human Aspects of Types and Reasoning Assistants (HATRA)*.

[17] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. 2024. Tyche: Making Sense of PBT Effectiveness. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3654777.3676407

[18] Zac Hatfield-Dodds. 2020. Falsify your Software: validating scientific code with property-based testing.. In *19th Python in Science Conference (SCIPY 2019)*.

[19] Anders Hovmöller and Many Contributors. 2024. *mutmut.* https://mutmut.readthedocs.io/en/latest/

[20] John Hughes. 2019. How to Specify It! A Guide to Writing Properties of Pure Functions. In *Trends in Functional Programming: 20th International Symposium, TFP 2019, Vancouver, BC, Canada, June 12–14, 2019, Revised Selected Papers* (Vancouver, BC, Canada). Springer-Verlag, Berlin, Heidelberg, 58–83. https://doi.org/10.1007/978-3-030-47147-7_4

[21] SAS Institute Inc. 2024. *JMP.*

[22] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

[23] JetBrains. [n. d.]. Python Developers Survey 2023 Results, 2023. https://lp.jetbrains.com/python-developers-survey-2023/.

[24] jqwik team. 2024. *Jqwik.* https://jqwik.net

[25] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.

[26] Gary King and Langche Zeng. 2001. Logistic regression in rare events data. *Political analysis* 9, 2 (2001), 137–163.
[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
[28] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. 2023. Continuous fuzzing: a study of the effectiveness and scalability of fuzzing in CI/CD pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 25–32.
[29] Jeshua S Kracht, Jacob Z Petrovic, and Kristen R Walcott-Justice. 2014. Empirically evaluating the quality of automatically generated and manually written test suites. In *2014 14th International Conference on Quality Software*. IEEE, 256–265.
[30] Holger Krekel and pytest-dev team. 2015. *pytest*. https://docs.pytest.org/en/stable/index.html
[31] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating good generators for inductive relations. *Proc. ACM Program. Lang.* 2, POPL, Article 45 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158133
[32] Leonidas Lampropoulos and Benjamin C. Pierce. 2023. *QuickChick: Property-Based Testing in Coq*. https://softwarefoundations.cis.upenn.edu/qc-current/index.html
[33] David MacIver, Zac Hatfield-Dodds, and Many Contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (21 11 2019), 1891. https://doi.org/10.21105/joss.01891
[34] David R. MacIver. 2025. *Hypothesis Documentation*. https://hypothesis.readthedocs.io/en/latest/settings.html#hypothesis.settings.load_profile
[35] Russell Mull. 2021-02-01. *Effective Property-Based Testing*.
[36] Andrew C. Myers. 1999. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561
[37] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401.
[38] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 537–548. https://doi.org/10.1145/3180155.3180183
[39] Zoe Paraskevopoulou, Cătălin HriŢcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 325–343.
[40] Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. 2022. An empirical comparison of EvoSuite and DSpot for improving developer-written test suites with respect to mutation score. In *International Symposium on Search Based Software Engineering*. Springer, 19–34.
[41] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 201–211. https://doi.org/10.1109/ASE.2015.86
[42] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 218 (Aug. 2023), 17 pages. https://doi.org/10.1145/3607860
[43] TIOBE. [n. d.]. TIOBE Index. October 2024. https://www.tiobe.com/tiobe-index/
[44] Cindy Wauters and Coen De Roover. 2024. Property-based Testing within ML Projects: an Empirical Study. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.
[45] Scott Wlaschin. 2014-12-12. *Choosing properties for property-based testing*.
[46] W. Eric Wong and Aditya P. Mathur. 1995. Reducing the cost of mutation testing: an empirical study. *J. Syst. Softw.* 31, 3 (Dec. 1995), 185–196. https://doi.org/10.1016/0164-1212(94)00098-0