Practical Automatic Lightweight Nondeterminism and Flaky Test Detection and Debugging for Python Libraries

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 unexpected and unwanted behavioral nondeterminism. Such nondeterministic behavior can result in software bugs and 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 extend an existing testing system to allow detection and debugging of nondeterminism. We show how to make delta-debugging effective for probabilistic faults in general, and that our methods can improve mutation score by 6% for a strong, full differential test harness for a widely used mock file system.

ACM Reference Format:

. 2020. Practical Automatic Lightweight Nondeterminism and Flaky Test Detection and Debugging for Python Libraries. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [11]. 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" [14], though the more accurate usage is "Mandelbug" [8, 26], 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 go awry 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. [12] observed coverage differences of up to 184 lines of code for the same test, and false positive rates as high as 96%. Nondeterminism is often problematic for developers, who want to assume that library code behaves in a predictable fashion; nondeterminism is frequently vexing in debugging efforts; most importantly, perhaps, nondeterminism is *often disastrous* for large-scale highly automated testing.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 impractical. We write complex software systems because we cannot predict their behavior (if we could perform the calculations in advance, we would just do so). As a consequence, rather than defining determinism in terms of prediction, we usually 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. Given the complete state of the system (which includes the entire state of the underlying hardware, the operating system, storage devices, etc.), most software *is* completely predictable. What we actually mean is that, *given a certain limited abstraction of state and inputs, observable behavior is repeatable.*

Because the programmer's abstraction of the system ignores such a large number of details, very few tests are "deterministic" in the sense that they eliminate all changes 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 is seldom of interest. What matters is when some kind of nondeterminism unexpectedly impacts computed values at a higher level of abstraction, 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.

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, automated test generation 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 asserting equality with all values produced during a reference

run, since the final step of a test is arbitrary, generated in an effort to increase code coverage, not establish functional correctness. This may be one reason that automated testing is seldom used to produce general regression tests. While bugs found by tools are added to regression suites, the full set of passing tests is seldom preserved, perhaps because such tests are likely to be *flaky*.

1.1.1 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 set of tests that can be run against softwarre every time it is modified, to ensure that the modification has not broken the system. *Flaky tests* [41] are regression tests that fail in an intermittent, unreliable fashion. 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 testing environment. 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 are often ignored, and therefore serious software faults may be missed. Finally, to mitigate these problems, flaky tests are often run multiple times, wasting computing resources and delaying acceptance of code changes. As an analogy, we note that 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 itself becomes a serious burden.

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, is on detecting sources of flaky-ness in libraries, before they propagate.

1.2 Contributions

This paper proposes (1) a number of formal definitions of types of nondeterminism (*horizontal* and *vertical*) 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. We also introduce necessary modifications to the widely used delta-debugging algorithm in order to better handle nondeterminism as a test property.

Our methods enable automated production of nondeterminism-free regression tests even for libraries with some sources of nondeterminism, documentation of sources of nondeterminism that might produce problems for manually-constructed tests, and, most importantly, detection and debugging of unexpected nondeterminism that is, itself, a bug. We use a set of case studies. including real-world, widely-used, Python libraries, to demonstrate the utility of our ideas. Our novel notion of 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.

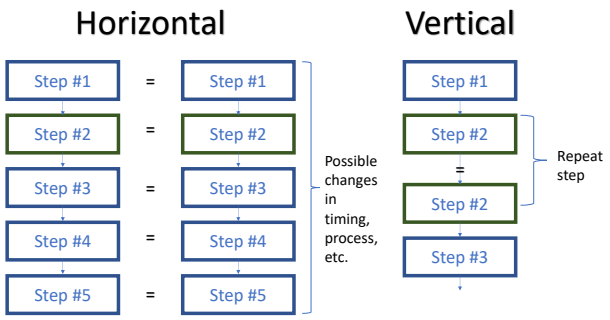


Figure 1: Types of determinism

2 PRACTICAL LIGHTWEIGHT NONDETERMINISM DETECTION

We define two basic types of determinism, shown in Figure 1: horizontal determinism and vertical determinism. In horizontal determinism 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 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. Vertical determinism is both rarer and more specialized than horizontal determinism. However, vertical determinism 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* [21, 39]. 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. Differential testing is extremely powerful, in that any (properly defined) divergence of behavior indicates a functional correctness fault in (at least) one of the systems under test, and is widely used for systems software such as compilers [39, 53] and file systems [18, 22]. The major limitation of differential testing 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 problem, then, becomes one of deciding at what granularity the reference equivalence will be checked: e.g., processor-instruction and memory-layout determinism is seldom necessary or even desired. 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. 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. 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 designation of some values as opaque.

2.1.3 Final State Determinism. Visible value determinism has significant limitations. While it provides the finest granularity, which is important for debugging purposes, it is also expensive, requiring checks on 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 a row should always return the same value

One approach, then, would be to identify idempotent operations in an automated test generation harness, and 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 an additional 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 should be "rolled back."** In other words, failure determinism is a property stating that an API is *transactional* with respect to failing calls. From a user's perspective, some behaviors of the Mac OS High Sierra root exploit (CVE-2017-13872 [43]) exhibited failure nondeterminism. Attempting to login with the root account with an empty password appeared to fail, then, on a repeated try, succeeded [32, 42]. 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` and `write` tend to explicitly return a degree of success, rather than signalling total failure, when appropriate.

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 easily be automatically checked. Moreover, the expected overhead should be even lower than for other vertical determinism, in that we can expect most failing operations to be fast (since 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 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 failure deterministic; this more restrictive notion still captures such issues as the Apple login bug.

2.3 Formal Definitions

We can formally distinguish the types of nondeterminism by using a variant of a labeled transition system (LTS)l, where (1) S is a set of states; (2) V is a set of *observable* states, where $|V| \leq |S|$; (3) $v : S \to V$ is a total function that, given a state, maps it into the set of observable states, such that every state has an observable component, which may be the complete state, or only an aspect of the full state; (4) $I \subseteq S$ is a set of initial states; (5) A is a set of *actions*; and (6) $T \subseteq S \times A \times S$ is a transition relation.

We assume that the *underlying* behavior of a system may be deterministic, or nondeterministic 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 $\forall i < n.(s_i, \alpha_i, s_{i+1}) \in T$. The concept of an action does not restrict this approach to reactive systems; it can be generalized to consider the only action that varies to be the selection of an input, α_0, with the remainder of the actions being internal τ actions.

2.3.1 Horizontal Determinism. 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.

2.3.2 Vertical Determinism. 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, in theory, 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. In an implementation, we would usually restrict the search to a finite number of repetitions; for many faults, two "copies" will suffice.

2.3.3 Failure Determinism. To define failure nondeterminism, we extend the transition relation to include a notion of *failure*: $T \subseteq S \times A \times S \times \mathtt{F} : \mathtt{bool}$, where the boolean F indicates whether the action A *failed*. A trace $t = s_0 \xrightarrow{\alpha_0} (s_1, F_0) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} (s_n, F_{n-1})$ shows *failure nondeterminism* if $\exists i$ such that F_i (that is, α_i *fails*) and either (1) $v(s_i) \neq v(s_{i+1})$ (the visible state changes) or (2) $\alpha_i = \alpha_{i+1} \wedge \neg F_{i+1}$ (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, the notion of observable components of a state, or any comparison of states. It only requires that we know whether an operation failed.

3 SOURCES OF NONDETERMINISM

In order to implement a practical nondeterminism-detection tool, it is important to consider the real-world sources of nondeterminism. Most nondeterministic behavior in tests is probably attributable to simple effects that can vary with a repeated execution, in the same (approximate) enviroment, of the same test. There are two broad classes of nondeterminism that can usually be detected by executing the same test twice in succession, without attempting to control any other factors. **External Environment** 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. Similarly, calls to `time` functions will usually return different values on different runs, and even if only elapsed time is used, due to system load, I/O, and other external factors. **Concurrency** is a second common cause for nondeterminism, when an algorithm is implemented using concurrency, in the (incorrect) belief that the result of the algorithm remains deterministic. E.g., a multi-threaded implementation of a search algorithm, that returns the location of an item in an unsorted list; in the presence of duplicate items, thread scheduling may change the index returned.

Lam et al. [29] 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 these two primary sources.

3.1 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. In terms of our formalism, an action needs to be introduced that indicates the generation of a fresh process. Address Space Layout Randomization (ASLR) [36] 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 and thus be detectable.

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, in order to mitigate hash-based denial of service attacks [44]. Until version 3.6 this not only resulted in changes in exact hash values, but in the order of iteration on dictionaries; in Python 3.6, the default dictionary implementation became an ordered dictionary with consistent iteration, though actual hashes remained nondeterministic. While very few programs rely on exact hash values, many relied on them in that they only functioned correctly with a predictable order for dictionary iteration. Testing Python code for nondeterminism based on 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.

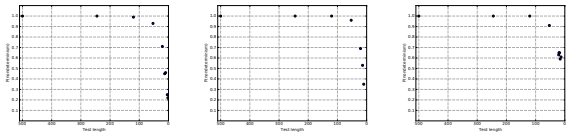
4 NONDETERMINISM AND DELTA-DEBUGGING

Delta-debugging [55] 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 with some property (usually "it fails") and produce a smaller test that has the same property. Delta-debugging as originally proposed uses a modified binary search, but we rely only on the general structure: [19, 49]: Given a test case, $T_{INIT} : test$ and a predicate $PRED : test \rightarrow bool$, we reduce T_{INIT} with respect to $PRED$ as follows:

(1) $T_{CURR} = T_{INIT}$
(2) Let T_{NEXT} = a variation of T_{CURR} that has not yet been tried. If none exist, stop and return T_{CURR}.
(3) Add T_{NEXT} to the set of variations that have been tried.
(4) If $PRED(T_{NEXT})$, set $T_{CURR} = T_{NEXT}$.
(5) Go to 2.

Delta-debugging in the context of 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 but very difficult with more than a hundred calls. In horizontal nondeterminism detection, however, delta-debugging also tends to *change the probability of nondeterministic behavior*; this can be both harmful and beneficial. This behavior is part of a more general (and, to our knowledge, not previously investigated) issue: reducing a test with respect to an arbitrary predicate *PRED* (including failure, but also code coverage, etc. [2, 15, 16]) that *only holds with a certain probability* (the predicate itself is nondeterministic). "The test behaves flakily" is an obvious relevant example of such a *PRED*.

For *monotonic PRED*, removing part of a test *cannot increase the probability of the predicate holding*: when we reduce t to r, $P(PRED(r)) \leq P(PRED(t))$. This is fairly common. In *non-monotonic* cases, however, there is no such upper bound. Removing a step in t may *increase* the probability of *PRED*. In probabilistic settings, we can often exploit non-monotonicity. For instance, a test may be "almost non-flaky" and we want $P(fail)$ to be closer to 50%, to aid debugging.



(a) No modification **(b) N=100** **(c) M=10, N=10**

Figure 2: Reducing the same simple test; x-axis is time/changing T_{CURR}; y-axis is true $P(PRED)$.

4.1 "Publication Bias" in Delta-Debugging

However, simply using delta-debugging off the shelf with a *PRED* such as $P(fail) > 0.3 \wedge P(fail) < 0.7$, to force a test to be highly 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 r that delta-debugging returns is always one such that $PRED(r)$ has evaluated to true at least once. 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]. We can think of each application of *PRED* to a candidate variant T_{NEW} as a scientific experiment. A predicate like $P(fail) > 0.3 \wedge P(fail) < 1.0$ cannot be evaluated by determining the true probability of failure; we have no more direct access to this value than a medical researcher has access to the true effect size and direction of a proposed medical treatment; rather, the test must be run some concrete number of times, and the number of failures counted. 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, test reduction algorithms explore a search space that often contains thousands or millions of tests. The predicate is evaluated on each of these, and so even with large N, it is likely that *some* evaluation will produce a poor estimate of $P(fail)$. At this point, reduction is "stuck" with the error, because the algorithms usually do not allow backtracking. After such a "mistake," finding further reductions will become harder, but may still be possible due to another unlucky evaluation. We define a *false positive* in probabilistic delta debugging as a case where *PRED* evaluates to true, but the actual probability constraints of *PRED* are not satsified (false negatives also exist, but are less of a problem).

The analogy to publication bias is simple. Assume that a paper in, e.g., a medical journal will only be published if it shows that a treatment is effective, with the statistical requirement that $p < 0.05$. Even if no researchers are dishonest, 5 of every 100 papers published in the journal will be report an effect with the wrong effect size or even direction! Now consider a "hot" topic, a kind of treatment many research teams may investigate. Major journals tend not to publish papers that say "this new treatment, which is not established, may not work," (especially if the result is not "we accept the null hypothesis" but only "the experiment did not show $p < 0.05$"), but they are very likely to publish a paper that says "this exciting new treatment works!" The *bias* in favor of positive results is shared by test reduction, which; is much more influenced by cases where *PRED* holds than cases where it does not. The "unpublished papers" are invisible, magnifying the effect of the "published" experiments.

4.2 Replication Mitigates "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 direction or effect size is a fluke. One way to produce this effect would be to allow reducers to backtrack if the probabilities observed in predicate evaluations suddenly exhibit a strong discontinuity. However, this requires modifying the implementations, which is difficult and sometimes not really feasible. Ideally, the solution should be implementable simply by modifying *PRED* itself.

A costly but plausible solution is to make N large in comparison to the number of expected predicate evaluations performed during delta-debugging. However, given the large number of evaluations performed, this will tend to make reduction extremely slow, since N must be very large indeed. We propose using a dynamic sampling approach, where N is small, but if the predicate evaluates to true, M repeated true evaluations ("replications") are required before the predicate is counted as holding. To evaluate the predicate *PRED* given N, M, and desired probability bound p, the algorithm is:

```
for i = 0 ... M − 1
   T = # times PRED is TRUE over N evaluations.
   if T/N < p, return FALSE.
return TRUE
```

A set of M repeated false positives with $N = \frac{K}{M}$ samples each is much less likely than a false positive with K 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 *PRED* 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; gathering experimental data during a trial delta-debugging run and tuning N and M to yield desired results is more effective. In fact, by tuning M, any degree of confidence can be achieved, with the basic tradeoff being between finding a test with the desired probabilities, and the speed and effectiveness of reduction. 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 in 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 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, many 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 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 or known flakiness. We have often observed delta-debugging taking a test that had such a high probability of failure that we were unaware it was "flaky" (only hundreds of repeated executions showed there was indeed a small chance the could pass) and transforming it into a test that only failed once in every 10 to 20 executions. *Delta-debugging can easily act as a flakiness multiplier.* Multiple evaluations using replications are our proposed solution.

5 IMPLEMENTATION

5.1 Pools and Visible State

While the formal definitions above offer a framework for checking nondeterminism, 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 processor instructions. However, this definition is highly impractical. First, it is extremely inefficient to compare multiple executions for full-memory-state equivalence at every instruction step. Second, and even more importantly, as noted in the introduction, very few programs are deterministic in this strict sense. 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 [45], showing that unit tests 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 also taken from a pool, e.g. `intVP[2] = fooVP[2].bar(intVP[0], intVP[1]);` (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.

We implemented our approach as a modification of the TSTL [23, 24, 28] system, an open-source language and tool for property-based testing [7, 35] of Python code.

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. and described above, 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 2.3, 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.

5.2 Nondeterminism Detection

Because TSTL supports differential testing [28], 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). In TSTL, *visible value determinism* is, as proposed above, based on comparing the values of *all pool variables* after each test action (pool values other than the one assigned to must

also be compared, since a common source of nondeterminism is when a call results in a change to a value passed as a parameter, or changes a shared reference held by two values). By default V as defined in Section 2.3 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.

We provide support for the real-world sources of nondeterminism described in Section 3, via control over timing and/or execution in a new process. Changing the timing of steps is also effective in exposing concurrency-related nondeterminism, since it often changes the interleaving of threads.

Because TSTL has an interface to AFL [54], we can use AFL's sophisticated heuristics to perform very thorough, week-long checks for nondeterminism. Moreover, since there is substantial overhead in nondeterminism detection, AFL can be used to predict whether tested Python code is a good candidate for checking for horizontal nondeterminism; if the AFL *stability* statistic (see the AFL documentation for details [54]) is lower than 100%, it may indicate horizontal nondeterminism. 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 for other problems.

First-class vertical nondeterminism checks are currently limited to failures. In terms of the formalism, if execution of action α_i raises an exception that does not, itself, indicate a test failure, then F_i is true, and TSTL repeats α_i in order to ensure that F_{i+1} is also true.

6 EXPERIMENTAL EVALUATION

In order to evaluate our approach and implementation, we applied nondeterminism detection to real Python code, including some widely used libraries (according to GitHub, `pyfakefs` is used by at least 338 other projects, and `redis-py` is used by at least 50,000). The primary points we wanted to explore were **RQ1:** whether our approach was able to reliably detect actual nondeterminism, **RQ2:** whether the overhead of nondeterminism detection for real systems was acceptable, and **RQ3:** whether the performance of delta-debugging in this setting was acceptable, in terms of both time and results (amount of reduction). We chose two of the subjects, `redis-py` and `pyfakefs`, based on the fact that they were previously-existing large TSTL harnesses, and serve to show how easily our approach can be integrated into an existing test generation effort. The other two subjects were chosen, and new harnesses written, specifically to demonstrate specific, important sources of nondeterminism in Python code: concurrency and hash salting. In addition to our core research questions (**RQ1-RQ3**) shared by all subjects, we report on aspects of using the TSTL nondeterminism tools specific to each subject.

Table 1 shows a summary of our experimental results. In all cases, our approach was able to detect interesting nondeterminism (in fact, all nondeterminism of which we are aware). The overhead on testing imposed by nondeterminism checks varied considerably, but in the worst case did not double the average cost of each action (the expected overhead, since we essentially execute each test twice), and thus never halved the amount of testing performed. Test reduction always reduced the size of generated tests by 85% or more, in less than 15 minutes, thus showing that reduction has a large payoff at an acceptable price.

	RQ1	RQ2	RQ3	
Subject	Detection	Overhead	Reduction %	Reduction time
Parallel Sort	Yes (see below)	~40%	~85%	~2 minutes
`redis-py`	Yes (see below)	~20%	~95%	~10 minutes
`datarray`	Yes (see below)	~93%	~90%	~92 seconds
`pyfakefs`	Yes (see below)	~8%	~99%	~1 second

Table 1: Overview of experimental results

6.1 Parallel Sorting

Our first subject is a simple implementation of a parallel merge sort[1]. The merge sort and harness can be found in the TSTL GitHub repository [25], 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 experimental subject serves 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).

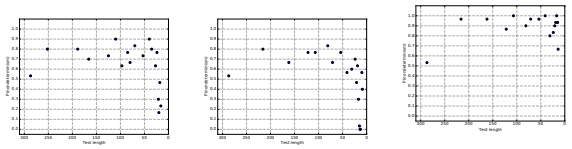
Figure 3 shows the complete TSTL harness for checking the parallel sorts. This harness produces sequences of integer pairs, where only the first integer is used in the comparison operator of the sorting algorithms, and checks that the results of `sort_parallel` calls are in fact sorted, using the `swapsort`'s 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 the `swapsort` behaving nondeterministically is almost instantly produced (it takes less than 3 seconds), thus answering yes to **RQ1**. If we comment out the call

[1] https://gist.github.com/stephenmcd/39ded69946155930c347

```
@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 3: Complete TSTL harness for checking two parallel sorting algorithms for nondeterminism.



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

Figure 4: Delta-debugging of the same `redis-py` nondeterministic test

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% (**RQ2**.

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, in this case reducing the original test from 56 to 8 steps.

```
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 The `redis-py` Library

The `redis-py` [37] module implements a (very) widely-used Python interface to the popular Redis in-memory database, cache, and message broker [48]. The TSTL `redis-py` harnesses can be found in the `examples/redis` directory in the TSTL GitHub repo [25].

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 `redis-py` has a stability of only 56.26%, a clear indicator of significant 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`) 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 nondeterministic tests exposing the root sources of nondeterminism/flakiness: we set the harness to repeatedly run, then produce a regression test. If the test is flaky (exhibits nondeterminism in result, over a set of 3 runs), we use the TSTL reducer to shrink it to a normaliized [19] test. At the end of the testing period, all distinct tests exhibiting nondeterminism are reported to the user. The process reduces the large number of different tests exhibiting flakiness to a small set of tests, each short, showing a different cause of nondeterminism. The set produced is not perfect, there is some duplication of causes, but it is sufficient for identifying causes of nondeterminism.

This process is unlikely to truly require 12 hours, of course; our point is that it is completely automated and can be given time proportional to the need to find even low-probability sources of nondeterminism. The actual amount of time given may be smaller, or, for that matter, larger (leaving an important library to self-document nondeterminism over a weekend is hardly a bad practice). In this case, the probability of flakiness is not important, just the possibility, so we let the reducer create a minimal test with some non-zero (but possibly very small) observed probability of nondeterminism. Our 12 hour run identified all sources of nondeterminism successfully, according to a series of 48 hour runs performed on the version of

the harness we modified to remove nondeterminism, which produced no flaky tests. We determined that the mean time to find all the sources of nondeterminism and produce minimal tests 3.2 hours, which is not trivial, but also, for this purpose, not excessive. Our answer to **RQ1** is thus a strong affirmative.

We removed 11 calls from the `redis` harness, and modified 2 calls, 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 seems acceptable. AFL stability was 56.26% with the harness allowing nondeterminism, but rose to 98.32% using the harness with nondeterministic operations removed.

Removing sources of expected nondeterminism also 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, answering **RQ2** also in the affirmative. 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 with a complex library like `redis-py`; running the test again and checking equality is relatively inexpensive.

As to **RQ3**, Figures 4a-4c show delta-debugging of a typical `redis-py` nondeterministic test, originally with 287 steps. The initial test behaves nondeterministically about half the time. Figure 4a 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 4.1, 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 4c) 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.

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 (but not root cause) 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, we expect that 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 almost useless.

6.3 Berkeley `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

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

[51]. 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 3.1).

Figure 5 shows TSTL code for generating inputs to the `datarray` algorithms (it can be found in the `examples/datarray_inference` directory in the TSTL repository [25]). 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), answering **RQ1** again in the affirmative. 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 (**RQ3**). Interestingly, the way that `python-afl` is used in TSTL means that AFL stability is 100% even for the original, broken, code, 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(**RQ2**. This is due to the high cost of subprocess creation and communication. However, even with this essentially worst case behavior (where test runtime is very low compared to process overheads), we sacrifice less than half of the test executions for a given time budget.

6.4 Vertical Determinism: `pyfakefs`

The `pyfakefs` [38] 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 [38].

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 [21, 22]).

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 essentially immediately, even without using a reference, when we compiled the harness with the `--checkFailureDeterminism` flag (an affirmative answer to **RQ1**). Moreover, the overhead for the check for failure determinism in a version of the code without the `remove` error was less than 8% (**RQ2**). 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). 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 (**RQ3**). For this hypothetical subtle bug, failure determinism checks not only make it possible to detect the fault without a reference implementation, they 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, to further answer **RQ1** in the special case of failure determinism. We therefore used `universalmutator` [20] 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 could have applied mutation testing to `redis-py`, but we are dubious of the value of such an analysis without a strong differential harness to compare to; the `redis-py` testing is limited in strength, and largely useful for producing high-coverage operation sequences from which to construct manual unit tests.

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. This is a strong affirmative answer to **RQ1** at least for a large set of hypothetical bugs.

We also checked the cost of introducing failure determinism checking by analyzing mutants that could be killed by both methods (**RQ2**). 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 *statistically* significant, this cost is almost certainly too small to be of any real practical importance.

6.5 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 [29], 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. 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 the nondeterminism detected by TSTL can be demonstrated with standalone tests that do not depend on our code. The source code for our TSTL version and all experimental subjects will be made available for inspection and replication.

7 RELATED WORK

At a very broad level, the topic of uncertainty (and thus nondeterminism) in software engineering has been addressed by Garlan [13], Elbaum and Rosenblum [10, 33], and Ziv and Richardson [56]. There is a general agreement that as systems become more complex, more distributed, and more *statistical*, these problems will only grow [33].

Gao et al. generally considered the question of how to make tests repeatable [12], 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 their "test harness factors" include delays, which can be of importance to nondeterminism at the library level. Shi et. al [52] examined what might be seen as a related, though in a

sense, 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. Determinism is also sometimes used as a property in domain-specific testing, e.g. for shader compilers [9].

The problem of test nondeterminism is closely related to the (extremely important in industrial settings) issue of flaky tests [31, 34, 41, 46]. How to handle flaky tests in practice (when they cannot be avoided) is a major issue in Google-scale continuous testing [40], 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 [30], or provided large-scale empirical examination of tests from one large open source project (Apache) [34, 46]. Palomba and Zaidman [46] 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 change so (2) the test is likely flaky. Comparing our approach with DeFlaker is difficult; DeFlaker applies only in the context of code *changes* and is essentially a heuristic that identifies 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. iDFlakies [29] 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, investigates methods for handling nondeterminism in property-based test generation [7, 47]. 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. 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" [27], and work with probabilistically failing tests. 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. We additionally introduce the first variation of delta-debugging that properly handles probabilistic reduction criteria.

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 true determinism.

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 such faults are related to the propagation of error conditions, which has been studied using static analysis [50].

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.

8 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 [14] and Mandelbugs [8, 26] are widely loathed), and leading to the costly problem of flaky tests [31, 41]. 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 utility of the approach on 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 opaque. This would make testing libraries mixing deterministic behavior and nondeterministic behavior, e.g. cryptographic librarie, much easier. Similarly, in order to extend the utility of vertical nondeterminism, we would like to automatically identify usually-idempotent operations. Finally, we are interested in using test decomposition [17] to more easily isolate, understand, and avoid nondeterminism in tests, and to mitigate flaky tests. Reliable detection of nondeterminism is a first step towards evaluating such fine-grained proactive flaky test mitigations.

REFERENCES

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

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

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

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

[40] 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.

[41] 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.

[42] 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/.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] 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.

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

[52] 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.

[53] 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.

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

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

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