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*, and a testing problem is composed of many sub-problems with different, often conflicting, solutions. Like many real-world problems, it admits no single 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 test generation. 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, unitary solution, a new "king of the hill."

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

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

ACM Reference Format:
Alex Groce. 2021. Let a Thousand Flowers Bloom: On the Uses of Diversity in Software Testing. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '21), October 20–22, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3486607.3486772

1 Introduction

"Variety's the very spice of life,
That gives it all its flavour."
- William Cowper, "The Task"

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 [29]), 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 will-power 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 certainly won't get just one answer, and you may well get more than three different answers!

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,

Onward! '21, October 20–22, 2021, Chicago, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9110-8/21/10...$15.00
https://doi.org/10.1145/3486607.3486772

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.

One way to understand the fundamental need for diversity in software test generation is to think about the testing problem as an instance of the coupon collector's problem [4]. The generalized coupon collector's problem [5] is a probability problem in which colored balls are drawn from an urn, with replacement. In testing, typically, there are many more balls than colors (distinct faults, including the non-fault of a test that exposes no bugs); in fact, under a particular test generation strategy, the distribution of bugs often follows a severe power law, with some bugs extremely rare and others very frequent [9][1]. Collecting all the "coupons" in such a setting is difficult, *even if you can bias the selection of balls*, since many methods of bias will only work for the balls whose chromatic properties you have already observed, which might make it harder to find novel colors.

Testing is, therefore, fundamentally, not best seen as an *optimization* problem, at least not a simple optimization problem, since there is no single solution[2]. We want to hit *all* the bugs; moreover, since we have no idea where the bugs are, we probably want to hit all of some set of coverage targets, or synthetic bugs (as in mutation testing) [16]. Perhaps, in fact, coupon collection, while useful as a tool for mathematical analysis, is the wrong analogy. A better way to think of software testing is as a *scavenger hunt*, where it might be a good idea to split up the team, since finding a teacup with blue flowers and finding a Bunsen burner will probably involve trips to very different locations. Some trips may be fruitless (perhaps the local tea-room has only green flowers, or even plain white teacups), but cannot be omitted, without increasing the chance of missing some items on the list. Of course, in the case of bugs, the matter is complicated by the fact that the list of items to be found is not given to the team.

This essay could, at this point, turn to marshaling evidence that a variety of approaches, even if some are "inferior" 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 [7], "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 diversity. If you are using one, known pretty-good, method for automatically generating tests, the best practice is likely 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 [17].

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 and solutions 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 [11, 24]. Since, at least, Montaigne [27], Cervantes' [8] admonition to not "put all your eggs in one basket", or perhaps the unknown Scotsman 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). And, of course, diversity is a widely embraced principle today: ACM has a council on Diversity and Inclusion.

There are moral, ethical, political, philosophical, utilitarian, and, especially, aesthetic, arguments for preferring a multifarious, rather than monolithic, world. This essay is only concerned with the *utilitarian* 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 be bad at finding the most dangerous security

[1]In fact, getting out of a bad distribution where the undetected bugs are very rare is one rationale for diversity in test generation; even a distribution where failing tests are very rare can be very useful if some previously rare bug is now made less impossible to find.

[2]Obviously, minimally, there is no single "best test" and potentially there is no truly optimal strategy, when strategies do not include mixed approaches.

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 (inside this essay) if diversity 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 third 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 explicit rationale that diversity is useful in test generation.

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 [21], 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 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 [19]. 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 homogeneous; 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

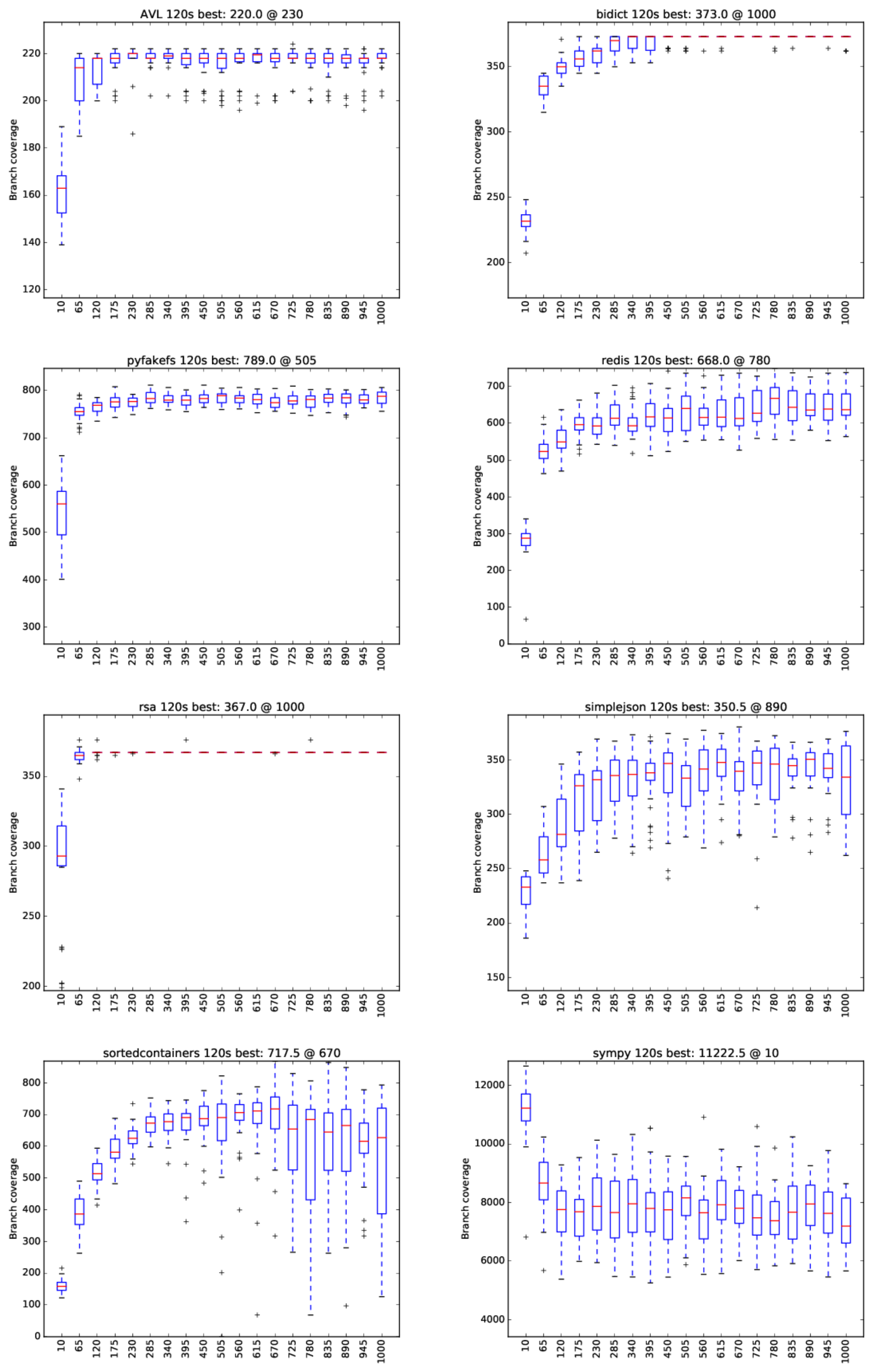

Figure 1. Best Test Lengths for Branch Coverage of Various Python Libraries

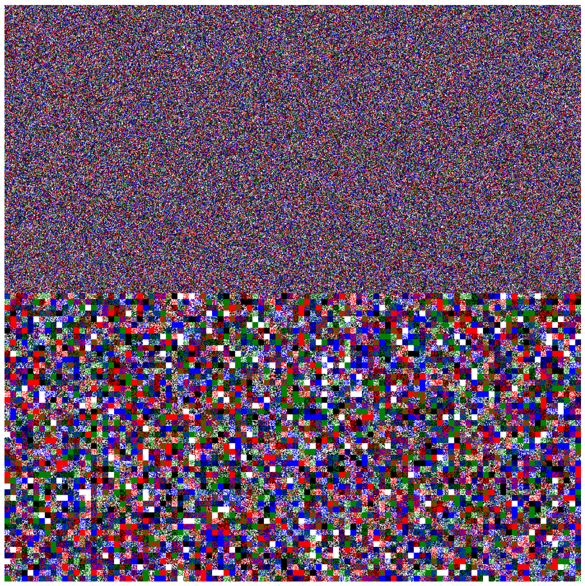

Figure 2. Kitchen-sink (top) vs. Swarm (bottom)

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. You'll *never* see a pure-red, or red-green, kitchen-sink test, even if it's technically possible. Probability is more important than possibility. A million monkeys *might* write Shakespeare, but perhaps it's best not to throw away your Riverside edition, just yet.

More concretely, consider the problem of testing a stack implementation that has incorrect overflow prevention code. Imagine the stack max size is 64 elements, and that the overflow allows writing to the 65th element (`stack[64]`), which will be detected by some form of memory safety analysis. If the stack has five methods we are testing, `push`, `pop`, `clear`, `top`, and `size`, and we assign equal probability to calling each of these methods, then random testing will tend to never generate a stack with 64 elements, that could expose the bug on a push. Rather, the mean stack size will be quite close to zero, and the maximum size reached in a test of length 100 will also be quite small. In swarm testing, on the other hand, one in eight generated tests will omit `pop` and `clear` and include `push` and so tend towards a quite large stack size. One in 32 tests will omit all calls *except* `push`, guaranteeing detection of the bug. The case is exactly analogous to the figure: if `push` is represented by **red** pixels, then the red "dots" in the bottom part of the figure represent tests guaranteed to detect the bug. In the top half of the figure, note that no square has a dominant red shade, so no tests would expose the bug.

The statistical impact of swarm testing is relatively easy to measure in a simple setting like this: the standard deviation of pixel counts for each color in the swarm image is more than five times higher than for the "kitchen sink" method. Diversity here is easily measured as increase in variance.

Swarm testing probably works because most coverage targets, and most bugs, likely *rely* on including some test features (e.g., function calls), which can be designated *triggers*, but also are *prevented* by other function calls (designated *suppressors*). In experiments, most targets and bugs in C compilers and file systems, at least, had a small number of triggers and a small number of suppressors, and were "indifferent" to other test features [18]. This makes sense; it is hard to design software humans can understand and use (much less implement successfully) where every functionality is intimately related to every other functionality[3]. Given this fact about the organization of software interfaces, it is clear that diversity of "kinds" of tests, in terms of features (elements of the interfaces) included is likely to be useful.

Swarm testing is fairly widely adopted in automated test generation. It is frequently applied to compiler testing [13, 23] and is a core element of the testing approach used for FoundationDB, the back-end database for Apple and Snowflake cloud services [30]. Tools implementing some form of swarm testing include TSTL [21], DeepState [15], and the very widely used Hypothesis framework [25][4]. Via DeepState, swarm testing can be used in function call or string generation in fuzzers such as afl and libFuzzer that do not natively implement a notion of swarm testing.

Swarm testing was inspired by swarm verification [22], which makes explicit-state model checking more effective by applying a variety of (often bad, but sometimes very good) tweaks to search parameters, effectively using a variety of model checking algorithms. Swarm verification is, arguably, essentially an *ensemble method.*

3.3 Ensemble Methods

Finally, ensemble fuzzing [10] is an approach that recognizes the need for diverse methods for test generation, at least in the context of fuzzing. Inspired by ensemble methods in machine learning [14], ensemble fuzzing runs multiple fuzzers, and uses inputs generated by each fuzzer to seed the other fuzzers. Ensemble fuzzing is currently supported by the Enfuzz website (http://wingtecher.com/Enfuzz) and by the DeepState front-end[5].

Less needs to be said about ensemble approaches than the other examples of how to add diversity. The whole point of

[3]Dijkstra famously [12] notes that the text of a program should correspond to its execution structure if we are to have any hope of working with it; the same factor limits most interfaces (and perhaps can help focus testing)
[4]See https://github.com/HypothesisWorks/hypothesis/issues/2643.
[5]See https://blog.trailofbits.com/2019/09/03/deepstate-now-supports-ensemble-fuzzing/.

ensemble methods is that a tool takes care of the diversity for you.

4 The Totalizing Nature of the Experimental Results Section

4.1 How It Is

In software testing conference papers, whether on fuzzing or on other test generation approaches (e.g., search-based testing [26]), the basic expectation is that there will be a research question like this:

> **RQ-N**: Does **our method** work better than **competing methods** for testing basically all programs we tried?

and an answer like

> **Our method** found a mean of 5.6% more faults than all other methods, including *N* previously undiscovered faults. **Our method** also improved code coverage, covering 3.4% more branches per run on average. Both results are statistically significant by Mann-Whitney U test [3], with $p < 0.001$.

If the answer to **RQ-N** is more nuanced than this, it is likely to result in at least one reviewer (which may be fatal at better conferences) or all reviewers questioning if the approach is really very useful, and if it is worth spending the precious time and space of the conference on such inconclusive results.

There are ways around this; one way is to hide the complexity of the results via various subterfuges, or to restrict the set of of subject programs to those whether the method worked, or just to lie; but we will restrict our attention to *honest* researchers.

It is permissible to write a paper with a subtler answer if you can *clearly define* the nature of the subtlety. For example, if your method is good for a particular kind of bug, then you can write a paper with a title like "A Method for Generating Tests to Detect **CATEGORY** Bugs" and then **RQ-N** can be answered thus:

> **Our method** found a mean of 5.6% more **CATEGORY** faults than all other methods, including *N* previously undiscovered faults. For non-**CATEGORY** faults, we found a mean of 8.1% fewer faults than other methods. **Our method** decreased code coverage, covering 3.4% fewer branches per run on average. However, when we identified branches associated with **CATEGORY** behavior, we improved code coverage by a mean of 1.4%. All results are statistically significant by Mann-Whitney U test [3], with $p < 0.001$. We conclude that our approach is successful at finding **CATEGORY** bugs.

However, either **CATEGORY** must be a known, accepted, and well-defined type of bug of general interest, or the bulk of the paper must be spent explaining and defending the introduction of **CATEGORY**. The success of the paper in being published will depend on whether reviewers decide that **CATEGORY** cleaves nature at the joints, or at minimum that **CATEGORY** relates to some hot topic of the day.

The basic expectation is *totalizing*. A good research paper in software testing will define a problem (perhaps a specialized, **CATEGORY**-defined problem), and then present a *new, best* way to solve that problem.

4.2 How It Should Be: Taking Diversity Seriously

Accepting any test generation paper where a method found at least one new bug would flood the research literature with uninteresting results. The situation described above did not arise through malice or ignorance, but through the desire of program committee members to weed out bad work. Nobody wants to fill ICSE or ISSTA with papers that end with a whimper, not a bang, e.g.:

> **Our method** found a mean of 2.6% fewer faults than all other methods. **Our method** also decreased code coverage, covering 1.4% fewer branches per run on average. Both results are statistically significant by Mann-Whitney U test [3], with $p < 0.001$.

Cowper, introducing his notion that variety is the spice of life, concluded that variety isn't all it is cracked up to be, and often leads simply to "monstrous novelty and strange disguise." In practice, however, this is not a realistic danger. Program committees will not accept many papers that end in a simple "our method did not work." However, it is also true that program committees, in a not-so-distant past, did probably accept too many interesting ideas evaluated poorly on toy problems[6].

There is a place for negative results, but most papers published should probably introduce *useful* methods. However, what if the answer was something like this?

> **Our method** found a mean of 2.6% fewer faults than all other methods. **Our method** also decreased code coverage, covering 1.4% fewer branches per run on average. Both results are statistically significant by Mann-Whitney U test [3], with $p < 0.001$. However, for 75% of subject programs, for at least *k* faults and *b* branch coverage targets, our approach improved the probability of detection per test by a factor of 10 or more, compared to all other methods. There was no way to predict which faults or branches would be thus affected. We speculate that the underlying

[6]In part, the current problem is a result of the increasing maturity of the field; learning how to evaluate research is often a challenge for emerging fields [28].

combinatorics of inputs relate to our use of **new technique**.

Assuming that standards are set high enough for k and b, and perhaps with some expectation that published methods at top conferences will show discovery of at least one novel fault, this seems like a potentially very useful result. In particular, the baseline comparison in a well-written paper is likely to be the set of widely-used methods at present. A problem facing most serious testing (especially fuzzing) efforts is that of saturation: you keep fuzzing with good tools, and you don't find any new bugs, or even add any new interesting inputs to your corpus, except when major changes are made to the code under test. Are you done? Is the fuzzing "finished"? Probably not, unless the program under test is fairly trivial. The problem is serious, and common. I myself was contacted by the team testing the bitcoin core implementation, after publishing (with John Regehr) a blog post on saturation[7].

One way to get out of saturation is to apply "worse" but useful methods. You used the best fuzzer, and the second-best fuzzer. Now try the fourth best fuzzer, and a brute-force really dumb but fast fuzzer, and this "one weird trick" that sometimes works, the literature shows. In the absence of a more diversity-aware literature, this kind of thing has to be done, but there's not a good, principled, peer-reviewed set of guidelines on how it should be done. For bitcoin core, the work is in progress, but simply running more fuzzers than the best-performing one (libFuzzer, in this case), for longer periods proved helpful.

Our research field is, to a large extent, failing to curate the literature on specialized, but ill-defined, test generation methods. The criticality of diversity, and in particular the problem of saturation, however, means that such specialized methods are likely *needed* in any truly high-stakes testing effort.

There is no denying that this places more burden on the program committee. The standards for what is a good-enough paper are already hard to realize; accepting the best partial methods means that in addition to scrutinizing experiments, reviewers have to give a lot more thought to the question of whether the underlying approach described makes sense, is unusual and not just an incremental improvement to a known method, whether it is well-enough described to make it plausible the partial results aren't just luck, and so forth. However, as it is, the approach tends to make it hard or impossible to publish many useful methods in many software engineering and testing conferences. It is likely that junior researchers, under the pressure to publish or perish, may abandon research on methods that are fruitful, but clearly specialized and unable to honestly climb over the hurdle of the expected **RQ-N** answer.

Software testing is *hard*. We need every genuinely useful, even specialized, tool we can discover, in order to find bugs. Right now, we're forcing test generation methods, in order to obtain publicity, to satisfy unrealistic constraints, and thereby rejecting many tools that probably should be in our belts.

[7]See https://blog.regehr.org/archives/1796.

4.3 How to Get Around It

In the absence of more diversity-friendly reviewers, there are ways to publish work on promising partial testing approaches. Each of these approaches has at least one major problem, however, in practice.

4.3.1 Journal Papers. Conference program committees basically only have the option to reject a paper with a weak answer to **RQ-N** or an unconvincing **CATEGORY** concept. In fact, the original swarm testing paper was rejected once, due to a single annoyed reviewer who said "this only seems useful for testing systems software" (with the implication, presumably, being that nobody would want to do *that*). Journal papers are a more collaborative kind of publication; unless the idea is just uninteresting or the results are *terrible*, it's possible to go back and forth and present a nuanced set of results. For one thing, there's space to elaborate on how often a method works, and explain that it is sometimes useful and sometimes not useful. A recent paper in TOSEM, where I had the honor of working the idea of a brilliant student through the process, after the student left the field, exemplifies the result [20]. The paper had trouble at conferences because the results showed that sometimes the proposed test generation heuristic was harmful, and for some programs resulted in much worse results. At conferences, even with a modest overall improvement (a middling-good **RQ-N** answer), that was fatal. In the expansive world of a journal paper, even at a top journal, it was acceptable to describe the highly varying results (sometimes the method was very powerful, and sometimes seriously detrimental), and even spend some time extolling diversity of methods. The journal setting gave us room to even spend some time expanding on the virtues of competing approaches. This is probably, right now, the best way to publish the best partial methods.

The problem is that people don't seem to read journal papers as much as they read conference papers. In practice, unless you have a wide following that reads all of your papers, even a TSE or TOSEM paper isn't going to obtain the audience that an ICSE or even ISSTA paper will. I tend to go look at the titles of the accepted papers lists for the top conferences in the field, to see if anything sounds exciting, but I only see a good TSE or TOSEM paper when Google Scholar decides to send me an alert.

Also, the turnaround for journal papers remains slow, so this may or may not be a useful avenue for junior researchers, depending on how they are evaluated. Finally, journal papers in good journals generally expect a very high standard of evaluation, much more than a conference paper will typically require for a promising and interesting-sounding method.

The iterative process where reviewers collaborate with authors, that makes it possible to publish partial results, also means that a published paper will likely incorporate various new experiments or evaluation measures demanded by reviewers as the price for admission. This is sometimes good, for readers of the paper, but can make the required effort, for junior faculty, or graduate students wanting to finish their PhD, look bad in a cost-benefit analysis. More importantly, it can delay the publication of important new methods that might be extremely useful in escaping saturation or as an addition to ensemble methods.

4.3.2 Ensemble Evaluation.

Speaking of ensemble methods, one interesting way to get around the totalitarian results section expectation is to "cheat" by adding your method to an ensemble approach. If you can show that, e.g., Enfuzz + **New Method** does much better by the traditional **RQ-N** standards, that may be enough to satisfy non-diversity-tolerant reviewers.

The problem is that ensemble frameworks are either fairly new and unstable (in fuzzing) or, to our knowledge, non-existent in many other automated test generation settings. This route may require substantial engineering effort, and some part of the paper will have to be devoted to justifying the use of an ensemble evaluation.

4.3.3 Preregistration.

One interesting way around the **RQ-N** wall is the use of preregistration. In preregistration, authors propose a method, and a paper is accepted or rejected based on how interesting and likely to be useful the method sounds. Only *after* in-principle acceptance or rejection is an evaluation conducted. Thus, if the method sounds good, and the **RQ-N** result is nuanced, the paper is still accepted. Proponents of preregistration argue that the approach avoid duplicated efforts, by allowing for the publication of negative results, and reduces the temptation for authors to over-claim in results sections. A group of top testing researchers has proposed to use preregistration to address problems in "the incentive structure for fuzzing research" and is planning to organize a journal issue to address the problem [6]. In practice, preregistration as a solution to publishing partial solutions is a subset of the journal solution, because by and large, well-known conferences in the field do not currently allow for preregistration.

4.3.4 Build a Tool.

Finally, if your testing work is not driven by the academic (or industrial lab) publish-or-perish imperative, you can find an interesting method and either build a tool that implements it, or add it to an existing tool. Testing tools and fuzzers are now widely used enough that if your approach is useful enough, even if only for some bugs, it may find an audience.

The problem is that this doesn't work for everyone, due to career needs [8]. Also, in practice, unless you have a base in a famous company or university, there are a lot of tools out there and nobody is going to hear about your tool. There are exceptions, such as David R. MacIver's Hypothesis, but by and large it is at least as hard to get attention for even a very useful new tool, unless you have an existing audience for your work, as it is to get attention for a new research paper or result. It is possible that using a tool to find bugs in important software is a way around this, but this requires a lot of work for uncertain payoff.

5 Conclusions

Software testing is a kind of scavenger hunt for the set of all bugs, where the consequence of not finding even one item on the (invisible) list may be a major security breach, the loss of a Mars mission, or the loss of human life. In practice, there are no silver bullets, no testing methods that are best for all bugs, and so the use of a variety of approaches is essential in serious testing efforts. There are existing practical ways for those who really want to find all the bugs to take advantage of diversity. However, it is likely that the available tools and methods are less diverse than they could be and should be, because of certain expectations of the research community. While these expectations are based on a desire to ensure the quality of testing research, they exclude important results. This essay offers no simple solution to this problem, but proposes that the software testing research community should begin to explore how to accept the reality that even some "poorly performing" testing methods may be essential foot soldiers in our war on error.

Acknowledgments: The author would like to thank the reviewers, and would particularly like to thank Tomas Petricek for a very interesting, *signed* review; the limits of time and my own understanding prevented me from making use of all his suggestions, but I now have **Image and Logic** on my reading queue, among other things. I would also like to thank numerous colleagues and collaborators for their discussion of these topics over many years; in particular John Regehr, Josie Holmes, Gerard Holzmann, Peter Goodman, Willem Visser, Mike Ernst, Andreas Zeller, and Gustavo Grieco, as well as my graduate students and many students in my software testing classes, have helped shape my thinking on these matters.

References

[1] James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. 2008. Random Test Run Length and Effectiveness. In *Automated Software Engineering*. 19–28.

[8]This essay may convince PCs that diversity is good, but it will never convince most promotion and tenure committees at universities that a tool that doesn't make the front page of the New York Times is worth even one minor journal publication.

[2] Andrea Arcuri. 2012. A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage. *IEEE Trans. Software Eng.* 38, 3 (2012), 497–519. https://doi.org/10.1109/TSE.2011.44

[3] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[4] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. 2010. Formal Analysis of the Effectiveness and Predictability of Random Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 219–230. https://doi.org/10.1145/1831708.1831736

[5] Leonard E Baum and Patrick Billingsley. 1965. Asymptotic distributions for the coupon collector's problem. *The Annals of Mathematical Statistics* 36, 6 (1965), 1835–1839.

[6] Marcel Böhme, László Szekeres, Baishakhi Ray, and Cristian Cadar. 2021. Journal Special Issue on Fuzzing: What about Preregistration? http://fuzzbench.com/blog/2021/04/22/special-issue/.

[7] Frederic P. Brooks. 1987. No Silver Bullet Essence and Accidents of Software Engineering. *Computer* 20 (1987), 10–19.

[8] Miguel de Cervantes. 1605. *Don Quixote*.

[9] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*. 197–208. https://doi.org/10.1145/2499370.2462173

[10] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *USENIX Security Symposium*. 1967–1983.

[11] G. K. Chesterton. 1920. *The Uses of Diversity: A Book of Essays*.

[12] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.). 1972. *Structured Programming*. Academic Press Ltd., GBR.

[13] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust typechecker using CLP (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 482–493.

[14] Thomas G Dietterich et al. 2002. Ensemble learning. *The handbook of brain theory and neural networks* 2 (2002), 110–125.

[15] Peter Goodman and Alex Groce. 2018. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*.

[16] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. 2014. Coverage and Its Discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward2014)*. Association for Computing Machinery, New York, NY, USA, 255–268. https://doi.org/10.1145/2661136.2661157

[17] Alex Groce and Martin Erwig. 2012. Finding Common Ground: Choose, Assert, and Assume. In *International Workshop on Dynamic Analysis*. 12–17.

[18] Alex Groce, Chaoqiang Zhang, Mohammad Amin Alipour, Eric Eide, Yang Chen, and John Regehr. 2013. Help, help, I'm being suppressed! The significance of suppressors in software testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 390–399.

[19] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *International Symposium on Software Testing and Analysis*. 78–88.

[20] Josie Holmes, Iftekhar Ahmed, Caius Brindescu, Rahul Gopinath, He Zhang, and Alex Groce. 2020. Using Relative Lines of Code to Guide Automated Test Generation for Python. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 28 (Sept. 2020), 38 pages. https://doi.org/10.1145/3408896

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

[22] Gerard Holzmann, Rajeev Joshi, and Alex Groce. 2011. Swarm Verification Techniques. *IEEE Transactions on Software Engineering* 37, 6 (2011), 845–857.

[23] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices* 49, 6 (2014), 216–226.

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

[25] David R. MacIver. 2013. Hypothesis: Test faster, fix more. http://hypothesis.works/.

[26] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14 (2004), 105–156.

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

[28] Dan R. Olsen. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (Newport, Rhode Island, USA) *(UIST '07)*. Association for Computing Machinery, New York, NY, USA, 251–258. https://doi.org/10.1145/1294211.1294256

[29] Glenn Reeves and Tracy Neilson. 2005. The Mars Rover Spirit Flash Anomaly. In *IEEE Aerospace Conference*.

[30] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *ACM SIGMOD*.