# Property-Based Testing in Open-Source Rust Projects

## A Case Study of the `proptest` Crate

**Antonios Barotsis**
**Supervisor(s): Dr. Andreea Costea, Sára Juhošová**
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 27, 2025

Name of the student: Antonios Barotsis
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Sára Juhošová, Marco Zuñiga Zamalloa

# 1 Introduction

Testing is a critical part of software development, since it helps developers ensure that their programs behave correctly. This is especially true for popular open source projects, because of how foundational they are to today's software and society as a whole [1].

Property-Based Testing (PBT) [2], a paradigm popularized by the Haskell framework QuickCheck [3] and later Python's Hypothesis [4], has emerged as a simple yet powerful new testing method. In traditional unit testing, software engineers manually write down input-output pairs which the testing framework then validates. In PBT, the engineers instead create formal specifications [5, p. 1] for the components they want to test. These are then validated by the testing framework against a great number (usually in the hundreds [3, p. 269], [6]) of randomly generated inputs. There are two unique concepts that PBT introduces: Generators and Shrinkers. Generators are responsible for creating the test inputs while shrinkers are responsible for finding a minimum reproducible example of any failing test cases.

PBT has also found success as a soft-proving technique; by carefully defining the properties a code unit should have, it makes catching even notoriously elusive bugs (e.g., race conditions [7], [8]) possible. Even though using PBT should not make anyone as confident as a proper proof of correctness or safety should, its relative simplicity compared to proofs makes it a very appealing, pragmatic candidate for real-world software where time is often limited [5].

While PBT as a paradigm has gained traction across various domains and programming ecosystems, and has been explored in previous, foundational research [9], [10], [11], few studies investigate how it gets used in real-world open-source projects. With this paper, we aim to give insights into the usage of PBT to both researchers and software engineers seeking to adopt or improve PBT practices.

In order to gather insights into the usage of PBT, this paper seeks to answer the following questions:

- Properties
  1) What type of properties do PBTs generally check?
  2) What do these properties look like?
  3) What role does PBT play within the correctness guarantees and bug-finding strategies of the project overall?
- Generators and Shrinking:
  1) How and when are generators implemented?
  2) In which cases is shrinking support explicitly added?

This work is part of a collaboration between the members of our research group. We all aim to answer the same research questions but in different programming languages or frameworks [12], [13], [14], [15]. This allows us to make interesting comparisons on the state of PBT across different ecosystems.

In our case, we chose to investigate the Rust programming language [16] and the `proptest` framework. Rust as a language has been gaining a lot in popularity recently. Seeing as how some of its main selling points are safety guarantees and the ability to catch many types of bugs at compile time, we consider investigating how Property-Based Testing fits in this ecosystem quite interesting. Rust has two prominent PBT frameworks, `proptest` and `quickcheck`. We decided to examine `proptest` in this paper.

# 2 Methodology

In this section we will detail how we chose and analyzed repositories as well as the Property-Based Tests found in them.

| Language | Files | Lines | Code | Comments | Blanks |
|---|---|---|---|---|---|
| JSON | 1 | 46 | 46 | 0 | 0 |
| TOML | 17 | 835 | 637 | 58 | 140 |
| Markdown | 19 | 6263 | 0 | 4595 | 1668 |
| ├ BASH | 1 | 17 | 9 | 4 | 4 |
| ├ Rust | 4 | 186 | 130 | 29 | 27 |
| ├ TOML | 2 | 4 | 4 | 0 | 0 |
| (Total) | | 6470 | 143 | 4628 | 1699 |
| Rust | 709 | 109884 | 84457 | 6261 | 19166 |
| ├ Markdown | 427 | 46552 | 1408 | 34219 | 10925 |
| (Total) | | 156436 | 85865 | 40480 | 30091 |
| Total | 746 | 163787 | 86691 | 45166 | 31930 |

*Table 1:* `tokei` *output of the* `tokio` *project.*

## 2.1 Finding Repositories

Our first step was to define what we consider an adequately impactful Open Source project to examine. We decided to use the following three metrics as an indicator (in this order):

1) total downloads,
2) recent downloads (in the last month), and
3) GitHub stars.

In practice, we used total downloads as our indicator and included all three as descriptors in our data. We chose to weigh total downloads higher than recent downloads purely for convenience. The Rust package registry - `crates.io`[1] - conveniently keeps track of every published project's dependencies. This allows us to easily perform a reverse search to find all projects that *depend* on `proptest` sorted by total downloads[2].

We considered Github stars to be less important for three reasons:

- We analyzed 16 repositories, all of which had tens of millions *recent* downloads but fewer than 500 stars. While downloads reflect substantial usage, GitHub stars often signify interest in contributing to or drawing inspiration from a project rather than measuring adoption. Thus, stars alone are a poor proxy for a library's real-world usage.
- It has been known for some time that Github stars can be purchased [17], [18], [19], which renders them unsuitable to be used for determining which projects to examine.
- They do not always make sense in the context of `Cargo` workspaces as multiple separate published crates are often uploaded as a single repository (see Section 2.1.1)

We nonetheless included them as an extra datapoint in order to remain consistent with the rest of the research group. In addition to the previously mentioned metrics, we also recorded the following metadata about each repository:

**Lines of code** There are a number of tools that automatically compute the number of lines of code automatically, we decided to use one called `tokei`[3]. See Table 1 for an example output. In the example, we are interested in the number `84457`, showing the total lines of *Rust* code.

**Total number of tests** Counting the amount of tests is easy as all Rust tests are marked with the `#[test]` macro. We used ripgrep[4] and the following expression to count them:

```
rg "#\\[test\\]" . --stats --quiet
```

**Total number of Property-Based Tests** Unfortunately we did not find an easy way to reliably count PBTs automatically as they look like regular Rust tests. We can however locate them

[1] https://crates.io
[2] https://crates.io/crates/proptest/reverse_dependencies
[3] https://github.com/XAMPPRocky/tokei
[4] https://github.com/BurntSushi/ripgrep

easily as they are always inside `proptest! {...}` blocks with the following regular expression:

```
rg proptest!
```

We then manually counted each test inside those blocks to get the amount of PBTs for each repository.

### 2.1.1 A Word on Rust Workspaces

Rust has a way of separating code into collections of one or more packages called *workspaces* [20]. In essence they function similarly to *namespaces* or *modules* from other programming languages. A key difference is that these packages are often published as separate units. Because of this, we end up with multiple packages under one single repository, akin to a *monorepo*[5].

As an example, when looking through `proptest`'s reverse dependencies we came across the following two packages: `anstream` and `anstyle-parse`. Both of these are part of the same workspace and repository `anstyle`. In these cases, instead of providing the number of Github stars, we instead provide a link to the *parent* repository. Similarly, instead of providing the *total* number of tests for the entire repository, we performed our searches *inside* each workspace member we were examining at the time.

## 2.2 Analyzing Individual Tests

We used qualitative data analysis techniques such as open coding [21] to analyze our tests. Our collected data is in the form of an Excel Sheet meant to both be human readable as well as programmatically parsable for easy further analysis A detailed explanation of what each of our 22 columns measures as well as what value types it contains can be found in our data archive [22].

### 2.2.1 Open Coding

We began by deciding that each member of our research group should perform an early, brief analysis in their respective ecosystem of choice and create labels for their data. We then combined and standardized our findings into a shared dictionary which was used to conduct analyses on all the considered ecosystems. Due to the differences between ecosystems, we all retained the freedom of adding new, non-shared codes that match our ecosystem but not everyone else's. This way, we kept our data as consistent as possible while not ignoring details specific to each language. We naturally iterated upon our labels and data structure as we discovered new patterns in our data worth keeping track of.

We arrived at the following codes for describing PBT types:

1) **DifferentPaths**: Tests multiple execution paths through the system (e.g., if/else branches, error handling). Verifies all logical routes behave correctly.
2) **RoundTrip**: Tests data transformations that should be reversible (encode/decode, serialize/deserialize).
3) **Invariant**: a property that must hold throughout the execution of a test,
4) **Idempotence**: Ensures repeating an operation yields the same result.
5) **StructuralInduction**: Tests recursive structures by verifying:
   a) base cases (smallest instance) and
   b) inductive step (if property holds for $n$, it holds for $n + 1$).
6) **HardToProveEasyToVerify**: Focuses on properties where correctness is complex to implement but simple to check.
7) **TestOracle**: Compares results against a known-correct reference implementation.

[5] https://en.wikipedia.org/wiki/Monorepo

#### 2.2.1.1 Example

Here is an example of what our analysis looks like in a PBT from the `nom`[6] crate (more detailed explanations can be found in our data archive [22]):

```
1  proptest! {
2    #[test]
3    #[cfg(feature = "std")]
4    fn floats(s in "\\PC*") {
5      println!("testing {}", s);
6      let res1 = parse_f64(&s);
7      let res2 = double::<_, ()>(s.as_str());
8      assert_eq!(res1, res2);
9    }
10 }
```

- Line 4: Uses a custom regex generator which filters on printable characters. Test has one input of type `String`.
- Line 6: Helper function that runs the standard library's parser behind the scenes. Serves as a TestOracle. Input is used directly.
- Line 7: The system under test. Input is used directly.
- Line 8: The single assertion

In addition we note things such as:

- The test does not use a custom shrinker.
- It asserts errors (`res1` and `res2` are `Result<Ok, Error>` monad types).
- This is a functional test.

This is a subset of the data we collected for each test, the rest are omitted from here to keep the section short. They are all explained in detail in our data archive [22].

# 3 Results

We first perform a general exploration of our dataset to get a feel of what our data looks like.

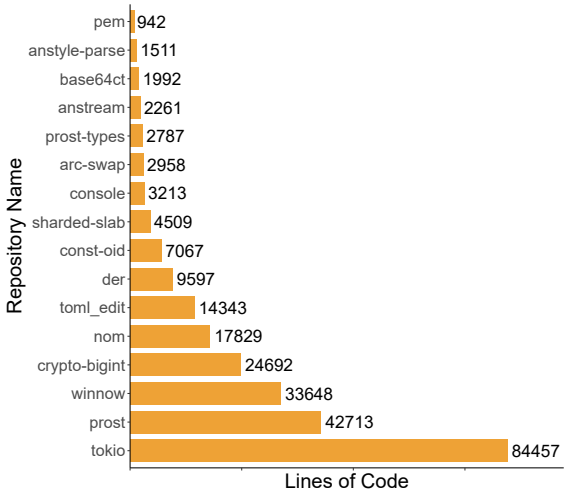


*Figure 1: Lines of code per repository*

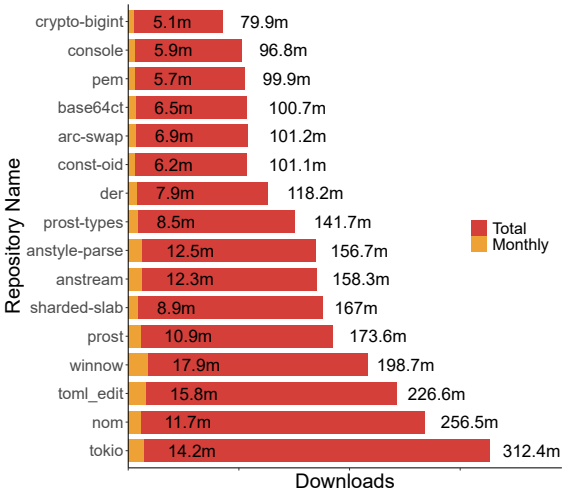


*Figure 2: Total and Monthly Downloads per Repository*

[6] https://archive.is/PxHWG

In Figure 1 we can see that the projects we examined showed significant variance in code size with `tokio` being considerably larger than the rest. In Figure 2 we can see that the amount of downloads were much closer together with `tokio` being the most popular in terms of total downloads. Monthly downloads were much more tightly packed together.

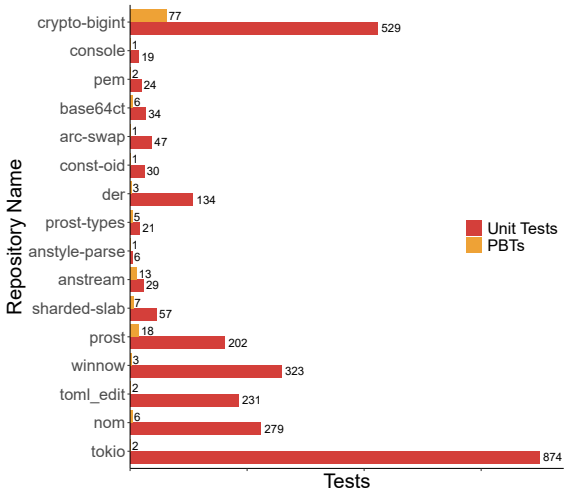


*Figure 3: Amount of Unit and PBTs per Repository*

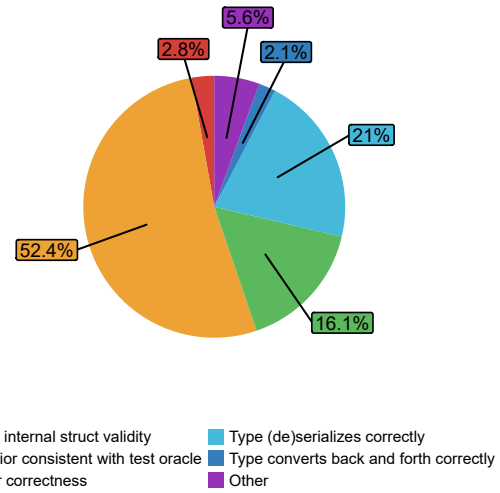


*Figure 4: PBT Category Breakdown*

There proved to be a surprising discrepancy between the number of unit and Property-Based tests as shown in Figure 3. `tokio` in particular was interesting, seeing as how it is a very well tested project but only used PBTs twice. `crypto-bigint` is a big outlier here as the number of PBTs it had was around half of all the PBTs from every other project we examined combined.

In Figure 4 We can see that the Test Oracle pattern is by far the most commonly used PBT type. We also notice that most other categories are related to some form of parsing/(de)serialization. In practice, a lot of the TestOracle tests are also testing parsers.

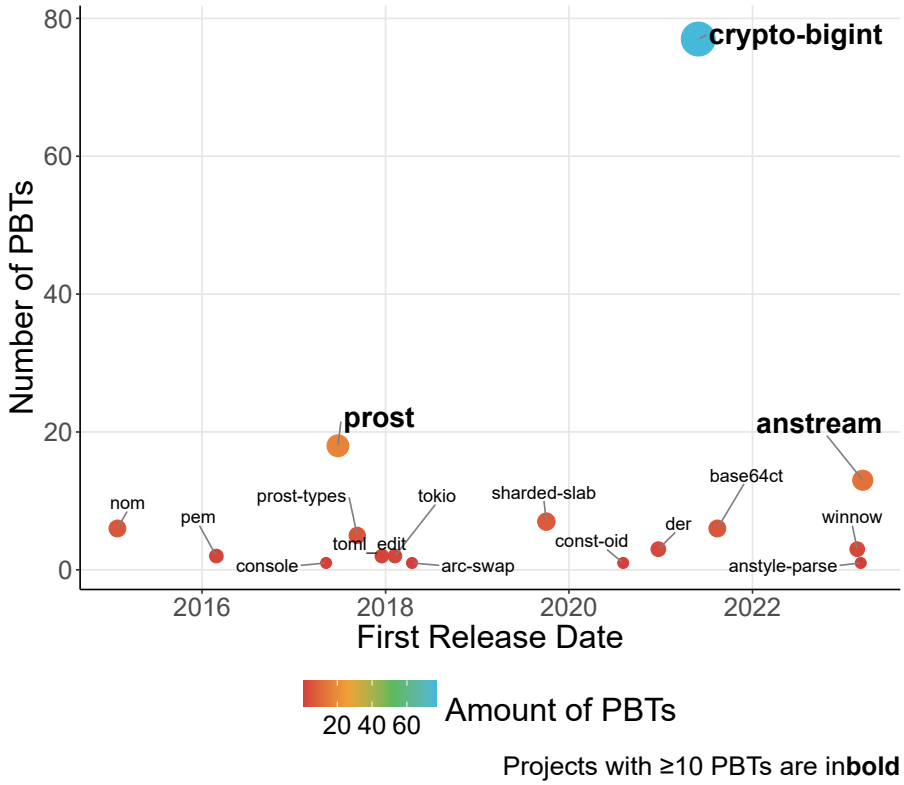


*Figure 5: First Release Date vs Amount of PBTs*

An interesting finding in Figure 5 is that crate maturity and amount of PBTs do not seem to correlate. We initially expected older projects to make more prominent use of PBT but it seems that this is not the case.

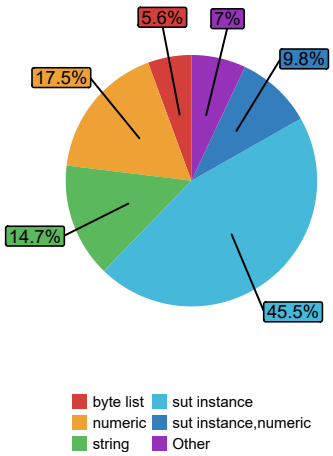


*Figure 6: PBT Input Type Breakdown. Commas in types indicate both being used together in one test.*

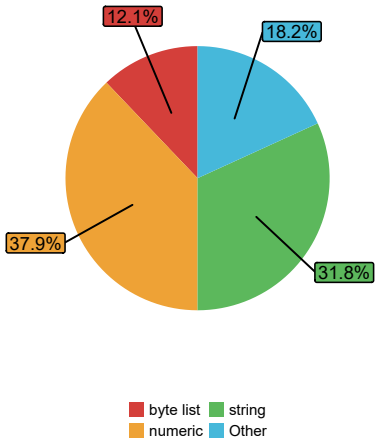


*Figure 7: PBT Input Type Breakdown, Excluding Outliers*

Finally, we take a look at the types on inputs produced by generators in Figure 6. While we see that `sut instance` is the most common one, removing the `crypto-bigint` outlier results in Figure 7 where we see that string and numeric primitives now constitute the majority instead.

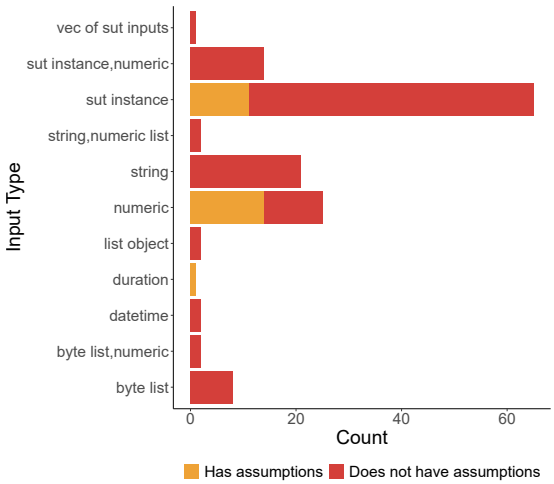


*Figure 8: Assumption Usage per Input Type*

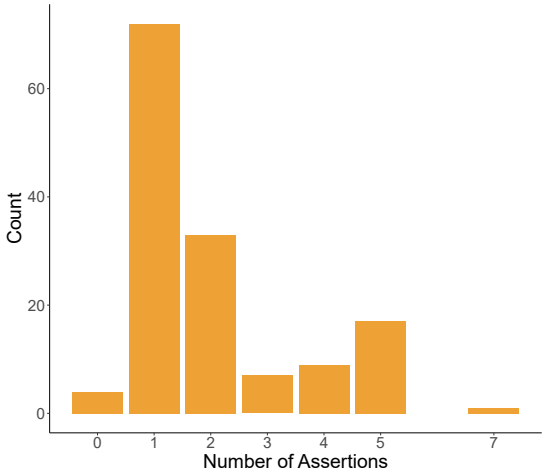


*Figure 9: Assertions per Test*

In Figure 8 we examine the different input types used in our PBTs and whether or not they made use of assumptions. We can see that `numeric` stands out for having an almost 50-50 split. This is rather surprising as initially we were expecting assumptions to be used for more complex data types, either to reject invalid or meaningless states or to help generate them correctly. Instead assumptions tend to be rather simple edge-case filtering. Lastly, in Figure 9 we can see the amount of assertions each test we examined had. Compared with some of the other ecosystems our group examined, this was rather low.

| repo | byte list | string | vec of sut inputs | byte list,numeric | sut instance | sut instance,numeric | numeric | string,numeric list | datetime | duration | list object |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **anstream** | 4 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **anstyle-parse** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **arc-swap** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **base64ct** | 3 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **console** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **const-oid** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **crypto-bigint** | 0 | 0 | 0 | 0 | 63 | 14 | 0 | 0 | 0 | 0 | 0 |
| **der** | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| **nom** | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **pem** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| **prost** | 0 | 0 | 0 | 0 | 0 | 0 | 13 | 0 | 0 | 0 | 0 |
| **prost-types** | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 2 | 1 | 0 |
| **sharded-slab** | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 2 |
| **tokio** | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| **toml_edit** | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **winnow** | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |

*Table 2: PBT Input Type Breakdown. Commas in types indicate both being used together in one test.*

The data presented in Table 2 is the same as with Figure 6 from earlier but from a different lens. The table makes it obvious that `sut instance` is almost exclusively used in the `crypto-bigint` crate. Additionally, we observe a limited input type variety within each repository with most repositories having just one or two different input types.

# 4 Discussion

## 4.1 PBT Complexity

Our tests were relatively simple compared to those of other ecosystems we examined. Only 13.3% of our tested properties can be decomposed. Looking at Figure 9, we can also see that most of our tests only had up to 2 assertions (in fact the mean is 2.01).

### 4.1.1 Why Are Our PBTs Simpler?

While we cannot make any definitive claims, we do have some unverified hypotheses which future work could examine further:

1) Rust has many specialized tools that help test for specific (and usually rather tricky!) bug types. Two of those that come to mind are `miri`[7] and `loom`[8] which test for undefined behavior (UB) and concurrency bugs respectively. We found that in some of the other ecosystems we explored, PBT was used to test for these. Since Rust does have these specialized frameworks, the complexity that comes with testing for UB and concurrency issues is moved away from PBTs.
2) Rust has a very concrete and descriptive type system. We found that in some other languages, PBTs were used to validate that variables had the expected types which is not a concern that you need to test for in *safe* Rust. When it comes to unsafe conversions, tools such as `miri` can be used as mentioned above.
3) Rust uses errors as values instead of exceptions.
4) `proptest` itself is both powerful and easy to use which helps with keeping the tests simpler. We found the generators and filtering specifically to be quite powerful as well as features such as shrinking support by default.
5) Lastly, when we began the project, we expected to find PBTs being used as documentation examples which turned out to *not* be the case, Rust has special natively supported `doctests`[9] that do exactly that. It could be the case that because the tests were therefore not meant to be user-facing, self-explanatory and contain everything there is to know about the system under test, they ended up being smaller and simpler.

## 4.2 Comparing to `quickcheck`

Unlike other language environments we examined, Rust happens to have two almost equally dominant PBT frameworks: `proptest` - *which this paper is examining* - and `quickcheck` [15]. Considering how unusual this is, we found it interesting to compare and contrast the two.

### 4.2.1 Shrinkers & Generators

We think that the usage of custom Generators and Shrinkers are linked to each other as expressive Generators can reduce the need for custom Shrinkers.

We were surprised to find *zero* custom shrinker implementations with `proptest` in all of the data we examined. We performed a code search using both SourceGraph Code Search - *a code search tool for large datasets* - looking for instances of custom shrinker implementations. We found custom shrinkers used in 20 `proptest` repositories and 116 `quickcheck` repositories. More details on how we arrived in these numbers can be found in Appendix A.

We suspect this to be caused by a couple of reasons:

1) As the `proptest` documentation [23] explains, in contrast to `quickcheck`, re-using existing generators is common due to how easily one can map them from one type to another.
2) `proptest` makes generating constrained inputs easy to achieve without the need for custom generators. This is achieved by defining special syntax that can be used in the generator definitions (as an example: `num in 0..42`).
3) `quickcheck` does *not* shrink by default unlike `proptest`. This results in a significant amount of the custom shrinkers in `quickcheck` to be rather trivial to implement by hand and would be implemented automatically by `proptest`.

[7] https://github.com/rust-lang/miri
[8] https://github.com/tokio-rs/loom
[9] https://doc.rust-lang.org/rustdoc/write-documentation/documentation-tests.html

An interesting thing to note is that the author of `quickcheck` believes that "`proptest` improves on the concept of shrinking" [24].

As expected, because `proptest` leans heavily on generators, most (74.1%) of the projects we examined used some form of custom generator.

#### 4.2.2 Assumptions

Assumptions are another interesting topic. Normally we define assumptions as conditions or constraints that are expected to hold true for the properties being tested. These are typically the preconditions that must be satisfied for the properties to be valid. The difference between assumptions and filtering is simple:

- Filtering happens at the input generation phase. PBT frameworks are configured to generate $n$ inputs per test and once input generation is done, the test is guaranteed to have $n$ inputs.

  In practice there are checks in place to ensure test generation does not go on forever, in which case a test might end up with less than the expected $n$ inputs but that is more of an implementation detail. On paper there should be $n$ inputs.
- Assumptions take place *during* the test execution which means that some subset of those $n$ test inputs may be ignored. This can be often dangerous as the test might execute significantly less times than one would expect.

An unorthodox implementation detail of `proptest` is that the built-in assumptions actually function like filtering: they attempt to regenerate the rejected test input. There are two ways of making assumptions:

1) By returning `TestCaseError::Reject` from a test,
2) By invoking the `prop_assume!` macro (which itself returns `TestCaseError::Reject`)

Both of these function as filters, in order to make a "true" assumption, `if` statements need to be used instead.

A surprising difference with `quickcheck` is that it does not support filtering in the way we defined it earlier. As a result, it makes significantly higher use of assumptions than `proptest`.

#### 4.2.3 What Types of Projects Use Each Framework?

That said, we anecdotally found `quickcheck` to be predominantly used in low-level projects (such as custom allocators, hashmaps, checksum implementations and more) while `proptest` was very commonly used in various, higher level projects and specifically (de)serializer implementations.

#### 4.2.4 Amount of PBTs

We were surprised to find that projects that used `quickcheck` had significantly more PBTs than projects that used `proptest`. The reasons for this phenomenon are unclear, however, we have formulated several hypotheses:

1) `quickcheck`'s generation is significantly *quicker*, by an order of magnitude [23]. This seems important, however, we anecdotally inspected the Continuous Integration (CI) runs of the `crypto-bigint`[10] crate (which had by far the most PBTs in our data set at around 70) and were surprised to find them surprisingly short.

   Tests took anywhere between 20 and 70 seconds to both compile and execute *all* tests in the project, not just PBTs. Considering `crypto-bigint` had around 500 tests total, we consider this to be reasonable. We can see that even though `proptest` is on paper a lot slower, it would take very extensive usage for it to become a bottleneck.

[10] https://github.com/RustCrypto/crypto-bigint

## 4.3 Answers to the Research Questions

1) **RQ1**: What type of properties do PBTs generally check?
   As we can see in Figure 4, the most common property type is TestOracle where results are compared against known, correct reference implementations. This is attributed to the prevalence of parsing/(de)serialization crates in our data.
2) **RQ2**: What do these properties look like?
   Simple and focused: Most (86.7%) tests are *not* decomposable and have two assertions on average. In Figure 7 we see that most input types are either `String`s or `numeric`.
3) **RQ3**: What role does PBT play within the correctness guarantees and bug-finding strategies of the project overall?
   - **Complementary to Rust's safety tools**: PBT focuses on logic errors (e.g., parser edge cases), while tools like `miri` (undefined behavior) and `loom` (concurrency) handle specialized bug classes (Section 4.1.1).
   - **Less critical for type safety**: Rust's strong static typing reduces the need to test type-related properties common in dynamically typed languages.
   - **Not documentation**: Unlike doctests, PBTs are internal checks and not designed as user-facing examples.
4) **RQ4**: How and when are generators implemented?
   - **Custom generators are common**: 74.1%of projects implement them (Section 4.2.1).
   - **Implementation**: Defined via `proptest!` blocks using:
     - Built-in syntax: Constraints (e.g., `num in 0..42`) simplify common cases.
     - Mapping: Reusing/composing generators via `.prop_map` (Section 4.2.1).
   - **When used**: For constrained inputs (e.g., regex-filtered strings) or complex types
5) **RQ5**: In which cases is shrinking support explicitly added?
   Zero custom shrinkers were found in the analyzed projects. Possible reasons for that include:
   - `proptest`'s design: Built-in shrinking is automatic and composable via generators.
   - Filtering over assumptions: `prop_assume!` regenerates inputs (acting like a generator filter), reducing the need for manual shrinking (Section 4.2.2).

## 4.4 Future Work

In Figure 5 we noticed that crate maturity does not seem to correlate with the amount of PBTs. Perhaps it would be interesting to examine when in each project's development history PBT was introduced. Did the projects consider PBT from the beginning or was it more of an afterthought? We unfortunately did not have the time to look into this any further.

Due to how time consuming the analysis of our data was, we believe that there is a lot of value to be gained in investigating whether it can be automated through the use of Machine Learning. This would allow us to get insights on a lot more repositories and tests, leading to more statistically significant insights.

## 4.5 Threats to Validity

- It is important to remember we only had time to examine very few projects compared to the total amount of repositories that make use of each of our respective frameworks. This means that any patterns we encountered could be drastically different if our input sizes were sizeably larger.
- We chose repositories based on their download counts. This prioritizes high-usage projects while niche or emerging crates could be overlooked. It is possible that those projects use PBT differently.
- Our data represents a snapshot of projects, however, PBT usage patterns may evolve over time.

# Appendix A: Querying for Custom Shrinkers

We archived our search results from SourceGraph for `proptest`[11] and `quickcheck`[12]

Our queries were as follows:

SourceGraph Query

```
1 lang:Rust
2 proptest::strategy AND
3 ValueTree AND
4 "fn simplify"
5 not repo:^github\.com/proptest-rs/proptest$
6 count:all
```

They should be rather self explanatory but very briefly:

1) The Trait we are looking for is `proptest::strategy::ValueTree`.
    a) We break this down to importing `proptest::strategy` and `ValueTree`. We keep them separate in case the import is of the form `proptest::strategy::{..., ValueTree}`.
    b) We additionally search for the `simplify` function, this is what actually defines the shrinker.
2) We filter for only Rust repositories.
3) We exclude the `proptest` repository from our search.

The exact numbers are meant to be indicative as they have not been manually reviewed to get rid of false positives (or find missing false negatives).

Lastly, for the sake of completeness, this is the query we used for `quickcheck`:

SourceGraph Query

```
1 lang:Rust
2 quickcheck AND
3 Arbitrary AND
4 "fn shrink"
5 not repo:^github\.com/BurntSushi/quickcheck$
6 count:all
```

[11] https://archive.is/UB3q8
[12] https://archive.is/j9ERk

# References


[1] M. Hoffmann, F. Nagle, and Y. Zhou, "The Value of Open Source Software," *SSRN Electronic Journal*, 2024, doi: 10.2139/ssrn.4693148.

[2] "What is Property Based Testing? - Hypothesis." Accessed: Apr. 26, 2025. [Online]. Available: https://archive.is/nJa5R

[3] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, Sep. 2000, doi: 10.1145/357766.351266.

[4] D. MacIver, Z. Hatfield-Dodds, and M. Contributors, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, Nov. 2019, doi: 10.21105/joss.01891.

[5] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head, "Property-Based Testing in Practice," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon Portugal: ACM, Apr. 2024, pp. 1–13. doi: 10.1145/3597503.3639581.

[6] "Config in proptest::test_runner - Rust." Accessed: May 08, 2025. [Online]. Available: https://archive.is/DJrnB

[7] K. Claessen *et al.*, "Finding race conditions in Erlang with QuickCheck and PULSE," *SIGPLAN Not.*, vol. 44, no. 9, pp. 149–160, Aug. 2009, doi: 10.1145/1631687.1596574.

[8] J. Hughes, "QuickCheck Testing for Fun and Profit," *Practical Aspects of Declarative Languages*, vol. 4354. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–32, 2006. doi: 10.1007/978-3-540-69611-7_1.

[9] H. Li, S. Thompson, P. Lamela Seijas, and M. A. Francisco, "Automating property-based testing of evolving web services," in *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, San Diego California USA: ACM, Jan. 2014, pp. 169–180. doi: 10.1145/2543728.2543741.

[10] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with quviq QuickCheck," in *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, Portland Oregon USA: ACM, Sep. 2006, pp. 2–10. doi: 10.1145/1159789.1159792.

[11] Y. Zhang, P. Li, Y. Ding, L. Wang, D. Williams, and N. Meng, "Broadly Enabling KLEE to Effortlessly Find Unrecoverable Errors in Rust," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, Lisbon Portugal: ACM, Apr. 2024, pp. 441–451. doi: 10.1145/3639477.3639714.

[12] H. Toth, "Property-Based Testing in the Wild!: Exploring Property-Based Testing in Java: An Analysis of jqwik Usage in Open-Source Repositories," 2025.

[13] Y. Zhao, "Property-Based Testing in the Wild!: A Study of QuickCheck Usage in Open-Source Haskell Repositories," 2025.

[14] D. de Koning, "Property-Based Testing in Practice using Hypothesis: In-depth study on how developers use Property-Based Testing in Python using Hypothesis."

[15] M. Derbenwick, "Property-Based Testing in Rust, How is it Used?: A case study of the quickcheck crate used in open source repositories."

[16] "The Rust Programming Language - The Rust Programming Language." Accessed: Jun. 03, 2025. [Online]. Available: https://doc.rust-lang.org/stable/book/

[17] H. He, H. Yang, P. Burckhardt, A. Kapravelos, B. Vasilescu, and C. Kästner, “4.5 Million (Suspected) Fake Stars in GitHub: A Growing Spiral of Popularity Contests, Scams, and Malware.” [Online]. Available: https://arxiv.org/abs/2412.13459

[18] “How Much Are GitHub Stars Worth to You? – The Guild.” Accessed: May 13, 2025. [Online]. Available: https://archive.is/TPXeq

[19] “Over 3.1 million fake "stars" on GitHub projects used to boost rankin….” Accessed: May 13, 2025. [Online]. Available: https://archive.is/6kyav

[20] “Workspaces - The Cargo Book.” Accessed: May 14, 2025. [Online]. Available: https://archive.is/bPihy

[21] R. Hoda, *Qualitative Research with Socio-Technical Grounded Theory*. 2024. doi: 10.1007/978-3-031-60533-8.

[22] M. Derbenwick *et al.*, “Property-Based Testing in the Wild!.” 2025. doi: 10.4121/368f63ab-10fc-4603-a15a-bde25e72e778.

[23] “Proptest vs Quickcheck - Proptest.” Accessed: Jun. 03, 2025. [Online]. Available: https://archive.is/0vLuK

[24] “GitHub - BurntSushi/quickcheck: Automated property based testing for ….” Accessed: Jun. 04, 2025. [Online]. Available: https://archive.is/CbeYm#alternative-rust-crates-for-property-testing