Does It Always Do That? Practical Lightweight Nondeterminism and Flaky Test Detection and Debugging

Alex Groce
School of Informatics, Computing & Cyber Systems
Northern Arizona University
Email: agroce@gmail.com

Josie Holmes
School of Informatics, Computing & Cyber Systems
Northern Arizona University
Email: josie.holmes@nau.edu

***Abstract*—A critically important, but surprisingly neglected in the software testing literature, aspect of system reliability is system predictability. Many software systems are implemented using mechanisms (unsafe languages, mulitple threads, complex caching, stochastic algorithms, environmental dependencies) that can introduce behavioral nondeterminism. Users of software systems, especially other software systems using library calls in a single-threaded context, often expect that systems will behave deterministically in the sense that they have predictable results from the same sequential series of operations. Equally importantly, even when it does not impact correctness or user experience, nondeterminism must be understood and managed for effective (especially automated) testing. Nondeterministic behavior that is not either controlled or accounted for can result in flaky tests as well as causing problems for test reduction, differential testing, and automated regression test generation. We show that lightweight techniques, requiring (almost) no effort on the part of a developer, can be used to extend an existing test generation system to allow checking for problematic nondeterminism. We introduce a set of lightweight nondeterinism properties, inspired by real faults, and tailor these notions to the practical, automatic, checking of Python library code.**

I. Introduction

For ourselves, we might prefer to think (and act as if) we had free will; however, we generally prefer our software systems to be as constrained in their actions as possible: in other words, we wish them to be largely deterministic, from our perspective.

Determinism is particularly important for testing and debugging, where being able to exactly reproduce system behavior is essential to productivity. If executing "the same test" can produce significantly different behavior each time it is run, the consequences can be unfortunate. Developers using a test exhibiting nondeterminism to debug a system face a challenge in reasoning about causality, in that an event observed may not even take place the next time the test is run. Automated test generation systems using test coverage results to drive the search for interesting inputs may be mislead when a run executes code only rarely. And, most significantly, regression testing effectiveness can be significantly reduced if tests fail only intermittently and unpredictably, as a result of environmental factors rather than bugs in changed code. Such behavior is unfortunately all too common: Gao et al. [4] observed coverage differences of up to 184 lines of code for the same test, and false positive rates as high as 96%. Nondeterminism is sometimes problematic for developers, who will usually want to assume that library code they use is deterministic; it is often disastrous for large-scale or highly automated testing.

A. What is Determinism?

A system is deterministic if, given the system's complete state at a point in time, it is possible, in principle, to predict its future behavior perfectly. We say "in principle" because in the real world, prediction may be possible but hopelessly impractical. We write complex software systems in many cases because we cannot predict their behavior (if we could perform the calculations in advance, ourselves, we would just do so). As a consequence, in software, rather than defining determinism in terms of prediction, we usually therefore simply say that a system is deterministic if, given the same state and inputs, it always produces the same outputs[1]. The change of definition, we will see, is central to our approach, and comes with a number of limitations.

Technically, many "nondeterministic" systems are not nondeterministic in a strong sense, at all. Given the complete state of the system (which includes the entire state of the underlying hardware, the operating system, storage devices, etc.), ignoring quantum effects, and treating outside interventions such as network traffic, human activity at an input device, etc. properly as inputs, most software *is* completely predictable. What we actually mean, usually, is that, given a certain limited abstraction of state and of inputs, observable behavior is repeatable. This abstraction, for most software, is not expected to include many elements outside of the software system itself.

Because the programmer's abstraction of the system ignores such a large number of details, very few tests are in fact "deterministic" in the sense that they eliminate all changes in behavior between executions. Running the same test twice almost always results in differences, given a low enough level of abstraction, since the system load, cache contents, branch

[1]Of course, this shift of definition is often adopted in arguments for physical or human behavioral determinism, due to the even greater difficulties of prediction in those arenas.

predictor history, etc. are almost never controlled for; however, this kind of nondeterminism is usually of no interest, unless the test involves extremely precise timing constraints. Rather, what matters is when some kind of nondeterminism unexpectedly impacts computed values at a higher level of abstraction; in general, if the operating system itself is not buggy, low-level nondeterminism, by design, is invisible except in fine-grained performance testing or real-time systems. Unexpected nondeterminism usually arises when there is an element of higher-level state or input that *is* critical to the produced behavior, but the programmer has not anticipated. E.g., when it is believed that the behavior of a thread scheduler will not matter, but a race condition in the code means that it does matter, or when the order of items in an iterator on a hash table is important, and the hash values used are salted and thus not predictable.

B. The High Cost of Unexpected Nondeterminism

Unexpected nondeterminism is, unfortunately, usually only discovered in a context that makes it very hard to debug. The most common such contexts are occasional rare failures of a system in deployment, and regression tests that do not behave reliably (known as flaky tests). Furthermore, unexpected nondeterminism makes it difficult to use automated test generation to produce effective regression tests for a system. While developers may know how to produce reliably deterministic unit tests, even in the presence of underlying nondeterminism, automated test generation, without a large investment in human time, usually does not have sufficient information to avoid producing the occasional such test. Moreover, where a human might make an assertion about the final value produced during a unit test, an automatically generated regression test is usually most easily produced by simply asserting equality with all values produced during a reference run, since the final step of a test is likely to be somewhat arbitrary, generated in an effort to increase code coverage, not establish some functional correctness property. This may be one reason that automated testing is seldom used to produce general regression tests. While failure-inducing tests produced automatically are often added to regression suites, the full set of passing tests is, to our knowledge, only infrequently preserved as part of a typical regression suite, perhaps because such tests are likely to be *flaky* without considerable human effort.

1) Nondeterminism and Flaky Tests: In order to help ensure that they are reliable and secure, complex modern software systems usually include a large set of *regression tests*. A regression test suite is a (usually large) set of tests that can be run against a software system every time it is modified, to ensure that the modification has not broken the system in some way. *Flaky tests* [17] are regression tests that fail in an intermittent, unreliable fashion, and thus degrade the utility of regression testing. The essence of a flaky test is that, for the same snapshot of test code and code under test (CUT), it sometimes fails and sometimes passes: the pass/fail result (disposition) of the test is not a deterministic property of the test code, code under test, and environment running the test. This produces three serious problems: first, a flaky test often wastes developer time and delays software changes by forcing the investigation of code-under-test that is not actually incorrect. Second, failures in flaky tests (for that reason) are often ignored, and therefore serious software faults may be missed. Finally, to mitigate these problems, flaky tests are often run multiple times, wasting valuable computing resources and also delaying acceptance of code changes. A canary in a coal mine is of little use if canaries frequently become ill for reasons unrelated to the presence of toxic gases. Mining may stop for no good reason, or miners may learn to ignore the canary, leading to tragedy; a third, more "practical" option is that miners may carry so many redundant canaries into the coal mine that canary-care becomes a serious burden on mining.

Flaky tests are, for us, simply a special case of general test nondeterminism, where the nondeterminism of test values is sufficient to cause the pass/fail result of the test to vary. The definitions and techniques proposed in this paper all apply to flaky tests, as a special case of various kinds of nondeterminism described below.

C. Contributions: Detecting and Debugging Nondeterminism

This paper proposes (1) a number of formal definitions of types of nondeterminism and (2) a number of techniques, based on these definitions, for detecting and debugging nondeterminism in property-based testing, particularly for library code. The methods also work for testing at the system level, but are perhaps most effective for identifying the low-level sources of nondeterminism in libraries. This enables a number of useful results: the automated production of nondeterminism-free regression tests even for libraries with some sources of nondeterminism, the documentation of library nondeterminism, and, most importantly, detection and effective debugging of unexpected nondeterminism that is, itself, a bug. We first describe the types of nondeterminism we aim to detect, and the extension of a widely used debugging tool to better handle nondeterminism as a test property, then describe an implementation of these ideas in a Python testing tool that is used on real-world projects. Finally, we use a set of case studies of real-world Python libraries to demonstrate the utility of our ideas and understand the cost of adding nondeterminism detection to existing automated testing.

II. Practical Lightweight Nondeterminism Detection and Debugging

We define two basic types of determinism, shown in Figure 1: horizontal determinism and vertical determinism. In horizontal determinism, which is what we usually think of when we think about deterministic behavior, a software system reliably produces the same behavior given the same steps of a test: the behavior of multiple executions that are "the same" in terms of inputs/actions can be aligned and checked for some type of equality. In vertical determinism, rather than behavior across executions, we are interested in behavior within an execution, where repeating the same step of a test twice should

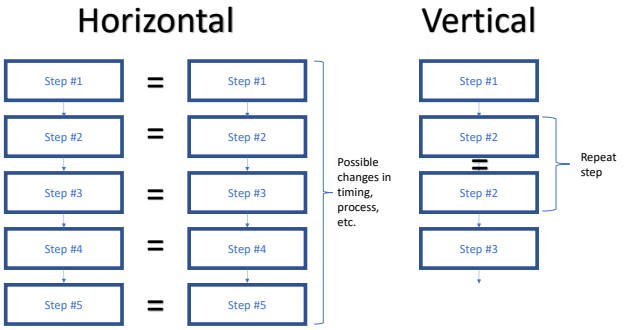

Fig. 1: Types of determinism

result in the same behavior. Obviously, vertical determinism is both rarer and more specialized than horizontal determinism. However, in those instances, vertical determinism is arguably more important than horizontal determinism, in that violations usually indicate a potentially serious fault.

A. Horizontal Determinism

1) Determinism and Reflexive Differential Testing: Horizontal determinism can be best understood by thinking of nondeterminism detection as an unusual kind of *differential testing* [15], [6]. In differential testing, a system is compared against a reference in order to ensure that it behaves equivalently, at some level of abstraction, to another implementation of the same functionality. Differential testing is extremely powerful, in that any divergence of behavior, if divergence is correctly defined, indicates a functional correctness fault in (at least) one of the systems under test. Being able to detect functional correctness errors without the cost of constructing a formal specification is extremely useful in automated test generation. Differential testing is widely used for systems software components such as compilers [15], [23] and POSIX file systems [7], [5], where multiple implementations are common. The major limitation of differential testing, of course, is that multiple implementations of a system are almost as rare as good correctness specifications.

For the special case of detecting nondeterminism, however, *a system can serve as its own reference implementation.* The question, then, becomes one of deciding at what granularity the reference equivalence will be checked: as discussed above, processor-instruction and memory-layout determinism is seldom necessary or even desired (it would greatly limit optimizations of code execution). We propose two approaches to "aligning" an execution with itself.

2) Visible Value Determinism: Visible value determinism uses the human-accessible outputs (displayed or stored to a file), and *values returned by functions or methods called as a library by other code*, as the criteria for determining if two executions are equivalent. The idea is simple: determinism is motivated by the desire to create consistent behavior for an observer, whether that observer is a human user, another software system, or a regression test. The values output by a software element are the only values of interest to the observer. In practice, of course, some values (time stamps, pointer addresses, etc.) are not expected to be deterministic by an observer; we call these values *opaque* in that they are not interpretable as "showing" the internal workings of the code being tested for determinism. Rather, they mask an abstraction, usually one managed by the operating system (system time, memory management). Any mechanism for visible value determinism needs to support automatic and manual designation of some values as opaque.

3) Final State Determinism: Visible value determinism has significant limitations. While it provides the finest granularity for detecting nondeterminism, which is important for debugging purposes, it is also expensive, requiring checks on (and possibly storage of) a potentially very large number of values. Additionally, in some cases so many values produced by a system are opaque that the annotation burden is inordinate. In these cases, it is better to only compare final states of a computation performed by the system being checked for determinism. The final state may have opaque components, but it is easy to define an abstraction that reduces the state to the actual value of interest.

B. Vertical Determinism

Vertical determinism is a property that expresses that some operations of a software system should, within the same trace, always behave the same way. Usually, for interesting cases, this is dependent on some state of the system, though some operations should be completely state-independent. E.g, the hash of a given bytestring returned by a security library should never change. This is one aspect of *pure* functions. For nondeterminism checking, the interesting cases are non-pure: a function depends on system state, but should always depend on it in the same way, and should not, itself, change system state in a way that would change its behavior on a subsequent run.

Many *idempotent* operations fall into this category. Consider adding an element to an AVL tree implementing a set, not a multiset. Assume the method call returns the parent node of the element, whether it was added or was already in the tree. Calling this mehtod any number of times in sequence should always return the same value, though the first call may have modified the contents of the AVL tree.

1) Failure Determinism: A specialized case of vertical determinism is failure determinism. Failure determinism is the following restriction on an API:

If a call fails, and indicates this to the caller, it should not modify system state in any way. Changes made before the failure should be rolled back.

In other words, failure determinism is a property stating that an API is *transactional* with respect to failures (it may not be so with respect to non-failing calls).

From a user's perspective, some behaviors of the macOS High Sierra root exploit (CVE-2017-13872 [19]) exhibited failure nondeterminism. Attempting to login with the root

account with an empty password appeared to fail, then, on a repeated try, succeeded [18], [13].

C. Nondeterminism and Delta-Debugging

Delta-debugging [25][2] is a widely used method for reducing the size of failing tests, making them easier to understand and debug. Delta-debugging in the context of detecting nondeterminism has two purposes. One is simply the usual goal of reducing the size of a test. Identifying the cause of nondeterminism may be very easy in a test consisting of ten library function calls (it is one of these ten calls), but very difficult in a test consisting of a hundred library function calls. This is no different than the common use of delta-debugging. However, in nondeterminism detection, delta-debugging can also be used to *change the probability of nondeterministic behavior*.

In *monotonic* cases, removing a part of a test cannot increase the probability of a predicate holding. This is fairly common. In these cases, when we use a reduction algorithm to reduce t to r, $P(p(r) \leq P(p(t))$. In *non-monotonic* cases, however, there is no such upper bound. Removing a step in t may increase the probability that t behaves nondeterministically.

The probabilistic predicate of interest for determinism is always of the form: **the test will exhibit at least two different behaviors in S runs, with probability P**. This predicate may be monotonic or non-monotonic, depending on the causes of the nondeterminism detected.

The only absolutely necessary changes required to use standard delta-debugging implementations in nondeterminism detection are simply removal of some "sanity checks" in the code: many implementations (including the Python code provided by Zeller) assert that the predicate holds on the original test. In probabilistic settings, this may not be true, and we may even be trying to find a subset where a predicate that is very far from holding on the original test holds (in a non-monotonic case where we aim to increase the probability of nondeterminism).

1) "Publication Bias" in Delta-Debugging: However, simply using delta-debugging off the shelf with a predicate like $P(fail) > 0.3 \wedge P(fail) < 1.0$, to force a test to be flaky, and force it to fail sufficiently often to be used in debugging, will often produce surprising and unfortunate results. In many cases, delta-debugging will indeed reduce the large test to a small subset. And, in a technical sense, delta-debugging will work: it will never convert a nondeterministic test to a completely deterministic test, because the reduced test that delta-debugging returns is always one where the predicate of interest has been seen to evaluate to true. However, if you run the resulting test, it will, in many cases, have a $P(fail)$ that is much, much smaller than 0.3, perhaps as low as 0.01. Why?

The problem is analogous to the problem of publication bias in scientific fields [1]. A predicate like $P(fail) > 0.3 \wedge P(fail) < 1.0$ cannot be evaluated by determining the true probability of failure; rather, the test must be run some concrete number of times, and the number of failures counted. However, even if the number of samples N is large, there is some probability (based on the sample size) of a result that diverges significantly from the actual probability of failure. If the predicate were run only once, and the number of samples reasonably large, this would not matter. However, by their nature, delta-debugging and other test reduction algorithms, explore a search space that often contains hundreds or even thousands or millions of candidate tests. The predicate is evaluated on each of these, and so even with large N, it is extremely likely that some evaluation will produce a very poor estimate of $P(fail)$. If such an evaluation causes the predicate to appear to hold for a test, delta-debugging is "stuck" with the error, because (to our knowledge) no test reduction algorithms allow backtracking. After such a mistake, finding further reductions will become harder, but may still be possible due to the same source of errors: the number of experiments run is far larger than the probability of an incorrect result.

It is the combination of a one-way bias on faulty evaluations (the consequences of a false positive for the predicate are much greater than for a false negative) and the huge number of experiments relative to error rate, that produces bad results, akin to the magnification of effect sizes in science due to publication bias. The bias in delta-debugging tends towards producing a reduced test where probabilities are much smaller than demanded by a predicate, due to the pressure on delta-debugging to produce shorter tests. Shorter tests have smaller probabilities, on average, due to two factors: first, timing-induced nondeterminism has a much smaller temporal space to operate in, and second, if some test operation introduces a small probability of nondeterminism, and only many repetitions of that operation make the probability large, small tests obviously on average have fewer instances of the problematic operation.

In scientific literature, the most frequently proposed solution is the use of replications: repeated runs of "successful" experiments to minimize the probability that a result is a fluke due to publication bias. One way to produce this effect would be to allow delta-debugging to backtrack if the probabilties observed in predicate evaluations suddely exhibit a strong discontinuity, a kind of "paper retraction" based on near-replications. However, this requires modifying the delta-debugging implementations, which is difficult and sometimes not really feasible (e.g., few users of AFL-fuzz [24] will wish to change its C code for test reduction). Ideally, the solution should be implementable simply by modifying the predictate that is evaluated.

A costly but effective solution is to make N large in comparison to the number of expected predicate evaluations performed during delta-debugging. If the error rate is low enough, the problem disappears. However, given the large number of evaluations performed, this will tend to make delta-debugging extremely slow. We propose using a dynamic sampling approach, where N is small, but if the predicate evaluates to true, M repeated true evaluations ("replications",

[2]We use delta-debugging to stand for all test reduction algorithms, even those [21], [9] that do not use the exact binary-search that distinguishes delta-debugging *per se*; the differences are immaterial for our purposes.

where the first true evaluation is counted as 1 replication) are required before the predicate returns "true" to the delta-debugging algorithm. A set of M repeated false positives with $\frac{N}{M}$ samples each is much less likely than a false positive with N samples; so long as we accept the resulting bias in favor of false negatives, we can therefore produce a reduced test with a desired $P(fail)$ much more cheaply: the use of replications not only means we only pay the full sampling price on rare occasions, but a desired accuracy for P can be obtained with a much smaller value $M \times N$ than a non-dynamically-sampled N. For example, if we the predicate is $P(fail) \geq 0.5$, and the true probability for a candidate test is 0.25, using $N = 8$ will give a false positive rate of over 10%. Using 4 replications on just 2 samples ($M = 4; N = 2$) yields a false positive rate of about 3.7%, yet requires almost 60% fewer executions of the test[3].

Tuning M, any degree of confidence can be achieved, with the basic tradeoff being between finding a test with the desired probabilities, and the speed and effectiveness of delta-debugging. Because increasing M makes false negatives more likely, larger M will usually result in less-than-optimal reduction of the original test.

III. Implementation

A. Sources of Nondeterminism in Practice

B. Implementing Nondeterminism Detection

We implemented our approach in the TSTL [8] system.

Because TSTL supports differential testing [10], horizontal nondeterminism detection can technically be implemented simply by declaring a system to be its own reference, using TSTL's notation for differential testing. However, such an approach requires some effort on the part of the user to control which values are checked for equivalence, and does not easily support injecting timing differences, or using a new process, in checking the behavior.

Because TSTL has an interface to afl [24], we can even use afl's sophisticated heuristics to perform very thorough, week-long checks for nondeterminism, using strategies built to find subtle security vulnerabilities in C programs.

IV. Experimental Evaluation

A. Case Study: Redis-Py

B. Case Study: pyfakefs and OS X

C. Case Study: Datarray Inference Algorithms

https://github.com/BIDS/datarray

[3]The probability calculations for such comparisons are relatively simple, but in real testing are usually not very useful, since the true probabilitiy distributions of test behaviors vary widely and dynamically during the delta-debugging process, making exact calculation unwieldy, even if the changing probabilities were known; exact results would be based on fictional probabilities, so gathering experimental data during a trial delta-debugging run and tuning N and M to yield desired results is more effective.

V. Related Work

Gao et al. generally considered the question of how to make tests repeatable [4], in the context of system-wide user interaction tests. Their work focused on systemic factors, such as execution platform and persistent configuration and database files, in contrast to our focus on identifying nondeterminism at the library level. However, what they refer to as "test harness factors" includes delays, which can be of importance to nondeterminism at the library level when asynchronous behavior is involved.

Shi et. al [22] examined what might be seen as a related, though in a sense the opposite, problem: they detect code that assumes a deterministic implementation of a non-deterministic specification. E.g., they detect instances when iteration through items of a set is explicitly not guaranteed to provide any particular order of items, but code depends on the order produced by a given implementation. This procedure, applied to Python code in a pre-3.3 environment, would have flagged many instances of the nondeterminism that arose on the introduction of salted hashing. Determinism is also sometimes used as a property in domain-specific testing efforts, e.g. in testing shader compilers [3].

The problem of test nondeterminism is closely related to the (extremely important in industrial settings) issue of flaky tests [17], [14], [20], [12]. How to handle flaky tests in practice (when they cannot be avoided) is a major issue in Google-scale continuous testing [16], and, as Memon et al. describe, the problem of flaky tests influences the general design of Google test automation. Previous work on flaky tests has either focused on test inter-dependence as a cause of flaky behavior [11], or provided large-scale empirical examination of tests from one large open source project (Apache) [14], [20]. Palomba and Zaidman [20] investigated the relationship between test code smells and flaky tests. The present paper, rather than focusing on flaky tests per se, investigates new tools for handling nondeterminism in property-based test generation. From our point of view, flaky behavior is simply a special case of nondeterminism, where the nondeterminism is sufficient to cause the test to have different pass/fail results at times.

Other efforts [2] have aimed to avoid nondeterminism in parallel implementations, by design, indicating the importance of avoiding nondeterministic behavior, even at the expense of adopting novel programming models, where the goal is not simply (as in, say, Rust) to avoid classic concurrency errors, but to enforce genuinely deterministic behavior.

VI. Conclusions and Future Work

References

[1] I. Ahmed, A. J. Sutton, and R. D. Riley. Assessment of publication bias, selection bias, and unavailable data in meta-analyses using individual participant data: a database survey. *British Medical Journal*, 344:d7762, 2012.

[2] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

```
@import inference_algs
@import datarray

<@
def flatten_and_sort(v):
    return (sorted(map(flatten_and_sort,v),key=repr) if type(v) in [list,tuple] else
                (flatten_and_sort(list(v.items())) if type(v) == dict else v))
def psplit(P):
    return ([P,1.0-P])
@>

pool: <P> 3
pool: <cpts> 3
pool: <evidence> 3 OPAQUE
pool: <ename> 3
pool: <event> 3

<P> := 0.01 * <[0..100]>

<ename> := "E" + str(<[1..5]>)

{Exception} <event> := [datarray.DataArray(psplit(<P>), axes = [<ename>])]
{Exception} <ename,1>!=<ename,2> -> <event> := [datarray.DataArray([[psplit(<P>)],psplit(<P>)],[<ename>,<ename>])]

<cpts> := []
~<cpts>.append(<event>[0])

<evidence> := {}
~<evidence>.update([(<ename>,0)])

{Exception} print(flatten_and_sort(inference_algs.calc_marginals_simple(<cpts>,<evidence>)))
{Exception} print(flatten_and_sort(inference_algs.calc_marginals_sumproduct(<cpts>,<evidence>)))
{Exception} print(flatten_and_sort(inference_algs.calc_marginals_jtree(<cpts>,<evidence>)))
```

Fig. 2: Complete TSTL harness for finding the hash-order bug in the datarray inference algorithms.

[3] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson. Automated testing of graphics shader compilers. *PACMPL*, 1(OOPSLA):93:1–93:29, 2017.

[4] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. Making system user interactive tests repeatable: When and what should we control? In *International Conference on Software Engineering*, ICSE '15, pages 55–65. IEEE, 2015.

[5] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.

[6] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[7] A. Groce, G. Holzmann, R. Joshi, and R.-G. Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *Workshop on Constraints in Formal Verification*, pages 1–15, 2008.

[8] A. Groce and J. Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.

[9] A. Groce, J. Pinto, P. Azimi, P. Mittal, J. Holmes, and K. Kellar. TSTL: the template scripting testing language. https://github.com/agroce/tstl.

[10] J. Holmes, A. Groce, J. Pinto, P. Mittal, P. Azimi, K. Kellar, and J. O'Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 20(1):57–78, 2018.

[11] W. Lam, S. Zhang, and M. D. Ernst. When tests collide: Evaluating and coping with the impact of test dependence. Technical Report UW-CSE-15-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Mar. 2015.

[12] J. Listfield. Where do our flaky tests come from? https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html, April 2017.

[13] R. Loyala. macos high sierra 'root' security issue allows admin access. https://www.macworld.com/article/3238868/macs/macos-high-sierra-root-security-issue-allows-admin-access.

[14] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[15] W. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[16] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming Google-scale continuous testing. In *International Conference on Software Engineering*, pages 233–242. IEEE, 2017.

[17] J. Micco. Flaky tests at Google and how we mitigate them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html, May 2016.

[18] S. Nichols. As apple fixes macos root password hole, here's what went wrong. https://www.theregister.co.uk/2017/11/29/apple_macos_high_sierra_root_bug_patch/.

[19] NIST. Cve-2017-13872. https://nvd.nist.gov/vuln/detail/CVE-2017-13872.

[20] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2017.

[21] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Programming Language Design and Implementation*, pages 335–346, 2012.

[22] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–90, April 2016.

[23] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation*, pages 283–294, 2011.

[24] M. Zalewski. american fuzzy lop (2.35b). http://lcamtuf.coredump.cx/afl/. Accessed December 20, 2016.

[25] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.