Let a Thousand Flowers Bloom: on the Uses of Diversity in Software Testing

Alex Groce
Northern Arizona University
United States

Abstract

Software testing is *hard*; like many real-world problems, it offers no optimal solution, but requires dexterity, and the opportunistic combination of many partial solutions. Exploration and experiment, even by practitioners, are important in real-world critical testing efforts. An important set of research results in the field endorse and codify the value of *diversity* in the field. However, our current approaches to evaluating research results arguably cut against this fundamental reality: while effective testing may need true diversity, combining many partial answers, the iron logic of the research results section often imposes a *totalizing* vision where authors must at least pretend to present a monolithic Solution with a capital "S."

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; **Software testing and debugging**.

Keywords: software testing, test diversity, swarm testing, ensemble methods, test length, reseach evaluation methods

ACM Reference Format:
Alex Groce. 2021. Let a Thousand Flowers Bloom: on the Uses of Diversity in Software Testing. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

Discovering all the bugs in a software system is an extremely difficult task. It is so difficult, in fact, that in the real world it is seldom really attempted, for most kinds of software. However, when software bugs can produce catastrophic consequences, as in the case of safety-critical software, or produce catastrophic monetary or institutional credibility loss (e.g., a billion dollar Mars rover mission fails because of a bug [11]), it is worth at least trying to find *all* the bugs, given our difficulty in estimating the probability that an undiscovered bug will trigger in practice.

The difficulty in discovering bugs in software is not, alas, even of a simple kind. It is conceivable that, while "truly thorough testing" or "complete verification" would be *costly* and *require extensive human resources*, the methods for achieving these goals would be widely known, and omitted out of simple cost-benefit calculations. Unfortunately, even given willpower and budget, we generally don't know the best way to go about trying to find all the bugs in a system. Complete verification, for many real-world systems, is often essentially impossible, and even if possible would rely on a formal specification that might not represent important requirements. And, in testing, we seldom know which approach to testing will work best. Even given a particular "method," such as *fuzzing*, it turns out that the ability of experts to predict which approach(es) will be most effective is limited. Ask ten fuzzing researchers or practitioners what the best fuzzer is, and you won't get ten answers; but you also probably won't get fewer than four.

Moreover, even if everyone agreed on the best fuzzer, running just that fuzzer would likely be a bad decision! Different fuzzers are best at discovering different bugs, for almost all software programs. Any competitive fuzzer that is not strictly worse than another (e.g., the exact same fuzzer, but slower) is likely to have some bug(s) for which it is better than the "best" fuzzer.

The need for multiple approaches is even more complex when we consider that some bugs may be best detected by code review, some bugs may be best detected by manually written unit or integration tests, some bugs may be best detected by turning up the toleration for false positives in static analysis, and being willing to wade through all the resulting complaints, and so forth. However, even if we limit the topic to automated test generation methods, the fact remains. There are a very large number of possible approaches, and users serious about finding bugs may easily miss important bugs if they only use one of these methods, or even if they use a few of the best methods. In other words, this is a situation where the utility of *diversity* is extremely high. Diversity, here, means employing a *variety* of different methods, where in some sense many of these methods may be *worse* (on average!) than others.

This essay could, at this point, turn to marshalling evidence that a variety of approaches, even if some are "inferior"

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

to others, is needed for finding most bugs. The research literature and practical commentary includes substantial evidence for this fact. Some of it will be cited below; however, I think this isn't really very useful. Few software testing researchers probably don't know that there is, to particularize Brooks' general principle [3], "no silver bullet" in test generation. Even fewer serious practitioners of software testing who are any good at it will think there is such a thing. However, many seasoned testers may use only one method, or a handful of methods, because of the problem of diveristy. If you are using one, known pretty-good, method for automatically generating tests, the best practice is likley pretty clear: throw as many computing resources into running this method as you can, and try to solve the hard problem of triaging the resulting bugs and false positives you run into. But what if you want to use a diverse set of methods? Learning to use one method or tool is often time-consuming; learning to use every method and tool sounds like a nightmare. Two solutions offer themselves: first, there are some ways to introduce diversity into testing even with a single approach, that apply to many tools. Second, there are solutions that act as front-ends to diverse arrays of methods, saving the bug-finder the effort of learning to "talk" to each approach in its own unique language [6].

This essay will consider two aspects of this state of affairs. First, there is advice for the working bug-finder (developer, test engineer, systems engineer, security auditor); what diversity-aware approaches are available, to minimize the burden on the humble bug-finder, and make it possible to act as if you are using one, best, method? Second, and perhaps in the long run more importantly, there is the problem that *there is a serious divergence between software testing research expectations and publication barriers and the actual problems of software testing in a diversity-critical world.*

2 Digression: Diversity in General

The idea that diversity is good is not novel, or specific to software testing, of course. The title of this essay is cobbled together from famous phrases relating to the virtues of diversity [5, 9]. Since, at least, Montaigne [10], Cervantes' [4] admonition to not "put all your eggs in one basket", or perhaps the unknown Scottsman who said that it is good we all don't like the same things, or there would be a powerful shortage of oatmeal, diversity has been lauded, on occasion, in literature and philosophy. The idea of the American political system as using states as the "laboratory of democracy" is essentially diversity-based, as are arguments against too much central regulation by the European Union. Arguably, the same could be said for the city-states of the ancient world (perhaps there is something to be said for both Athens and Sparta, someone must have pointed out). There are ethical, utilitarian, and especially, aesthetic, arguments for preferring a multifarious rather than monolithic, world. This essay is only concerned with the *utilitiarian* approach to diversity exemplified by the folk saying about not putting all your eggs in one basket. If you only run one (very good!) fuzzer and it happens to bad at finding the most dangerous security vulnerability in your code, you will be at least as unhappy as the young maid who trips and pitches her one big basket of eggs on the ground in 17th century Spain. We don't care (here) if diveristy is right, or diversity is pretty, we just care that it is useful.

3 Diversity in Practice

In case anyone who actually goes about finding bugs in software is reading this essay, let's start by looking at some existing ways of making use of diversity. The first two approaches described will introduce "meta-diversity" — diversity within a single tool or method. These approaches describe ways to make the behavior of a single automated test generation method more diverse, with relatively little effort. They do not always apply to all methods, and, of course, as the nature of the problem this essay faces suggests, sometimes they don't work very well! But they are worth trying, as low-cost ways to add diversity when you can't, or don't want to, learn a new method or tool.

The second approach described is fundamentally based on the uses of diversity: using *ensemble* methods to allow a "single" tool to (behind the scenes) apply many methods, with the explicity rationale that diversity is good.

3.1 Test Length

Most automated test generation tools have a notion of maximum length of the tests generated. For fuzzers, this will often be a number of bytes; for API-call sequence generation, it will be the number of calls in each test. Most tools have default values for this parameter, and few users probably change those values. However, the value length can have a huge impact on the testing, and the impact is not as simple as there being an optimal value for the length [1]; different bugs may "want" different lengths!

Figure 1 shows the range of branch coverage achieved in two minute runs of the Python property-based API-sequence testing tool TSTL [8], for eight real-world libraries. The x-axis shows test sequence lengths, and the y-axis branch coverage achieved over 100 repeated runs. For each method, the "optimal" test length is noted; this length ranges from 10 to 1000 (the maximum length I used in experiments). However, just because the optimal length for a library is high or low does not mean that some branches and bugs for that library don't need a very different test length. For every single target, at least two branches were found where the optimal length for detecting *that branch* was not within 100 of the optimal value. For `pyfakefs`, `sympy`, and `sortedcontainers`, I know for a fact that real bugs, which I reported, are most

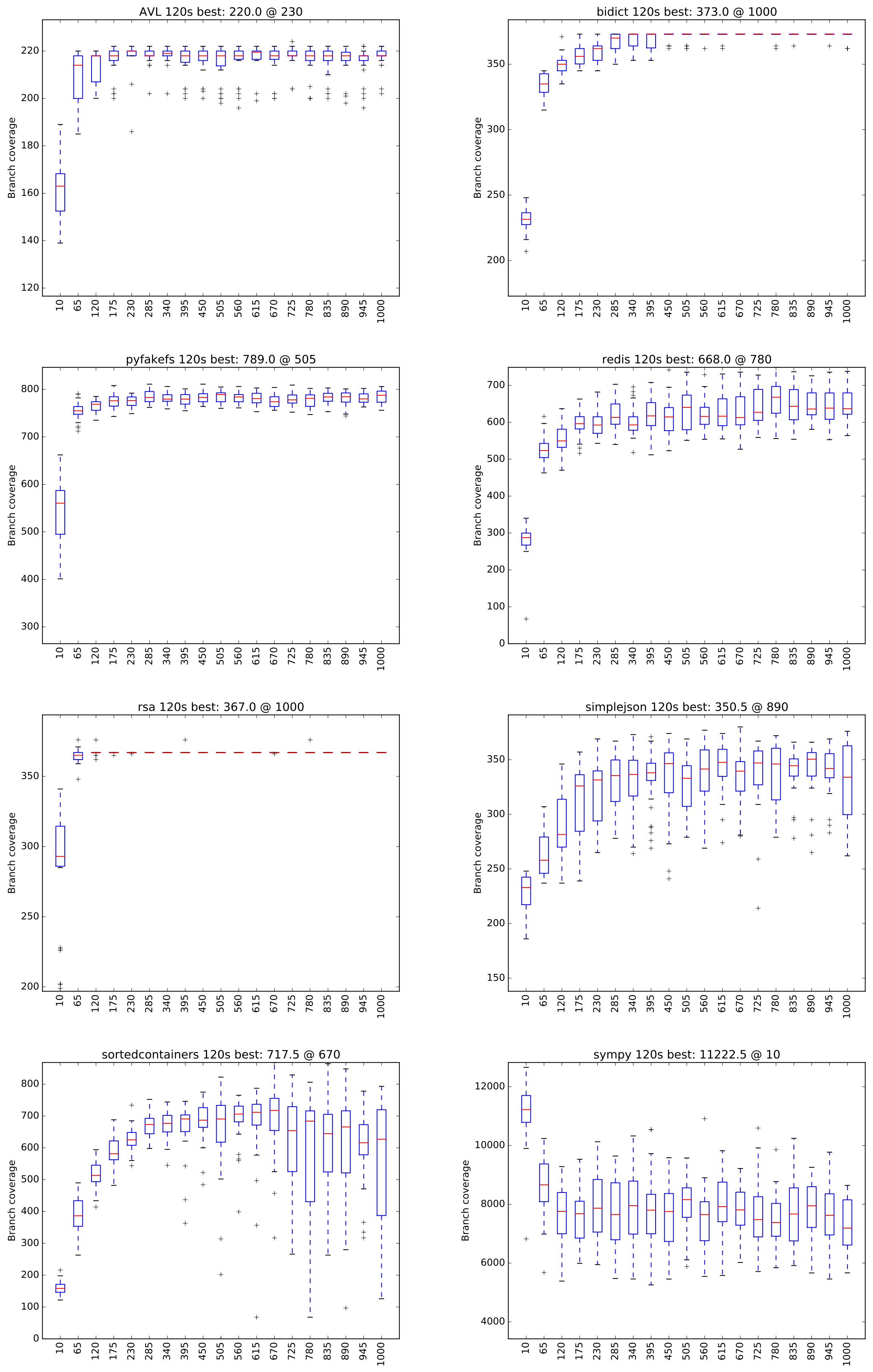

Figure 1. Best Test Lengths for Branch Coverage of Various Python Libraries

easily found using different test length choices than the optimal value, and moreover, from each other. In fact, for every program for which I have reported bugs, the bugs "live" at different test lengths.

It's not hard to understand why: consider a bug that is caused by an uninitialized value. Imagine there are two function calls, `foo` and `bar`. If you call `foo`, it will, as a side effect of some other activity, initialize the uninitialized value. In any given test sequence, after there is a call to `foo`, the bug will no longer be detectable. If you call `bar`, but haven't first called `foo`, on the other hand, the bug will immediately be detected, given certain choices for parameters to `bar`. In this case, rather than running a smaller number of long tests, where any testing after `foo` is called cannot find the bug, it is best to run a great number of fast, short, tests, in the hopes of hitting the right `bar` call before `foo` shows up.

On the other hand, consider the classic overflow bug, where calling `baz` 64 times in a row causes no problems, but calling `baz` the 65th time overflows a fixed-size buffer and causes a crash. If there are 20 different calls that can be made, and the length of a test is fixed at 100 calls, it's pretty hard to call `baz` enough times to trigger the bug before the "deadline."

As these examples show, the "kinds of things" that generated tests do really vary depending on the length of tests. In general, perhaps, "longer is better" [2] but for any particular target (bug or branch) you may be better off using diversity than optimality.

Thus, if you are using a test generation tool that lets you control the length of inputs, and especially of call sequences, it is a very good idea to vary the length parameter; it's a cheap way to get significant diversity in your testing, almost like running different tools without all the hassle!

3.2 Swarm Techniques

Figure 2 shows the basic logic of swarm testing [7]. The figure shows a 1,000x1,000 array of pixels, where each 10x10 block represents a sequence of 100 function calls in an API sequence test. Each pixel is a call to a function, and the calls to five different functions are coded by color (black, white, red, green, and blue). The top half of the figure is what traditional sequence generation will tend to do in such a setting, assuming each call is given equal probability: every test will look like every other test. The details will vary, but at a certain level the arrangement will be very homogenous; in fact, the eye can't tell where one test ends and another begins! Let's call this the kitchen-sink approach to testing: every test generated throws in everything it can, at least potentially.

The bottom half of the figure represents *swarm testing*. In swarm testing, before each test is generated, a coin is flipped for each of the five calls. If the coin turns up "heads", then that call is potentially included in the test; if the coin turns up "tails" the call is not made at all *in this test*. On average, the diversity between calls within each test is *much worse* for the swarm portion of the testing. However, it is easy to tell tests apart, with practical consequences. Beyond the visually obvious impact of swarm testing, there is simple statistical reality. While it is *possible* for a single method to be called 100 times using the kitchen-sink approach, the most instances of any single call we observed was 37. For the swarm tests, of course, each method was called more than 50 times (and in fact 100 times) in multiple tests. The set of behaviors is simply larger, and more diverse.

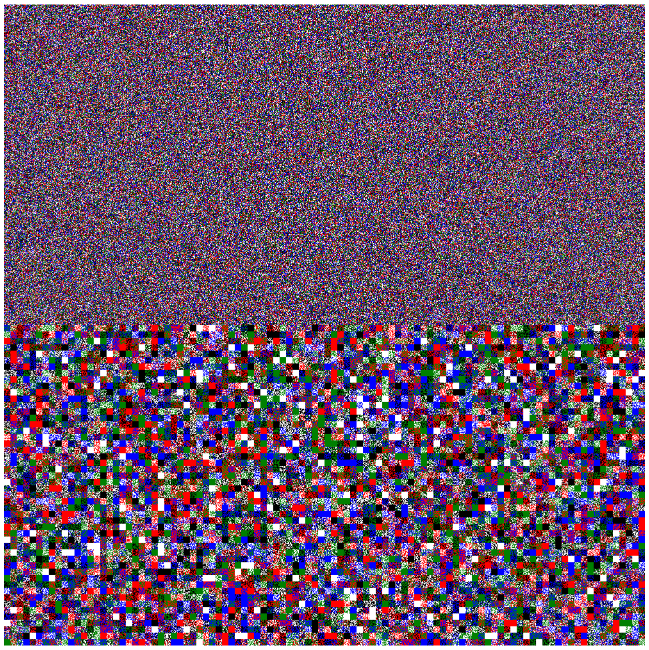
Figure 2. Kitchen-sink (top) vs. Swarm (bottom)

3.3 Ensemble Methods

4 The Totalizing Nature of the Experimental Results Section

Swarm testing [7] is pretty cool.

References

[1] James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *Automated Software Engineering*, pages 19–28, 2008.
[2] Andrea Arcuri. A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Trans. Software Eng.*, 38(3):497–519, 2012.
[3] Frederic P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20:10–19, 1987.
[4] Miguel de Cervantes. *Don Quixote*. 1605.
[5] G. K. Chesterton. *The Uses of Diversity: A Book of Essays*. 1920.
[6] Alex Groce and Martin Erwig. Finding common ground: Choose, assert, and assume. In *International Workshop on Dynamic Analysis*, pages 12–17, 2012.
[7] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *International Symposium on Software Testing and*

Analysis, pages 78–88, 2012.

[8] Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevi n Kellar, and James O'Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 20(1):57–78, 2018.

[9] Roderick MacFarquhar. *The Hundred Flowers Campaign and the Chinese Intellectuals.* Praeger, 1966.

[10] Michel de Montaigne. *Essays.* 1595.

[11] Glenn Reeves and Tracy Neilson. The Mars Rover Spirit Flash anomaly. In *IEEE Aerospace Conference*, 2005.