Does It Always Do That? Practical Automatic 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, multiple 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 nondeterminism 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 [8]. 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; this pernicious phenomenon has been dubbed the dreaded "Heisenbug" [10]. 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. [9] 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 frequently vexing in debugging efforts; it is often disastrous for large-scale 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.

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 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 randomly salted.

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* [32] 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. An analogy may make the extent of the problem in large-scale test automation clear: 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.

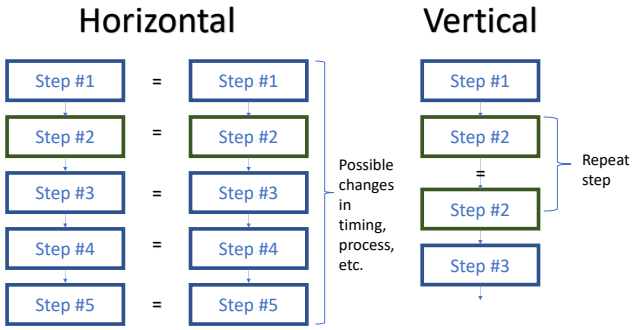


Fig. 1: Types of determinism

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 *horizontal* nondeterminism described below.

C. Contributions: Detecting and Debugging Nondeterminism

This paper proposes (1) a number of formal definitions of types of nondeterminism and (2) an implementation, based on these definitions, for detecting and debugging nondeterminism in property-based testing. 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 nondeterminism, and, most importantly, detection and effective debugging for unexpected nondeterminism that is, itself, a bug. We propose modifications to the widely used delta-debugging algorithm in order to better handle nondeterminism as a test property, and describe an implementation of our approach in a Python testing tool that is used on real-world projects. Finally, we use a set of case studies of real, widely-used Python projects to demonstrate the utility of our ideas.

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 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* [30], [16]. 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 [30], [42] and POSIX file systems [17], [14], 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 method 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.

One application of vertical determinism, then, is to identify idempotent operations in an automated test generation harness, and simply automatically retry all such operations, checking that the result is unchanged. The overhead of vertical nondeterminism detection will generally be much lower than that for horizontal nondeterminism (horizontal checks must re-execute an entire test; vertical checks only re-execute proportional to the number of idempotent operations performed). However, identifying idempotent operations is a fairly serious specification burden. Are there instances where a tool can automatically identify a limited kind of idempotent behavior?

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 Mac OS High Sierra root exploit (CVE-2017-13872 [34]) exhibited failure nondeterminism. Attempting to login with the root account with an empty password appeared to fail, then, on a repeated try, succeeded [33], [24].

Many library APIs are largely failure deterministic. For instance, if we exclude actual I/O errors, most POSIX file system calls either succeed or do nothing (interfaces that may only partially succeed, such as `read` tend to explicitly return the degree of success rather than signaling an error on partial success). In fact, the design of the POSIX `write` call is carefully crafted to largely maintain failure determinism. If the failure mode is one that can be anticipated, such as insufficient space to write all bytes requested (but some bytes can be written), the call returns the actual bytes written, and does not set a failure code.

In languages with a clear mechanism for expressing failure of a call (e.g., exceptions in Python and Java, or Option/Result types in Rust, Haskell, and ML), failure determinism can be automatically checked. Moreover, the expected overhead should be even lower than the general overhead for vertical determinism, in that we can expect most failing operations to be fast (since in test generation, failure is usually due to invalid parameters).

C. Nondeterminism and Delta-Debugging

Delta-debugging [44][1] 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 horizontal nondeterminism detection, delta-debugging also tends to *change the probability of nondeterministic behavior*; this can be both harmful and beneficial. The behavior is part of a more general (and, to our knowledge, not previously investigated) issue: using delta-debugging to minimize a test with respect to a predicate that only holds with a certain probability (the predicate itself is nondeterministic). Note that while most descriptions of delta-debugging discuss reducing a test with respect to *test failure* (hence the term "delta-*debugging*") we adopt the wider view of delta-debugging as reducing the length of a test while maintaining that an arbitrary predicate remains true of the shorter versions of the test. The traditional predicate is "the test fails" but other predicates, especially those related to code-coverage, can also be useful [12], [11], [2]. Our contribution here is to consider what happens when the predicate to be evaluated is not deterministic. "The test behaves flakily" is a simple, important, example of such a predicate.

[1]We use delta-debugging to stand for all test reduction algorithms, even those [39], [20] that do not use the exact binary-search that distinguishes delta-debugging *per se*; the differences are immaterial for our purposes.

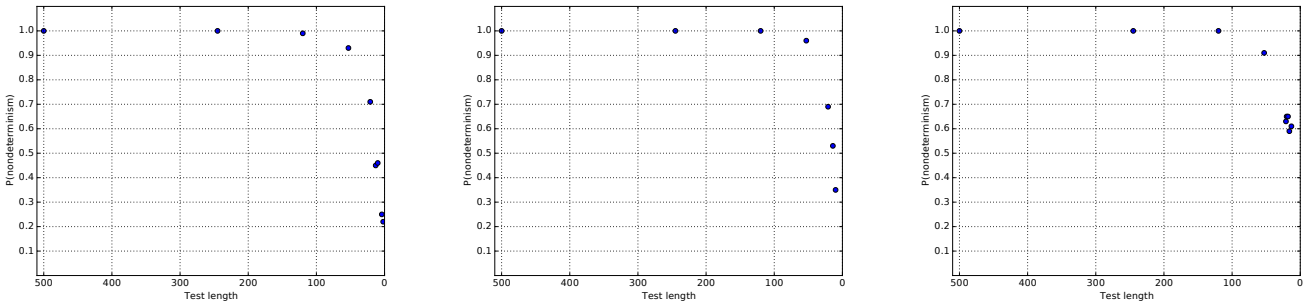
In *monotonic* probabilistic delta-debugging, removing a part of a test cannot increase the probability of the predicate holding. This is fairly common. In these cases, when we use a reduction algorithm to reduce t to r, $P(p(r) \leq P(p(t))$. In *non-monotonic* cases, however, there is no such upper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



(a) Standard delta-debugging (b) P=0.5, 100 samples (c) P=0.5, 10 samples, 10 replications

Fig. 2: Delta-debugging of the same `redis-py` test

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 (the test may be over before a "time-bomb" set by an operation goes off, for example), and second, if some test operations introduce a small probability of nondeterminism, and only many repetitions of those operations make the probability large, small tests obviously, on average, have fewer instances of the problematic operations.

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 probabilities observed in predicate evaluations suddenly 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 [43] will wish to change its C code for test reduction). Ideally, the solution should be implementable simply by modifying the predicate 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", 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. The probability calculations for such comparisons are relatively simple, but in real testing are usually not very useful, since the true probability 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.

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.

To make the basic concepts, more clear, Figures 2a-2c graphically show the interplay of delta-debugging and non-deterministic predicates for a simple example. In the example, tests consist of a sequence of operations that behave nondeterministically with (independent) probabilities of 0.01, 0.05, and 0.10, respectively, storing the value of the operation one of five array locations. There is also an operation that clears out an array location, making it possible to store another (possibly nondeterministic) value in that array slot. An example test sequence would be:

```
array[2] = op01();
array[3] = op05();
clear(array[2]);
array[2] = op10();
```

The probability that this test behaves nondeterministically is 0.84645, the result of the simple calculation $(1-0.01)\times(1-0.05)\times(1-0.10)$. Figure 2a shows one run of delta-debugging on a test of length 500. The initial test is essentially guaranteed to behave nondeterministically; delta-debugging only checks that it *can* behave nondeterministically, and so, in the course of reducing the test length to a test with only two operations, the delta-debugging algorithm also reduces the probability of nondeterminism to only about 20%. If we use a predicate that "forces" the test to behave nondeterministically at least half the time, by sampling the predicate value 100 times and only returning true when at least 50 of the evaluations report true, we see the behavior in Figure 2b: the final test is slightly longer, but still falls well short of our target of exhibiting nondeterminism 50% of the time. Finally, Figure 2c shows what happens if we use the same target of 50% nondeterminism, but use only 10 samples, with 10 replications (N = 10, M = 10): the test is not much longer than in Figure 2b, but the probability of nondeterminism is above our target value, close to 60% (and, as a bonus, fewer test executions are required on average for each check of the predicate). Note that this example is purely monotonic; the pattern of probability changes in non-monotonic settings can be even more difficult to predict and control, but the principle of forcing a probability by sampling and replication still holds.

Finally, we note that this problem is not limited to debugging nondeterminism/flakiness itself. In some settings, we have observed delta-debugging taking a test that had such a high probability of failure we were unaware it was "flaky" (only hundreds of repeated executions showed there was indeed a small chance of the test passing) and transforming it into a test that only failed once in every 10 to 20 executions. Delta-debugging, without taking potential nondeterminism into account, can act as a flakiness magnifier, with unfortunate consequences for debugging. Multiple evaluations, with replications, are our proposed solution.

III. Implementation

A. Sources of Nondeterminism in Practice

In order to implement a practical nondeterminism-detection tool, it is important to consider the sources of nondeterminism. Most of these sources can, in principle, be detected simply by re-running a test twice and comparing results, either at all visible values or at the final state. For example, if some library call uses a random number generator (perhaps in a stochastic algorithm), the value should be different on a second run. Timing dependencies may change on a second execution, especially if a delay is added between steps of the test (a more principled version of the unfortunate tradition of testing concurrent code by adding ad-hoc `sleep` statements). However, simply running a test twice (possibly with a delay between steps) does not suffice in all cases.

1) Process-Based Nondeterminism: Some sources of nondeterminism unfortunately require executing a test in a new process environment, because the source is inherently tied to the process in which code runs; simply re-executing a test in the same process will not reveal the problem.

Address Space Layout Randomization (ASLR) [27] is probably the most important source of nondeterminism that arises (only) from change in process. ASLR scrambles the layout of memory of the process in which an executable runs in order to make it harder to exploit memory-safety vulnerabilities in code. As a side effect, it means that a test that, in some circumstances, causes a crash, may at other times not fail at all. Even if no inputs produce a crash, however, memory errors may produce variation in values that are overwritten or arise from uninitialized memory, and thus be detectable as nondeterminism.

Another example of process-based nondeterminism was discovered by numerous Python developers when Python version 3.3 introduced automatic random salting of hashes on a per-process basis (a new salt chosen for each Python process), in order to mitigate hash-based denial of service attacks [35]. Until version 3.6 this not only resulted in changes in exact hash values, but in the order of iteration on dictionaries[2]. While relatively few programs rely on exact hash values, many implicitly relied on them in that they only functioned correctly with a predictable order for dictionary iteration. Testing Python code for nondeterminism based on this hash seed requires running in a new process: Python does not allow changing the salt, since changing hash values on-the-fly would break all existing dictionaries and other structures relying on hashing.

Other, less common, effects tied to process also exist. A few tests may somehow depend on their actual PID (Process ID) or at least the parent process' ID. Process change increases the likelihood that on a multi-core system a test will execute on a different CPU than in the first execution, which can have many subtle effects, mostly related to changed timing.

B. Implementing Nondeterminism Detection

We implemented our approach in the TSTL [18], [19], [21] system, a language and tool for property-based testing [26], [6] of Python code. TSTL has been used to detect (and usually fix) errors in a number of widely used Python libraries, the Python implementation itself, the Solidity compiler and a Solidity static analysis tool, and Mac OS [20]. Because TSTL supports differential testing [21], 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 considerable effort on the part of the user to express which values are checked for equivalence, and does not (without a great deal of effort) support injecting timing differences, or re-executing in a new process, in checking for nondeterminism.

We therefore instead made nondeterminism detection a first-class property in TSTL, using TSTL's existing notation

[2]In 3.6, the default dictionary implementation became an ordered dictionary with consistent iteration, though actual hashes remained nondeterministic.

TABLE I: TSTL Method Calls for Nondeterminism Detection

Name	Purpose
`nondeterministic`	returns True if test exhibits final state nondeterminism in k tries with delay d, optionally only over pools in $\mathcal{P}$
`stepNondeterministic`	as above, except checks all visible values (compares state at every step)
`processNondeterministic`	as above, except runs test provided in subprocess and compares non-opaque output values
`P`	returns probability that a predicate holds for a test (with optional number of sample to use in estimate)
`forceP`	wraps a predicate for test reduction, checking that it has desired probability of holding, using sample size and replications parameters

TABLE II: TSTL Command Line Options for Nondeterminism Detection

Name	Tool(s)	Purpose
`--checkDeterminism`	`rt`, `reduce`	Checks tests for deterministic behavior (visible value in rt, final value in reduce)
`--checkProcessDeterminism`	`rt`, `reduce`	Checks determinism using more expensive process-changed execution
`--determinismTries INT`	`rt`, `reduce`	Number of times to repeat test checking for nondeterministic behavior
`--determinismDelay FLOAT`	`rt`, `reduce`	Delay between steps when re-executing to check for determinism
`--checkFailureDeterminism`	tstl compiler	Produce a harness that checks determinism on all failing actions
`--probability P`	`reduce`	force property with respect to which test is being reduce to hold with probability P
`--samples S`	`reduce`	use S samples when checking predicate probability
`--replications R`	`reduce`	use R replications when checking predicate probability

for marking some types of values as *opaque* (not usefully compared for equality), used in automatic abstraction-based testing. In TSTL, a test is a sequence of *actions*; parameter values to method and function calls of the Software Under Test (SUT) are stored in finite-sized variable *pools*. A TSTL test is (essentially) a sequence of assignments to pool values, and method/function calls using pool values (including method calls on pool objects). This formalism is a widely used and efficient way to represent unit tests [36], [3]. In TSTL, *visible value determinism* is based on comparing the values of *all pool variables* after each test action (pool values other than the one assigned to must also be compared, since a common source of nondeterminism is when a call results in a change to a value passed as a parameter, or even due to a shared reference held by two values). TSTL omits comparison only of pools marked as `OPAQUE` in the TSTL language test harness (and values that do not support equality checking). *Final State Determinism* simply performs the same comparison, but only on the final values of all pool variables (or a set of designated pools) after a test has finished executing.

Tables I and II show, respectively, the primary additions to the TSTL Python API and the TSTL command line tools. The TSTL test generation tool is called `rt`, since the default mode is a fast pure Random Tester, and `reduce` is TSTL's tool for delta-debugging and test normalization (more aggressive reduction that also tends to ease debugging and test triage by making tests failing due to the same fault more similar [15]). With these additions, developers of TSTL-based testing tools, or developers using TSTL tools to test code can easily add nondeterminism to the set of properties checked. It is even possible to write a TSTL harness that automatically checks itself for nondeterminism, even if the random tester is not run with `--checkDeterminism`, simply by adding:

```
property: not self.nondeterministic(self.test())
```

to the harness; in fact, by removing the `not`, a test harness can even specify that behavior should not be deterministic, a property of possible use in some security-related libraries.

First-class vertical nondeterminism checks are currently limited to failures: `--checkFailureDeterminism` causes the TSTL compiler to emit a harness where every action that causes an expected exception to be raised is repeated to ensure the same exception is still raised. Users can also express that any action should exhibit vertical determinism by wrapping the action in a function that checks for failure and calls the code again.

Unlike the other additions, explicitly designed for nondeterminism detection, the `P` query, `forceP` wrapper, and `--probability`, `--samples`, and `--replications` reducer options are general tools for use with nondeterministic predicates in delta-debugging. Of course, nondeterminism and flakiness are the most common probabilistic test properties we are aware of, but in theory these tools can be used for other probabilistic debugging problems (e.g., if the property of interest is that a value is above a certain threshold in 90% of test executions, these TSTL additions can be used to find a small test where this property fails to hold). In addition to the basic probabilistic reduction issues discussed above, actual implementation of test reduction for nondeterminism introduces some subtle issues. By default, for example, TSTL automatically removed non-enabled actions from tests during reduction, because these are never actually executed. However, in timing based nondeterminism, such steps may be essential to causing the nondeterminism to appear (we had to turn off this feature for the `redis-py` example discussed below).

Because TSTL has a first-class interface to afl [43], 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. Equally importantly, since there is substantial overhead in nondeterminism detection, afl can be used to predict whether a program has potential nondeterminism; if the afl *stability* statistic (see the afl documentation for details [43]) is lower than 100%, it may indicate a problem. Because of the idiosyncrasies of afl instrumentation and process behavior, this is not always a reliable guide, but it is a very low cost indicator produced as a by-product of fuzzing.

IV. Experimental Evaluation

In order to evaluate our approach and implementation, we applied nondeterminism detection to three real world examples. The primary points we wanted to explore were (1) whether our approach was able to reliably detect actual nondeterminism, (2) whether the overhead of nondeterminism detection for real systems was acceptable, and (3) the performance of delta-debugging for real nondeterministic predicates.

A. Case Study: `redis-py`

The `redis-py` [28] module implements a widely-used Python interface to the popular Redis in-memory database, cache, and message broker [38].

Using TSTL's harness for `redis-py`, both `redis-py` and Redis itself can be tested; unfortunately, generating standalone high-coverage regression tests for `redis-py` and Redis using TSTL has proven difficult, as numerous Redis commands introduce nondeterministic behavior: thus the resulting tests are very often *flaky*. Using `tstl_afl_fuzz`, we can see that the `redis-py` has a stability of only 56.26%, a clear indicator of nondeterminism. Some of the problematic commands are obvious simply by inspection (e.g., `randomkey`, `srandmember`). And, given an understanding of Redis semantics, it is also clear that the various commands producing data with a limited lifetime (e.g., `expire`, `pexpire`) also introduce timing-based nondeterminism.

However, that a command such as `restore` takes an expiration argument is not obvious to a non-expert, nor is the behavior of `spop` which pops a random value from a set. Moreover, it is difficult to guess whether the `pipe` mechanism, which allows a large sequence of commands to be queued up in a pipe and executed all at once, introduced the potential for nondeterminism (does it execute sequentially before other command are handled, or is it in parallel with further commands). Deep experience with Redis would make these issues clear, but tests using a library are often written by those not intimately familiar with the semantics of every used library (otherwise they would not have been as likely to introduce flaky tests in the first place). This is even more true in the case of automated test generation, where the test engineer is often chosen for expertise in test generation tools, not in the domain of the software under test (SUT), and is seldom the original developer of the code.

Using the original TSTL harness, approximately *over 20%* of all `redis-py` regression tests (of length 200) generated were flaky. Using the nondeterminism checker to reduce flaky tests to a minimal set allows us to simply set up an overnight run that will, in 12 hours, produce a set of minimal flaky tests exposing the sources of nondeterminism/flakiness. In this case, the probability of flakiness is not important, just the possibility, so we let the delta-debugging create a minimal test with some non-zero (but possibly very small) observed probability of nondeterminism.

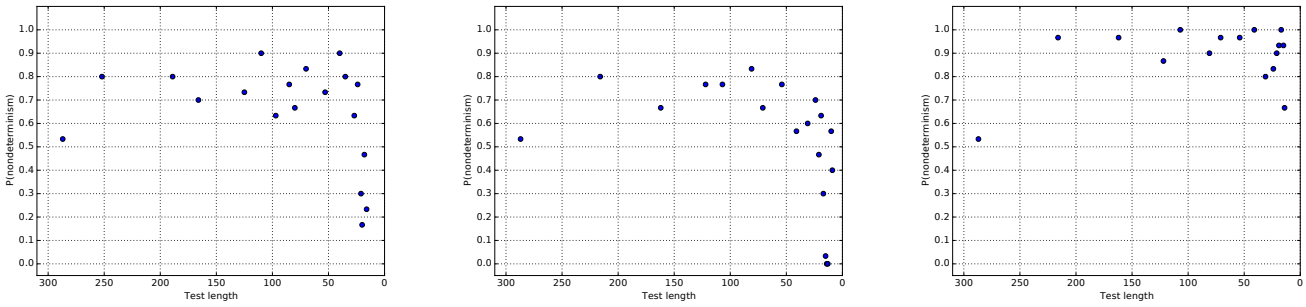
Figures 3a-3c show delta-debugging of a typical `redis-py` nondeterministic test, originally with 287 steps. The initial test behave nondeterministically about half the time. Figure 3a shows that this is a non-monotonic reduction problem, where removing steps can either increase or decrease the probability of nondeterminism. At first, unmodified delta-debugging actually improves the probability of failure, but eventually it produces a test of length 16 that only behaves nondeterministically 24% of the time. The reduction takes only 38 seconds. For the purpose of identifying commands leading to nondeterminism, this is acceptable. However, if we were actually debugging a complex nondeterminism bug in Redis itself, we might want a more reliably nondeterministic test. Using the same parameters as in Section II-C1, we see the same pattern. Simply making a predicate that "forces" the probability to remain high, with a large number of samples, does not work (Figure 3c) and requires over 2000 seconds to produce a test with an even worse probability of nondeterminism. Using 10 samples and 10 replications, on the other hand, actually improves the probability of nondeterministic behavior, and (in this case) even produces a smaller test (only 14 steps) in just over 10 minutes.

After removing all identified sources of nondeterminism (the `random-` functions, all expiration-related functionality and `restore` calls with non-zero lifetimes, and `spop`), but leaving all other suspect functionality in place (e.g. pipes), no flakiness was observed in a sample of over 2,000 length 200 tests. Moreover, because the removals were limited to actually observed sources of flakiness, the code coverage loss was minimal. Mean branch coverage, for regression tests of length 200 was reduced by less than 5 branches (a less than 1% decrease). Obviously, it is impossible to test the removed calls, but the overall coverage loss is both minimal and known, and can be made up for using specially-crafted tests (for instance, wrapping the problematic calls in a way that does not check the values, or adding a delay after an expiration to allow the data to expire). As a price to pay for the ability to produce fast-executing high-coverage full regression tests (not just tests for crashes and unexpected exceptions), this seems acceptable.

Moreover, turning off sources of expected nondeterminism makes it possible to aggressively test `redis-py` and Redis for unexpected nondeterminism arising from actual bugs. The overhead for such determinism checks, with no delay between operations, is only 20%, much lower than the expected cost of running each test twice. This is because choosing the actions in a test (and determining which actions are enabled at each step) consumes a large part of the test generator's time; running the test again and checking equality is relatively inexpensive.

B. Case Study: `datarray` Inference Algorithms

The `datarray` module [4] is a prototype implementation for numpy arrays with named axes to improve data management, developed by the Berkeley Institute for Data Science. As part of its code, it provides a set of algorithms for inference in Bayesian belief networks [40]. An earlier version of these algorithms produced nondeterministic (and in some cases incorrect) results due to dependence on the order of values in an iterator over a Python dictionary, on Python versions above 3.2, until 3.6 (see Section III-A1).



(a) Standard delta-debugging (38s) (b) P=0.5, 100 samples (2131s) (c) P=0.5, 10 samples, 10 replications (617s)

Fig. 3: Delta-debugging of the same `redis-py` nondeterministic test

Figure 4 shows TSTL code for generating inputs to the `datarray` algorithms. The `flatten_and_sort` function is needed because we care about actual differences in probability values, not simply the order of list, tuple, or dictionary items. Running this harness using TSTL's `--checkProcessDeterminism` flag required less than 10 seconds on average to produce a test exhibiting process-level nondeterminism in the `calc_marginals_sumproduct` function (the only broken algorithm). Reducing this 60 step test to a minimal test of only 6 steps, showing an extremely simple input producing the issue, required another 92 seconds. Interestingly, the way that `python-afl` is used in TSTL means that afl stability is 100%, because the `PYTHONHASHSEED` has already been chosen before the fork in afl executions.

Removing the nondeterministic call, we can see that the cost of checking for process nondeterminism, with no delay between operations, is high, a 93% slowdown. This is due to the high cost of subprocess creation and communication.

C. Vertical Determinism Case Study: `pyfakefs`

The `pyfakefs` [29] module implements a fake file system that mocks the Python file system modules, to allow Python tests both to run faster by using an in-memory file system and to make file system changes that would not be safe or easily performed using real persistent storage. Originally developed in 2006 at Google by Mike Bland, `pyfakefs` is now used in over 2,000 Python tests, inside and outside Google [29].

The TSTL harness for `pyfakefs` has been used to detect (and correct) over 50 faults. However, the testing completely relies on the existence of a reference file system implementation. One purpose of failure nondeterminism is to make it somewhat easier to perform property-based testing of complex APIs like this even without a reference implementation, or in cases where the implementations do not use the same error codes (as is common, e.g. in NASA flight software [16], [17]).

We introduced a subtle bug into `pyfakefs`, where the `remove` call checks that its target is not a directory, and returns the correct error, but still carries out the remove operation. Using `os.remove` to delete directories does not break any file system invariants, but violates the Python `os` specification (and, indirectly, the usual POSIX implementation behavior where `unlink` does not work for directories). Detecting this bug using the TSTL `pyfakefs` harness is normally impossible without using another file system as a reference. However, the fault was detected immediately, even without using a reference, when we compiled the harness with the `--checkFailureDeterminism` flag. Because vertical reduction does not require running complete tests multiple times, and does not affect the delta-debugging algorithm's performance, reducing the failing test to 3 steps required less than a second. Moreover, the overhead for the check for failure determinism in a version of the code without the `remove` error was less than 8%. Detecting the fault using a reference file system required 17% more runtime before detection, and took over twice as long to reduce the failure to a slightly longer failing test, which did not have `remove` as its final operation (since further operations are required to expose the problem).

V. Related Work

Gao et al. generally considered the question of how to make tests repeatable [9], 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 [41] 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 [7].

```
@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. 4: Complete TSTL harness for finding the hash-order bug in the datarray inference algorithms.

The problem of test nondeterminism is closely related to the (extremely important in industrial settings) issue of flaky tests [32], [25], [37], [23]. How to handle flaky tests in practice (when they cannot be avoided) is a major issue in Google-scale continuous testing [31], 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 [22], or provided large-scale empirical examination of tests from one large open source project (Apache) [25], [37]. Palomba and Zaidman [37] 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 [5] 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

Unexpected nondeterminism of software systems frustrates users, whether they be humans or (more importantly) other software systems. Nondeterminism is even more pernicious in software testing, frustrating debugging efforts ("Heisenbugs" are widely loathed [10]), and leading to the costly problem of flaky tests [32], [23].

This paper proposes a formulation of types of nondeterminism, and a practical approach to using automated test generation to detect nondeterminism, especially in the library code that underlies most systems. In addition to traditional *horizontal* nondeterminism (the phenomenon behind Heisenbugs and flaky tests), we also discuss the related concept of *vertical* nondeterminism, which is more frequently simply a software bug. We implemented our approach in the TSTL automated test generation system for Python, and demonstrated the simplicity and basic utility of the approach on three real-world examples.

As future work, we would like to make the automatic detection of values that should not be visible (e.g., count towards nondeterminism) possible: if a value (e.g., a timestamp) is always different in every run, it is likely of no interest to developers. This would make testing libraries mixing deterministic behavior and nondeterministic behavior, e.g. cryptographic libraries offering high-quality random number generators, easier for users, who would only be shown "surprising" nondeterminism. Similarly, in order to extend the utility of vertical nondeterminism detection, we would like to automatically identify usually-idempotent operations, and check them for deviation from the expected behavior. We would also like to lower the cost of checking process nondeterminism, perhaps by using methods taken in afl [43] to avoid high startup costs for test executions.

More generally, we are interested in using test decomposition [13] to more easily isolate, understand, and avoid nondeterminism is tests, and to mitigate the problem of flaky tests. Reliable detection of nondeterminism is a first step towards evaluating such pro-active flaky test mitigations.

Acknowledgments: The authors would like to thank John Micco, Jeff Listfield, and Celal Ziftci at Google, for discussion of flaky tests, Andreas Zeller and David R. MacIver for discussion of the problem of probabilistic delta-debugging, and Chris Colohan for discussion of sources of process-based nondeterminism.

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] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce. Evaluating non-adequate test-case reduction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 16–26, 2016.

[3] J. Andrews, Y. R. Zhang, and A. Groce. Comparing automated unit testing strategies. Technical Report 736, Department of Computer Science, University of Western Ontario, December 2010.

[4] Berkeley Institute for Data Science. Prototyping numpy arrays with named axes for data management. https://github.com/BIDS/datarray.

[5] 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.

[6] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

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

[8] D. Firesmith. The challenges of testing in a non-deterministic world. https://insights.sei.cmu.edu/sei_blog/2017/01/the-challenges-of-testing-in-a-non-deterministic-world.html, January 2017.

[9] 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.

[10] J. Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12, 1986.

[11] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability*. accepted for publication.

[12] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction for quick testing. In *International Conference on Software Testing, Verification and Validation*, pages 243–252, 2014.

[13] A. Groce, P. Flikkema, and J. Holmes. Towards automated composition of heterogeneous tests for cyber-physical systems. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Testing Embedded and Cyber-Physical Systems*, TECPS 2017, pages 12–15, New York, NY, USA, 2017. ACM.

[14] 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.

[15] A. Groce, J. Holmes, and K. Kellar. One test to rule them all. In *International Symposium on Software Testing and Analysis*, 2017. accepted for publication.

[16] 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.

[17] 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.

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

[19] A. Groce, J. Pinto, P. Azimi, and P. Mittal. TSTL: a language and tool for testing (demo). In *ACM International Symposium on Software Testing and Analysis*, pages 414–417, 2015.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] D. R. MacIver. Hypothesis: Test faster, fix more. http://hypothesis.works/, March 2013.

[27] H. Marco-Gisbert and I. Ripoll. On the effectiveness of full-aslr on 64-bit linux, 2014.

[28] A. McCurdy. redis-py: Python Redis Client. https://github.com/andymccurdy/redis-py.

[29] J. McGehee. pyfakefs implements a fake file system that mocks the python file system modules. https://github.com/jmcgeheeiv/pyfakefs.

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

[31] 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.

[32] 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.

[33] 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/.

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

[35] Open Source Computer Security Incident Response Team. ocert-2011-003 multiple implementations denial-of-service via hash algorithm collision. http://ocert.org/advisories/ocert-2011-003.html.

[36] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[37] 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.

[38] redislabs. Redis. https://redis.io/.

[39] 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.

[40] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[41] 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.

[42] 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.

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

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