Does It Always Do That? Practical Automatic Lightweight Nondeterminism and Flaky Test Detection and Debugging for Python Libraries

Alex Groce

School of Informatics, Computing & Cyber Systems
Northern Arizona University

Josie Holmes

School of Informatics, Computing & Cyber Systems
Northern Arizona University

Abstract

A critically important, but surprisingly neglected, aspect of system reliability is system predictability. Many software systems are implemented using mechanisms (unsafe languages, concurrency, caching, stochastic algorithms, environmental dependencies) that can introduce behavioral nondeterminism. Users of software systems, especially other software 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 series of operations. Equally importantly, even when it does not impact correctness, 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 little effort on the part of developers, can be used to extend an existing testing system to allow detection of 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. In particular, we demonstrate

Email addresses: agroce@gmail.com (Alex Groce), https://agroce.github.io (Alex Groce), josie.holmes@nau.edu (Josie Holmes)

that our proposed failure nondeterminism can improve mutation score by 6% for a strong, full differential test harness for a widely used mock file system.

Keywords: determinism, flaky tests, library testing, random testing, specification mechanisms, test case reduction

---

1. 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 [12]. 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 sometimes referred to as the dreaded "Heisenbug" [15], though the more accurate usage is "Mandelbug" [27, 8], since Heisenbugs, properly speaking, are bugs that disappear or alter behavior under instrumentation (especially in debugging). 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. [13] 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 behaves in a reliably predictable fashion; nondeterminism is frequently vexing in debugging efforts; most importantly, perhaps, nondeterminism is *often disastrous* for large-scale highly automated testing.

1.1. 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.

1.2. 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.

1.2.1. 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* [43] 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 in which the test is run[1]. 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.

[1]Again, in some sense the result *is* clearly determined by the environment in which the test is run, but from the point of view of a developer or test engineer, the differences in environment are invisible and unknown.

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.

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. Our focus, however, unlike all other tools and approaches we are aware of, is on detecting the *sources* of flaky-ness *before* they propagate to cause actual test failure.

1.3. Contributions: Detecting and Debugging Nondeterminism

This paper proposes (1) a number of formal definitions of types of nondeterminism (*horizontal* and *vertical*) (see Section 2.4) and (2) an implementation, based on these definitions, for detecting and debugging nondeterminism in property-based testing. The implementation is based on an approach where (3) horizontal determinism is considered as a kind of *reflexive differential testing* (4) vertical determinism is specialized to the common case of *failure determinism*, and (5) in both cases the formalism is made practical by using the *value pool* model of unit tests to define a useful granularity for nondeterminism detection (see Section 4.2). We further propose modifications to the widely used delta-debugging algorithm in order to better handle nondeterminism as a test property.

The underlying idea in this paper is that most approaches to nondeterminism/flaky test detection and debugging rely either on 1) identifying potential sources of nondeterminism such as test inter-dependence [32] or certain code smells [48] or 2) use a heuristic approach to mark failures as likely flaky, such as that taken by DeFlaker [4]. In the real world, people usually detect flaky tests by the most obvious heuristic of all: they observe a test both fail and pass for the same code version. In a sense, even DeFlaker relies on this method, but avoids having to observe a flaky test actually passing, by inferring that if a test was previously passing, and executes no

changed code, its failure after a code change *must* be due to flakiness/nondeterminism. Our approach to the problem is fundamentally different, and essentially orthogonal; we concentrate on the detection and avoidance of nondeterminism/flakiness using *automated test generation*, and make it possible for a developer or test engineer to detect when important values generated in automated testing differ without a change in the test itself or the code under test. While some of these changes may be harmless, and unable to propagate to cause test failures, they are potential sources of flaky behavior. To our knowledge, this is the first approach to the problem that allows detection of (potential sources of) flakiness, based on real divergence in behavior, without having to observe a test actually fail due to nondeterminism.

This approach 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 sources of nondeterminism that might produce problems for manually-constructed tests, and, most importantly, detection and effective debugging for unexpected nondeterminism that is, itself, a bug. Additonally, by identifying (using automated testing) library behaviors that are nondeterministic, our approach should make it much easier to flag statements in existing human-written unit tests that may produce flaky behavior: simply scan the code for calls to functions identified as sources of nondeterminism.

We use a set of case studies. including real-world, widely-used, Python libraries to demonstrate the utility of our ideas. Failure determinism directly produces a more than 5% improvement in mutation score for an already highly effective automated test generation tool for a file system, with no additional specification or test design burden.

2. Practical Lightweight Nondeterminism Detection

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

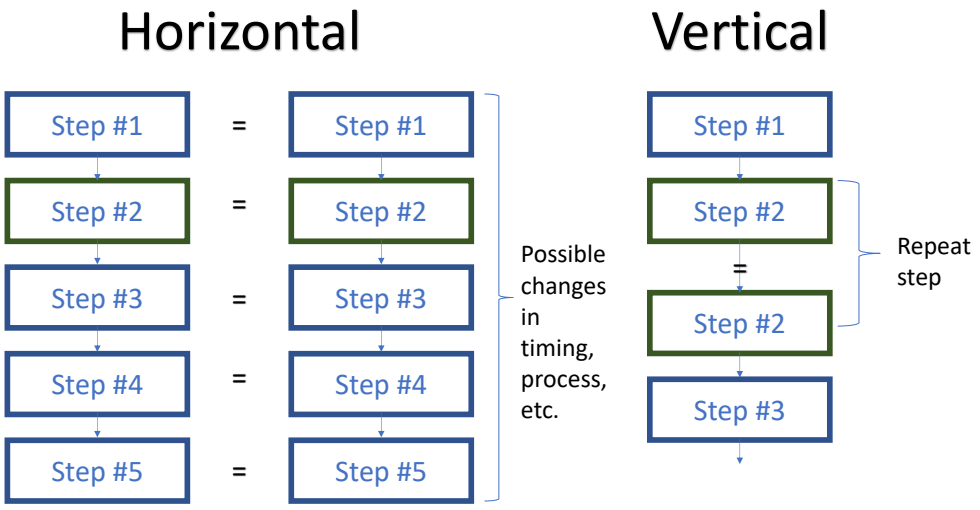


Figure 1: Types of determinism

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.

2.1. Horizontal Determinism

2.1.1. Determinism and Reflexive Differential Testing

Horizontal determinism can be best understood by thinking of nondeterminism detection as an unusual kind of *differential testing* [41, 22]. 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 [41, 55] and POSIX file systems [23, 19], 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.

2.1.2. 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.

2.1.3. 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.

2.2. 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 call to that function.

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?

2.2.1. 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 [45]) exhibited failure nondeterminism. Attempting to login with the root account with an empty password appeared to fail, then, on a repeated try, succeeded [44, 34].

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). Checking equivalence of the full observable state, however, is still expensive, and requires defining the observable components of a state. A more lightweight, but (as we will show) still quite powerful way of checking for failure nondeteterminism is to repeat the failing operation, and check if it still fails. If not, this is a sign that the operation is not deterministic with respect to failure. Note that this more restrictive notion of failure nondeterminism captures such issues as the Apple login bug.

2.3. Nondeterminism vs. Flakiness

A key concept in this paper is that flaky tests (tests that sometimes pass and sometimes fail, recall) exist due to underlying nondeterministic behavior. Our interest is in identifying such nondeterministic behavior, not in addressing flakiness *per se*. Of course, the concepts are closely related. We make "did not behave nondeterministically" an additional *property* that a test can fail. Due to the nature of (horizontal) nondeterminism, such failure is almost always going to be inherently flaky — sometimes a test will behave nondeterministically, and sometimes it will not. But the idea is a separate one from flakiness; in order to show horizontal nondeterminism, we need only see a change in values in a test, not in the disposition (failed/passed) of the test. It may be that, without the additional property of determinism, the change cannot actually cause failure, and thus flakiness. The real relationship between our nondeterminism and flakiness is that tests (automatically generated or manually constructued) that rely on code that can be shown to exhibit horizontal nondeterminism contain a potential source of flakiness.

If the code behaves nondeterministically, and the values that are observed to change can flow to influence the outcome of the test, flakiness may well result.

Vertical nondeterminism, in fact, is potentially *completely* unrelated to flakiness, in that some bugs (improper handling of hardware errors in a file system, for example) produce *reliably* "nondeterministic" (from a software library users' point of view) behavior. Vertical nondeterminism may rely on "real" nondeterminism that could produce a flaky test, but it does not have to do so. In other words, vertical nondeterminism is often not a problem for testing (flakiness), but simply a bug in the behavior of a software system, where an implicit specification on failed operations is consistently violated under certain conditions. The behavior is "nondeterministic" from a caller's point of view, however, in that there is an expectation that a call will always "do the same thing, under the same circumstances" which is the *conceptual* common factor in all kinds of expected determinism.

2.4. Formal Definitions

We can formally distinguish the types of nondeterminism by using a variant of a labeled transition system (LTS) as our model:

- S is a set of states.
- V is a set of *observable* states, where $|V| \leq |S|$.
- $v : S \to V$ is a total function that, given a state, maps it into the set of observable states. Every state has an observable component, which may be the complete state, or only an aspect of the full state; we even allow V to contain a single element, in which case no two states in S can be distinguished by an observer.
- $I \subseteq S$ is a set of initial states.
- A is a set of *actions*.
- $T \subseteq S \times A \times S$ is a transition relation.

We assume that the underlying behavior of a system may be deterministic, or nondeterministic, in the strict sense, by allowing for a transition relation that *may* be a function of $S \times A$. A *trace* t is a finite sequence $t = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ where $s_0 \in I$ and $\forall i < n.(s_i, \alpha_i, s_{i+1}) \in T$.

A pair of traces (t_1, t_2) where $t_1 = s_0^1 \xrightarrow{\alpha_0^1} s_1^1 \xrightarrow{\alpha_1^1} \ldots \xrightarrow{\alpha_{n-1}^1} s_n^1$ and $t_2 = s_0^2 \xrightarrow{\alpha_0^2} s_1 \xrightarrow{\alpha_1^2} \ldots \xrightarrow{\alpha_{n-1}^2} s_n^2$ are said to show *visible value nondeterminism* if $\exists i > 0$ such that: $\forall j < i . \alpha_j^1 = \alpha_j^2 \wedge v(s_j^1) = v(s_j^2)$ but $v(s_i^1) \neq v(s_i^2)$. *Final state nondeterminism* is defined in the same way, except with the restriction that $i = n$. A pair of traces may exhibit visible value nondeterminism but not final state nondeterminism, but any pair of traces exhibiting final state nondeterminism also demonstrate visible value nondeterminism.

Vertical nondeterminism, in contrast, is demonstrated by a single trace t, not a pair of traces. To define vertical nondeterminism with respect to idempotency, we can define $\mathcal{I} \subseteq A$, the set of *idempotent* actions. A trace $ts_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ shows *vertical nondeterminism with respect to idempotent operations* if $\exists i < n$ such that $\alpha_i \in \mathcal{I} \wedge \alpha_i = \alpha_{i+1}$ and $v(s_{i+1}) \neq v(s_{i+2})$. Note that vertical nondeterminism may exhibit only after a sequence of more than two successive applications of a (supposedly) idempotent operation; the definition only requires that after *some* sequence of actions, possibly with $\alpha_{i-1} = \alpha_i$, the visible state changes.

To define failure nondeterminism, we extend the transition relation to include a notion of *failure*: $T \subseteq S \times A \times S \times \texttt{bool}$, where the boolean indicates whether the action *failed*. A trace $t = s_0 \xrightarrow{\alpha_0} (s_1, F_0) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} (s_n, F_n)$ shows *failure nondeterminism* if $\exists i$ such that F_{i+1} (that is, α_i *fails*) and either (1) $v(s_i) \neq v(s_{i+1})$ (the visible state changes) or (2) $\alpha_i = \alpha_{i+1} \wedge \neg F_{i+2}$ (the same action is repeated, but this time does not fail). The second possibility offers a *lightweight* approach to checking for failure nondeterminism. Unlike all of the other ways to demonstrate nondeterminism we consider, this second type of failure nondeterminism does not require the use of v, or even the notion of observable components of a state, or (for that matter) any comparison of states. It only requires that we be able to say whether an operation failed.

A further point to emphasize is that while horizontal determinism requires two traces, it does not require a test generation algorithm to produce pairs of traces, which would require substantial modification of most such tools (e.g., random testers or fuzzers, and even model checkers). In general, testing tools do not produce *traces*, they produce *action sequences*. The two traces used to show nondeterminism, by definition, have the same action sequences, which means that a testing tool only needs to search for an action sequence which, when executed twice, produces a t_1 and t_2 meeting the requirements for showing nondeterminism — non-equivalence of $v(s)$ for some s.

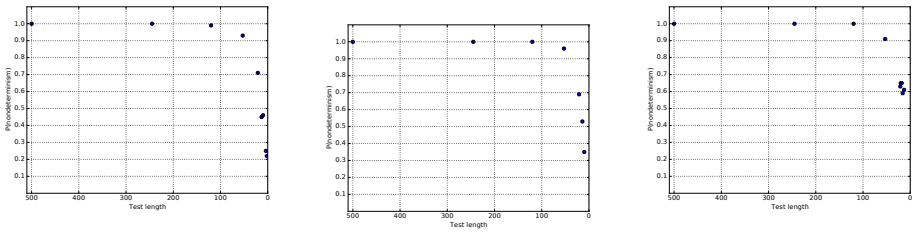
3. Nondeterminism and Delta-Debugging

Delta-debugging [57][2] is a widely used method for reducing the size of failing tests, making them easier to understand and debug. The core idea of delta-debugging is to take a test that satisfies some property (usually the property is "it fails in a particular way") and produce a smaller test that satisfies the same property. Delta-debugging, as proposed by Zeller and Hildebrandt, relies on a modified binary search to produce this smaller test, though the following discussion relies only on the general structure of delta-debugging, and also applies to the greedy algorithms used to reduce failing tests in some work [51, 20].

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 [17, 16, 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.

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,

[2]We use delta-debugging to stand for all test reduction algorithms, even those [51, 26] that do not use the exact binary-search that distinguishes delta-debugging *per se*; the differences are immaterial for our purposes.



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

Figure 2: Delta-debugging of the same simple test

$P(pred(r)) \leq P(pred(t))$. 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 basically 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).

3.1. "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 (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.

3.2. Using Replications to Mitigate "Publication Bias"

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 [56] 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 trade-off 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 nondeterministic 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.

4. Implementation

4.1. 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). Most nondeterministic behavior in tests is probably attributable to a such effects that can vary with a repeated execution, in the same (approximate) enviroment, of the same test. The two broad classes of nondeterminism that can usually be detected (with some probability) by simply executing a test twice in succession are:

- **External Environment:** This class of nondeterminism arises when a test's behavior depends on factors outside the program under test, which are subject to uncontrolled variance. Calling `random()` in Python is an obvious example; the call is almost guaranteed to return a different value each time it is called in a test, unless we explicitly re-seed the random number generator at the beginning of each test. Similarly, calls to `time` functions will usually return different values on different runs, and even if only elapsed time, rather than absolute, is used to influence behavior, subtle variances in timing are to be expected due to system load, I/O, and other external factors. Finally, in distributed system testing, the environment is often remote servers (or clients) that may be unavailable, slower than expected, or subject to faulty behavior for reasons not under the control of the test.

- **Concurrency:** Perhaps the second most common cause for nondeterminism in tests is when an algorithm is implemented using concurrency,

in the (incorrect) beleif that the result of the algorithm remains deterministic. E.g., consider a multi-threaded implementation of a search algorithm, that returns the location of an item in an unsorted list. If the list may contain duplicates, and a developer has not accounted for this possibility, the algorithm may return different locations of an item for the same list, depending on the scheduling of threads. Arguably, this can be subsumed under the first class, with the behavior of the processor/scheduler as the "enviromental" factor; relying on scheduling is, in a sense, equivalent to calling `random` or `time`.

Lam et al. [31] note that flaky tests not due to order dependence are generally due to "concurrency, timeouts, network/IO, etc." a description that is well accounted for by our two primary sources, and most of which can be detected, at least in principle, by multiple runs of a test within the same testing process. However, simply running a test twice in the same process (possibly with a delay between steps) does not suffice in all cases.

4.1.1. 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) [38] 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 [46]. Until version 3.6 this not only resulted in changes in exact hash values,

but in the order of iteration on dictionaries[3]. 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.

One common cause of flaky tests is *test order dependence* [31]. This is less of a concern in our setting, since many problems due to test order are easily exposed in random testing, since the order-dependence-inducing operations will seldom usually happen in the same order in randomly generated tests. For example, consider the case where a test usually run "early" initializes an uninitialized value in a system, and a later test will only fail if it somehow is run before the initializing test. In random testing, the order of initialization/access is likely to be random, and some run of a random tester is likely to expose the problem as a (deterministic, not flaky) failure.

However, for cases where there is an order dependence that is hard to translate into an actual bug, using a fresh process may expose such dependencies — the "environment" of the parent process will have the, e.g., flag set or value initialized, but the new process will not.

4.2. Pools and Visible State

While the formal definitions of nondeterminism proposed in Section 2.4 offer a framework for checking for nondeterministic behavior, they leave unspecified the key notions of *state*, *visible state*, and *actions*. For example, we could identify the state and visible state of a system as identical with the full set of memory locations accessed by that system, and actions with the individual processor instructions executed. However, this definition is highly impractical for two reasons. First, it is extremely inefficient to compare multiple executions for full-memory-state equivalence at every instruction step.

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

Second, and even more importantly, as noted in the introduction, very few programs will behave deterministically when considered at this low level, unless the full state of the operating system, processor, hardware devices, etc. are all taken as controlled inputs.

What is a practical level of granularity for (visible) states and actions that can be efficiently checked, and matches (for the most part) developer and tester intuitions about what should behave deterministically?

Andrews et al. formalized a value-pool-based model of unit tests that is widely used in actual testing tools, including Randoop [47], showing that all Java unit tests (and, more generally, unit tests in similar languages) can be represented in a *canonical* form, where a test consists of three parts:

1. A declaration of a set of array variables, referred to as *value pools*, e.g., `int [] intVP = new int[3];`.

2. A set of assignments of constant values to elements of primitive type value pools, e.g. `intVP[2] = 1;`.

3. A part in which all statements are assignments of calls of a method to a value pool, with all arguments to the method also taken from a value pool, e.g. `intVP[2] = fooObjectVP[2].bar(intVP[0], intVP[1]);` (these are referred to as *array-canonical* statements).

Given such a form for unit tests, there is an obvious mapping to states and actions in our formalism for nondeterminism: the state is the full state of the system, but visible state is restricted to *the values stored in the value pools*. Actions are the array-canonical statements that call code under test to modify the pools. We can ignore constant assignments and declarations, since (assuming the language implementation itself is basically deterministic) these cannot introduce nondeterminism. Because assertions of correctness and *what a test actually does* depend on the values in value pools, it seems likely that developers and testers expect determinism in unit tests, at the granularity of value pool assignments.

4.3. A Brief Introduction to TSTL

We implemented our approach in the TSTL [24, 25, 29] system, an open-source language and tool for property-based testing [37, 7] of Python code. TSTL, including full support for our nondeterminism detection methods, is available at `https://github.com/agroce/tstl`, and the `examples` directory

Table 1: 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 2: 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

```
pool: <int> 5
pool: <l> 5
pool: <s> 5

<int> := <[1..20]>
<l> := []
<l> := list(<s>)
<s> := set()
<s> := set(<l>)

<l>.append(<int>)
{ValueError} <l>.remove(<int>)
<s>.add(<int>)
{KeyError} <s>.remove(<int>)

property: len(<l>) >= len(set(<l,1>))
property: len(<s>) == len(list(<s,1>))
```

Figure 3: Simple TSTL harness testing lists and sets

contains the source code needed to duplicate all of our experiments[4]; TSTL can also be installed using Python's `pip` tool, by typing `pip install tstl`; you will need to clone the github repository to have access to the case studies used in the paper, however.

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 [26].

In TSTL, a test consists of a sequence of *actions*, where actions execute arbitrary Python code, but are expected to work by modifying the *state* of a test, which is stored in a fixed set of pools containing Python objects. TSTL's approach essentially follows the pool-based canonical form defined by Andrews et al., but relaxes many of its restrictions (e.g., a constant value can be used in a method call without first assigning it to a pool). The TSTL actions defined by a test harness correspond to the set A of actions in Section 2.4, and transitions between states result from executing out the Python code in an action; these actions are, roughly, the array-canonical statements of the value pool formalism. Figure 3 shows a very simple TSTL harness that tests Python's built-in `set` and `list` types. The harness defines the (visible) state of a test as consisting of three named pools of Python objects, `int`, `l`, and `s`. These pools have no inherent associated type, though a type can be

[4]In some cases, you will also need to install an appropriate version of the tested software, by doing, e.g., `pip install pyfakefs`.

annotated as an additional constraint that TSTL will check holds during test execution. Initially all the pools hold the special Python value `None` indicating they have not been initialized. In terms of our formalism, $I = \{$`[<int> = [None, None, None, None, None]`, `<l> = [None, None, None, None, None]`, `<s> = [None, None, None, None, None]]`$\}$. Actions with the `:=` assignment operator initialize pools. For example, the line of code `l := []` defines a set of 5 actions, each of which initializes a different member of the list pool `l` to an empty list — `l0 = []` ... `l4 = []`. Actions without a `:=` require that pools used in them already be initialized. TSTL provides constructs like `<[1..20]>` for choosing values in a range for use in a test, and allows a user to specify which exceptions an action may raise without causing the test to fail (as in `{KeyError} <s>.remove(<int>)`).

Compiling this harness using the TSTL compiler produces a Python module that allows test generation. If we use the random testing tool provided by TSTL, `tstl_rt`, it will randomly generate valid sequences of actions, and report no problems. If we change the first property to read `property: len(<l>) == len(set(<l,1>))`, however, TSTL will report a failing test, shown in Figure 4.

4.4. Implementing Horizontal Nondeterminism Detection

Because TSTL supports differential testing [29], 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 horizontal nondeterminism detection a first-class property in TSTL, using TSTL's existing notation for marking some types of values as *opaque* (not usefully compared for equality), used in automatic abstraction-based testing. Recall that in TSTL, a 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 [47, 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, changes a shared reference

```
int0 = 1                                                                    # STEP 0
ACTION: int0 = 1
int0 = None : <type 'NoneType'>
=> int0 = 1 : <type 'int'>
================================================
l0 = []                                                                     # STEP 1
ACTION: l0 = []
l0 = None : <type 'NoneType'>
=> l0 = [] : <type 'list'>
================================================
l0.append(int0)                                                             # STEP 2
ACTION: l0.append(int0)
int0 = 1 : <type 'int'>
l0 = [] : <type 'list'>
=> l0 = [1] : <type 'list'>
================================================
l0.append(int0)                                                             # STEP 3
ACTION: l0.append(int0)
int0 = 1 : <type 'int'>
l0 = [1] : <type 'list'>
=> l0 = [1, 1] : <type 'list'>
================================================
ERROR: (<type 'exceptions.AssertionError'>,
  AssertionError(), <traceback object at 0x1032b83f8>)
TRACEBACK:
  File "/Users/adg326/scratch/pyexample/sut.py",
  line 13695, in check
    assert len(self.p_l[0]) == len(set(self.p_l[0]))
```

Figure 4: A failing test (showing that converting a list to a set may not preserve the size of the object).

held by two values). By default V as defined in Section 2.4 is just the set of all possible pool values, and v simply extracts the pool values from S. TSTL allows restriction of which state is visible by removing pools marked as `OPAQUE` in the TSTL language test harness (and values that do not support equality checking) from $v(S)$. *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 1 and 2 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 [20]). 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.

4.5. Implementing Vertical Nondeterminism Detection

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.

4.6. Tools for Nondeterministic Delta-Debugging

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).

4.7. Nondeterminism and AFL Stability

Because TSTL has a first-class interface to AFL [56], 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 [56]) 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.

5. A Simple Example

Figures 5 and 6 show a self-contained example of a Python library and test harness with both vertical and horizontal nondeterminism. The example is not meant to be realistic, but to better concretely explain the detection and debugging of nondeterminism. A few additional TSTL features need to be described. First, lines beginning with an `@` indicate raw Python code, here used to import the module to be tested. Second, `init:` indicates code that is to be executed at the beginning of each test, in order to restore the system under test to its initial state.

```
@from storage import *

init: init()

pool: <int> 5

<int> := <[-10..10]>
<int> := getNewVal()
{KeyError,ValueError} store(<int>)

property: len(contents()) == len(set(contents()))
```

Figure 5: TSTL harness for Python library with (independent) vertical and horizonal nondeterminisms

5.1. Horizontal Nondeterminism

If we compile and test this code without using any of the nondeterminism-detection functionality, TSTL will not report any failing tests. The property (avoidance of duplicate entries in storage) holds:

```
$ tstl storage.tstl
...
$ tstl_rt
Random testing using config=Config...
storage: TEST #1396 STEP #0 (0:00:30.000188) Mon Feb
25 11:13:18 2019 [ 15 stmts 22 branches ] (no cov+ for
1391 tests) 139500 TOTAL ACTIONS (4649.97/s) (test
0.0/s) 2 cov+ tests
```

TSTL reports information on the testing in a status line, here showing that it has performed 1,396 tests, with no failures, and two tests that increased code coverage. TSTL can be run in a mode that stores minimized versions of the tests that obtained new coverage, for use as, e.g., regressions:

```
$ tstl storage.tstl
...
$ tstl_rt --quickTests
```

This will result in generation of two[5] files, `quick.0.test` and `quick.1.test`. Sometimes, TSTL will fail at this point with an internal error (`REDUCED TEST DOES NOT PRESERVE COVERAGE`), due to the attempt to reduce one of the coverage-obtaining tests losing coverage, which is a consequence of horizontal nondeterminism. However, TSTL will often produce test files such

[5]In some rare cases, TSTL may generate one test that achieves complete code coverage, but this will only happen rarely here.

```
import random
storage = []
def contents():
    global storage
    return storage
def init():
    global storage
    storage = []
def store(n):
    global storage
    if n in storage:
       raise KeyError
    storage.append(n)
    if n < 0:
       raise ValueError
def getNewVal():
    global storage
    if len(storage) >= 1:
        bot = min(storage)
        top = max(storage)
    else:
        bot = -10
        top = 10
    v = random.randint(bot-1,top+1)
    while v in storage:
        v = random.randint(bot-1,top+1)
    return v
```

Figure 6: Code for Python library with (independent) vertical and horizonal nondeterminisms

as those shown in Figure 7. TSTL lets us convert these tests, which are in TSTL's internal format, into standalone Python regression tests that check the properties in the test harness and ensure that the behavior (values of pools) are the same as in the recorded run:

```
$ tstl_standalone quick.0.test quick0.py --regression
...
$ tstl_standalone quick.1.test quick1.py --regression
```

However, if we execute these Python tests, they will be very flaky; sometimes they will pass, but `quick0.py` will almost always fail, and `quick1.py` will fail about two thirds of the time. The reason, of course, is a trivial case of horizontal nondeterminism, due to the use of the `random.randint` function in `getNewValue`. The regression test will record whatever value `getNewValue` returned the when the TSTL test file was executed in order to produce the Python standalone test. The value in future executions will seldom match this value. In this case, removing the nondeterminism is easy

quick.0.test:

```
self.p_int[0] = getNewVal()
store(self.p_int[0])
store(self.p_int[0])
self.p_int[0] = -10
store(self.p_int[0])
```

quick.1.test:

```
self.p_int[0] = -8
store(self.p_int[0])
self.p_int[0] = getNewVal()
```

Figure 7: Coverage maximizing tests for Python library with (independent) vertical and horizonal nondeterminisms

(simply add a fixed random seed to the `init:` line in the harness, e.g., `init: init(); random.seed(0)`). However, in real-world testing, the source of the nondeterminism, may not be obvious, and only some regression tests may (infrequently) fail. However, if we ask the random tester to check for nondeterminism, it quickly shows us what the problem is:

```
$ tstl_rt --checkDeterminism
...
int2 = getNewVal()              # STEP 50
MISMATCH IN REPLAY VALUE:
   int2 : 7 VS. -5
TEST WAS NOT DETERMINISTIC!  WRITING FAILURE AS
ndfail.test
```

We can use TSTL's test reduction tool to produce a small test showing the problem:

```
$ tstl_reduce ndfail.test hnd.test --checkDeterminism
...
NEW LENGTH 1
int0 = getNewVal()              # STEP 0
TEST WRITTEN TO hnd.test
```

Nothing is required to produce nondeterminism other than calling `getNewVal`, and this test is more frequently nondeterministic than the original, since it calls `getNewVal` without any stored data to restrict the range of random values.

5.2. Vertical Nondeterminism

The `storage.py` program also contains an (again, unrealistically simple) example of *vertical* nondeterminism. The `store` method is supposed to 1) allow only a single copy of a value to be in the storage and 2) to disallow negative values. Unfortunately, the check for negatives is *after* the value is stored, which means that a second attempt to store the same negative value will result in a `KeyError`, not a `ValueError`. TSTL can detect this problem if we compile the harness with `--checkFailureDeterminism`, which produces a version of the testing interface that ensures failing actions (ones that raise an expected exception) always fail again if performed again without any intervening actions, and always fail with the same exception. Note that the check for the same exception is a slight extension of our formal definition of failure determinism, recording the last observed exception as an additional visible state element.

```
$ tstl storage.tstl --checkFailureDeterminsim
...
$ tstl_rt
...
FINAL VERSION OF TEST, WITH LOGGED REPLAY:
int0 = -4                                    # STEP 0
ACTION: int0 = -4
int0 = None : <type 'NoneType'>
=> int0 = -4 : <type 'int'>
================================================
store(int0)                                  # STEP 1
ACTION: store(int0)
int0 = -4 : <type 'int'>
RAISED EXPECTED EXCEPTION:
  <type 'exceptions.ValueError'>
RAISED EXCEPTION:
  <type 'exceptions.KeyError'>  ON RETRY
  ERROR: (<type 'exceptions.AssertionError'>,
  AssertionError('Failure with different exception.',),
<traceback object at 0x105e9edd0>)
```

With failure nondeterminism, we don't have to use TSTL's specialized reduction tool, since the failure does not require multiple replays of a test. The random tester itself can properly reduce the test case, since it is just a "normal" failure, exhibited by a single trace.

6. Experimental Evaluation

In order to evaluate our approach and implementation, we applied nondeterminism detection to real Python code, including some widely used systems (`redis-py` and `pyfakefs` are well-maintained, widely-used libraries).

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.

6.1. Case Study: Parallel Sorting

Our first case study is a simple implementation of a parallel merge sort in Python `https://gist.github.com/stephenmcd/39ded69946155930c347`. The merge sort and harness can be found in the TSTL GitHub repository [26], under `examples/parallelsorts`. When a sorting algorithm is implemented using concurrency, a key question to ask is whether the ordering of elements that are equal under the comparison operator is consistent across sorts of the same original sequence. This is related to the question of whether a sort is a stable sort (preserving the original relative order of equal-by-comparison elements), but imposes a less strict requirement on the sort; a concurrent sort might not be stable, but might still produce the same (non-stable) ordering from the same provided sequence, every time. This is a particular instance of a common expectation noted above: while a parallel implementation of an algorithm will likely have internal nondeterminism (due to scheduling of work), it is often expected to behave deterministically, from the point of view of a caller.

This case study has two purposes: first, to show that the nondeterminism detection features of TSTL can be used to, with acceptable overhead, show that a parallel sort implementation provides a consistent, deterministic ordering, and second to show that the mechanism can also detect the nondeterminism in a parallel sort that does exhibit nondeterministic behavior. For the second purpose, we implemented a very simple swap-based parallel sorting algorithm that repeatedly has multiple threads scan through a sequence and swap out-of-order neighbors, until the sequence is sorted (a kind of parallel bubble-sort with a simpler termination criteria).

Figure 8 shows the complete TSTL harness for checking the parallel sorts. This harness produces sequences of integer pairs, where only the first integer is used in the comparison operator of the sorting algorithms, and checks that the results of `sort_parallel` calls are in fact sorted, using the `swapsort`'s function for checking if a sequence is sorted.

Running TSTL's random tester without checking for determinism, we can see that both sorting algorithms always produce sorted output. If we run with `--checkDeterminism --determinismDelay 0`, however, a test case showing

```
@import mergesort
@import swapsort

pool: <val> 10
pool: <data> 10
pool: <sorted> 10

<val> := <[-10..10]>

<data> := []
<data>.append((<val>,<val>))
len(<data,2>) < len(<data,1>) -> <data>.extend(<data>)

<sorted> := mergesort.merge_sort_parallel(<data>)
<sorted> := swapsort.swap_sort_parallel(<data>)
<sorted>

property: swapsort.is_sorted(<sorted>)
```

Figure 8: Complete TSTL harness for checking two parallel sorting algorithms for nondeterminism.

the `swapsort` behaving nondeterministically is almost instantly produced (it takes less than 3 seconds). If we comment out the call to `swap_sort_parallel` and only check the parallel merge sort, however, TSTL finds no nondeterministic behavior. Running tests of the merge sort with nondeterminism detection turned on results in a slowdown of approximately 40%.

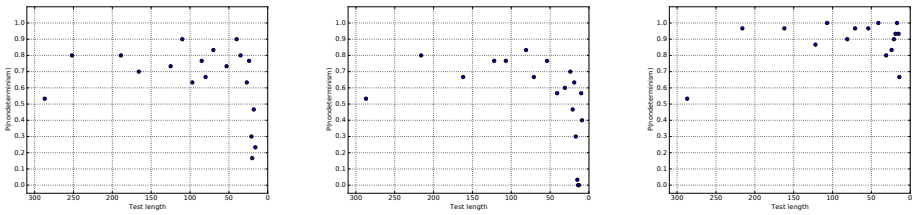
Reducing a lengthy test showing nondeterminism of the swapsort to a short test case using delta-debugging usually takes less than two minutes. If the number of attempts to produce nondeterminism is increased in the reducer using the `--determinismTries 10` option, the resulting test will be usually be very compact, e.g.:

```
data0 = []
val0 = 3
val1 = -9
data0.append((val0,val1))
val2 = -1
data0.append((val1,val2))
data0.append((val1,val1))
sorted0 = swapsort.swap_sort_parallel(data0)
```

For this simple example, more sophisticated delta-debugging strategies are not needed; we only want to see the code behaving nondeterministically, and do not care about preserving a higher probability of nondeterminism.

6.2. Case Study: `redis-py`

The `redis-py` [39] module implements a widely-used Python interface to the popular Redis in-memory database, cache, and message broker [50]. The



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

Figure 9: Delta-debugging of the same `redis-py` nondeterministic test

TSTL `redis-py` harnesses can be found in the `examples/redis` directory in the TSTL GitHub repo [26].

Using TSTL's harness for `redis-py`, both `redis-py` and Redis itself can be tested; unfortunately, generating stand-alone 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 library call (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, just 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 9a-9c show delta-debugging of a typical `redis-py` nondeterministic test, originally with 287 steps. The initial test behave nondeterministically about half the time. Figure 9a 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 3.1, 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 9c) 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.

Figure 10 shows all calls removed or modified after identification of sources of nondeterminism. After making these changes, 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

REMOVED:

```
<key> := <r>.randomkey()
<r>.expire(<key>,<int>)
<r>.pexpire(<key>,<int>)
{redis.exceptions.ResponseError} <r>.spop(<key>)
{redis.exceptions.ResponseError} <r>.srandmember(<key>)
{redis.exceptions.ResponseError} <r>.srandmember(<key>,<int>)
<pipe>.expire(<key>,<int>)
<pipe>.pexpire(<key>,<int>)
{redis.exceptions.ResponseError} <pipe>.spop(<key>)
{redis.exceptions.ResponseError} <pipe>.srandmember(<key>)
{redis.exceptions.ResponseError} <pipe>.srandmember(<key>,<int>)
```

MODIFIED:

```
{redis.exceptions.ResponseError} <r>.restore(<key>,<int>,<dump>)
 ⇒
{redis.exceptions.ResponseError} <r>.restore(<key>,0,<dump>)

{redis.exceptions.ResponseError} <pipe>.restore(<key>,<int>,<dump>)
 ⇒
{redis.exceptions.ResponseError} <pipe>.restore(<key>,0,<dump>)
```

Figure 10: `redis-py` calls removed or changed to avoid nondeterminism.

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.

Finally, AFL stability was 94.32% with the harness allowing nondeterminism, but rose to 98.32% using the harness with nondeterministic operations removed.

6.2.1. Visible Value vs. Final State Nondeterminism

We also used `redis-py` to investigate the tradeoff between overhead and ability to detect nondeterminism when using the two proposed approaches to horizontal determinism. We produced 300 tests of length 100, using the known-nondeterministic version of the `redis-py` harness. We then used `nondeterministic` and `stepNondeterministic` calls, both with a delay of 0.005 seconds and a single additonal execution of the test, to check these tests for nondeterministic behavior. Final state nondeterminism actually

detected one instance of nondeterminism that visible value nondeterminism failed to detect (21 detections vs. 20); in general, final state nondeterminism is less able to detect nondeterminism, but in this instance the difference is less important than the nondeterminism of whether a particular test will exhibit nondeterministic behavior in a single run. This difference is obviously not statistically significant; for the nondeterminism in `redis-py`, with these parameters, the two approaches cannot be statistically distinguished as to effectiveness in detecting nondeterminism.

Final state nondeterminism was also faster; visible value nondeterminism took, on average, 0.48% longer to check for nondeterminism on the 300 tests, 1.199 mean seconds vs. 1.205 mean seconds. The difference was significant, with p-value $< 1.19 \times 10^{-23}$ by a paired Wilcoxon test.

These results, of course, depend on the parameters of the check for nondeterminism. If we increase the delay to 0.01 seconds and the number of replays of a test used to check for nondeterminism, we almost double the number of nondeterministic tests discovered. Both visible value and final state approaches find 40 nondeterministic tests, though the overlap of tests thus detected is not quite perfect — each method detects two tests the other does not. The difference in overhead, interestingly, is very similar: 0.48%; however, under these parameters, visible value nondeterminism is actually cheaper on average than final state nondeterminism, due to early termination of the additional replays, when nondeterminism is detected (the difference is again significant by Wilcoxon test, with $p < 1.44 \times 10^{-10}$.

In practice, visible value nondeterminism and final state nondeterminism will often be very similar in both ability to detect nondeterministic behavior and overhead. However, under unusual conditions, the behavior of the approaches can be very different. The additional overhead of visible value nondeterminism is usually low because re-executing a test, not comparing state values for equality, is the primary cost in nondeterminism detection; however, if there are a very large number of state components, or some state components have an expensive equality check (e.g., recursive structures with cycle detection or depth limits), the overhead can grow, proportional to the length of tests under consideration. Similarly, final state nondeterminism in a setting such as `redis-py`, where divergences in behavior tend to propagate to other state components, or at least persist until termination of a test, detects nondeterminism quite effectively. However, in a setting where changes in behavior can easily by overwritten by future test behavior, and do not causally influence future computation, final state nondeterminism may be

```
@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>)
<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>)))
```

Figure 11: Complete TSTL harness for finding the hash-order bug in the datarray inference algorithms.

almost useless.

6.3. Case Study: `datarray` *Inference Algorithms*

The `datarray` module [5] 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 [53]. 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 4.1.1).

Figure 11 shows TSTL code for generating inputs to the `datarray` algorithms (it can be found in the `examples/datarray_inference` directory

in the TSTL repository [26]). 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.

6.4. Vertical Determinism Case Study: `pyfakefs`

The `pyfakefs` [40] 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 [40].

The TSTL harness for `pyfakefs` has been used to detect (and correct) over 80 faults. The harness can be found in the TSTL GitHub repository in the `examples/pyfakefs` directory, and the bugs discovered can be viewed at `https://github.com/jmcgeheeiv/pyfakefs/issues?q=label%3ATSTL`. However, the testing largely relies on the existence of a reference file system implementation. One purpose of failure nondeterminism is to make it somewhat easier to perform effective property-based testing of complex APIs like this even without a complete reference implementation, or in cases where the implementations do not use the same error codes (as is common, e.g. in NASA flight software [22, 23]).

6.4.1. Manually Inserted Fault

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 testing time 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 bad file system state the operation introduces). For this hypothetical subtle bug, failure determinism checks not only make it possible to detect the fault without a reference implementation, it improves on detection speed and ease of debugging even compared to a full-blown reference implementation (the MacOS file system, operating on a RAM disk) and strong, hand-tuned, differential testing.

6.4.2. Mutation Analysis

We wanted to determine whether there were also simple faults that could be detected by failure nondeterminism, but *not* detected by differential testing, and quantify the additional specification strength provided by failure determinism checks. We therefore used `universalmutator` [21] to produce 2,350 mutants of the `pyfakefs` core file system code. We restricted the generation to only mutate code covered during a 60 second run of the test harness, with differential testing turned off. In this *non-differential* mode, the harness will only detect a fault when it causes an unexpected exception to be raised, or results in a timeout due to, e.g., an infinite loop.

We first analyzed the mutants using 60 seconds of non-differential testing *without failure nondeterminism checks*, then with the full differential harness (also without failure nondeterminism checks) for 120 seconds; the additional 60 seconds is to make sure we accounted for the observed 17% extra time to detect in the manually constructed example: we want to maximize the chance to detect a bug using the strong version of the harness. For both comparisons, we then tested all surviving mutants using the non-differential

harness for 60 seconds, this time with failure nondeterminism checking turned on. We used the same random seed for all runs, so that the only differences would be specification-based.

Non-differential testing, without failure determinism checks, killed 872 mutants, a 37.1% kill rate. Adding a failure determinism check allowed the testing to kill an additional 98 mutants, an 11% improvement. The differential harness, with an additional 60 seconds of testing time, killed 1148 mutants, improving the kill rate to 48.8%. Failure nondeterminism checking added an additional 71 mutants to the total killed, a 6% improvement even for this strong differential harness with a larger test budget. Using failure nondeterminism was the only way to push the mutation score above 50%. The kill rates are generally low because `universalmutator`, by design, includes many operations that can produce equivalent mutants, but can also produce hard-to-kill non-equivalent mutants not produced by other mutation tools, e.g. code that throws away exceptions raised by a function call (see below), or code reversing a list.

In order to further investigate the additional specification power provided by failure nondeterminism detection, we inspected the mutants killed using the failure determinism check but not killed by the strong differential testing. The largest category of mutants not killed (22 of the 71 mutants) was what we refer to as "exception swallowing" mutants, which transform a Python statement into the same statement, but wrapped in a `try` block with a catch that ignores any raised exceptions, e.g., `foo()` becomes:

```
try: foo()
except: pass
```

It is easy to see that such mutants may introduce faults in the handling of errors, and thus would tend to cause failure nondeterminism. However, these are a minority of the mutations introducing hard-to-detect but non-equivalent failure nondeterminism. Other mutation operators resulting in subtle flaws not (at least easily) detectable by differential testing compared to a correct file system include: arithmetic operation changes, statement deletions, logical operator modifications, constant replacements (including replacement of a string with the empty string), and introducing a break into a loop. The variety of mutant types suggests that no more specifically tailored strategy such as checking for exception propagation, will work as well as introducing a notion of failure nondeterminism.

We also checked the cost of introducing failure determinism checking by analyzing mutants that could be killed by both methods. Even though in some cases the failure determinism check allows a mutant to be detected sooner, the mean time to kill mutants was 0.006 seconds larger with failure determinism checking, and this change was significant by Wilcoxon test ($p < 1 \times 10^{-15}$). While significant, this cost is probably too small to be of any real practical importance.

7. Threats to Validity

There are several primary threats to validity: first, our empirical results are limited to a small set of Python programs, ranging from relatively small and simple to large and complex libraries; the representative nature of these subjects is not clear. Furthermore, because no other tools implement the kind of approach taken here, based on automated generation of value pool based unit tests for a system (imperative property-based testing), to our knowledge, we were unable to perform a meaningful comparison with another nondeterminism detection tool. Available tools, such as DeFlaker [4] (`http://www.deflaker.org/`), perform a completely different task (DeFlaker is essentially a plugin for Maven regression tests, and cannot check nondeterminism at any smaller granularity than that of a whole tests's pass/-fail result) and target the Java language. iDFlakies [31], similarly, "does not further classify the causes of the flaky tests" beyond identifying them as likely related to test order, or not.

However, the primary aim of our results is to show 1) that nondeterminism at the library level can exist in real Python programs, 2) that it can be detected by an implementation of the formalism proposed in this paper, and 3) that the overhead for such detection, and cost to delta-debug nondeterministic tests, is not prohibitive. We believe that the case studies show that users testing Python libraries, and concerned about nondeterminism, would find the TSTL tools useful, and would be able to detect and debug nondeterministic behavior using our approach, a functionality that we believe to be unique in the automated test generation literature (and supported in no other tool). Comparison with a competing tool is not required to validate the basic idea, since precise empirical measurements of overheads, or improvements on an existing approach, are not proposed for evaluation.

Another threat to validity is posed by implementation errors. TSTL and the mutation tool are tested for bugs by a set of Travis CI tests, and

are (for academic research code) moderately well-used by external users, making this a lesser concern. The nondeterminism detected by TSTL can be demonstrated with standalone tests TSTL generates that do not depend on any TSTL code. The source code for the tools and all case studies is available for inspection and replication in the TSTL repository.

8. Related Work

At a very broad level, the topic of uncertainty (and thus nondeterminism) in software engineering has been addressed by Garlan [14], Elbaum and Rosenblum [11, 35], and Ziv and Richardson [58]. There is a general agreement that as systems become more complex, more distributed, and more *statistical*, these problems will only grow [35].

Gao et al. generally considered the question of how to make tests repeatable [13], 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 [54] 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 [10].

The problem of test nondeterminism is closely related to the (extremely important in industrial settings) issue of flaky tests [43, 36, 48, 33]. How to handle flaky tests in practice (when they cannot be avoided) is a major issue in Google-scale continuous testing [42], 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 [32], or provided large-scale empirical examination of tests from one large open source project (Apache) [36, 48]. Palomba and

Zaidman [48] investigated the relationship between test code smells and flaky tests.

Bell et al. proposed DeFlaker [4], which makes flaky tests much easier to detect by relying on the observation that if a test fails, and does not cover any changed code then (1) presumably it was a passing test in the past, or the tests would not be "green" before the latest code change so (2) the test is likely flaky. Comparing our approach with that of DeFlaker is difficult; DeFlaker applies only in the context of code *changes* and is essentially a heuristic that identifies certain test failures as due to nondeterminism. Our approach is primarily focused on automatically providing a more extensive *specification* in automated test generation, where a test "failing" is itself a sign of nondeterministic behavior, that could give rise to flaky behavior.

iDFlakies [31] is a framework and dataset for flaky tests, but again focuses on whole-tests, and detecting flakiness by actually observing it. Their work shows the order-dependence accounts for a little more than 50% of flaky test causes, and they (like us) classify the rest as due to "concurrency, timeouts, network/IO, etc.' — the kinds of cause that our approach focuses on identifying.

The present paper, rather than focusing on flaky tests as such, therefore, investigates methods for handling nondeterminism in property-based test generation [7, 49]. Again, 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. Our approach is, in a sense (especially the delta-debugging modifications) in line with Harman and O'Hearn's proposal to simply accept that "All Tests Are Flaky" [28], and work with probabilistically failing tests.

Other efforts [6] 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.

Vertical/failure determinism is a less-studied concept, and to our knowledge the formulation here, as a kind of determinism in a different "direction," has not previously appeared; the kinds of errors that are exposed, however, are not new. In particular, we believe that many faults discovered using failure determinism are related to the propagation of error conditions in code, which has been studied in, e.g., file systems, using static analysis [52].

TSTL's approach to test generation, and our instantiation of the formal

definitions of nondeterminism are based on a formulation of unit tests using *pools* of values [3], which provides a practical solution to the problem of defining the visible state to be compared when checking for nondeterminism.

Finally, Cotroneo et al. [9, 8], and Grottke and Trivedi [27] have followed on early work on understanding bugs and how they manifest, including transient [30] software faults. This work informs our attempt to identify sources of nondeterminism, and should provide other, context and project-specific, sources that could be introduced into our general framework.

9. 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 [15] and Mandelbugs [27, 8] are widely loathed), and leading to the costly problem of flaky tests [43, 33].

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 Mandelbugs 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 [56] to avoid high startup costs for test executions.

More generally, we are interested in using test decomposition [18] 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.

[1] Ikhlaaq Ahmed, Alexander J Sutton, and Richard 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] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex 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] Jamie Andrews, Yihao Ross Zhang, and Alex Groce. Comparing automated unit testing strategies. Technical Report 736, Department of Computer Science, University of Western Ontario, December 2010.

[4] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. DeFlaker: automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering*, pages 433–444. ACM, 2018.

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

[6] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc 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.

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

[8] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi. Fault triggers in open-source software: An experience report. In *International Symposium on Software Reliability Engineering*, pages 178–187, 2013.

[9] Domenico Cotroneo, Roberto Pietrantuono, Stefano Russo, and Kishor Trivedi. How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation. *Journal of Systems and Software*, 113(C):27–43, March 2016.

[10] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *PACMPL*, 1(OOPSLA):93:1–93:29, 2017.

[11] Sebastian Elbaum and David S. Rosenblum. Known unknowns: Testing in the presence of uncertainty. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 833–836, 2014.

[12] Donald 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.

[13] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen 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.

[14] David Garlan. Software engineering in an uncertain world. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 125–128, 2010.

[15] Jim 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.

[16] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability*. accepted for publication.

[17] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *International Conference on Software Testing, Verification and Validation*, pages 243–252, 2014.

[18] Alex Groce, Paul Flikkema, and Josie 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.

[19] Alex Groce, Klaus Havelund, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.

[20] Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *International Symposium on Software Testing and Analysis*, 2017. accepted for publication.

[21] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *International Conference on Software Engineering: Companion Proceeedings*, pages 25–28, 2018.

[22] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[23] Alex Groce, Gerard Holzmann, Rajeev Joshi, and Ru-Gang 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.

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

[25] Alex Groce, Jervis Pinto, Pooria Azimi, and Pranjal Mittal. TSTL: a language and tool for testing (demo). In *ACM International Symposium on Software Testing and Analysis*, pages 414–417, 2015.

[26] Alex Groce, Jervis Pinto, Pooria Azimi, Pranjal Mittal, Josie Holmes, and Kevin Kellar. TSTL: the template scripting testing language. `https://github.com/agroce/tstl`.

[27] M. Grottke and K. S. Trivedi. Fighting bugs: remove, retry, replicate, and rejuvenate. *IEEE Computer*, 40(2):107–109, 2007.

[28] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2018.

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

[30] P. Jalote, Y. Huang, and C. Kintala. A framework for understanding and handling transient software failures. In *International Symposium on Software Testing and Analysis*, 1995.

[31] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: a framework for detecting and partially classifying flaky tests. In *IEEE International Conference on Software Testing, Verification and Validation*, 2019.

[32] Wing Lam, Sai Zhang, and Michael 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, March 2015.

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

[34] Roman 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`.

[35] Jian Lu, David S Rosenblum, Tevfik Bultan, Valerie Issarny, Schahram Dustdar, Margaret-Anne Storey, and Dongmei Zhang. Roundtable: the

future of software engineering for internet computing. *IEEE Software*, 32(1):91–97, 2015.

[36] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

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

[38] Hector Marco-Gisbert and Ismael Ripoll. On the effectiveness of full-aslr on 64-bit linux, 2014.

[39] Andy McCurdy. redis-py: Python Redis Client. `https://github.com/andymccurdy/redis-py`.

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

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

[42] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale continuous testing. In *International Conference on Software Engineering*, pages 233–242. IEEE, 2017.

[43] John 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.

[44] Shaun 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/`.

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

[46] 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`.

[47] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

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

[49] Manolis Papadakis and Konstantinos Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 39–50, New York, NY, USA, 2011. ACM.

[50] redislabs. Redis. `https://redis.io/`.

[51] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Programming Language Design and Implementation*, pages 335–346, 2012.

[52] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 270–280, 2009.

[53] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[54] 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.

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation*, pages 283–294, 2011.

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

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

[58] Hadar Ziv and Debra Richardson. The uncertainty principle in software engineering. In *International Conference on Software Engineering*, 1997.