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

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

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

***Abstract*—A critically important, but surprisingly neglected, 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.**

I. Introduction

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

Determinism is particularly important for testing and debugging, where being able to exactly reproduce system behavior is essential to productivity. Developers using a test exhibiting nondeterminism to debug a system face a serious challenge. Regression testing effectiveness can be significantly reduced if tests cover code or, worse yet, fail, only intermittently and unpredictably. Such behavior is unfortunately all too common: Gao et al. [7] observed coverage differences of up to 184 LOC for the same test, and false positive rates as high as 96%. Nondeterminism is problematic for developers, who want to assume that library code behaves in a predictable fashion; nondeterminism is vexing in debugging; nondeterminism is *often disastrous* for large-scale automated testing.

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. In the real world, prediction may be possible but impractical, or some details neither predictable nor interesting. We write complex software systems because we cannot predict their behavior, and we usually want to abstract away from machine-level details. As a consequence, rather than defining determinism in terms of prediction, we usually say that a system is deterministic if, *given a certain limited higher-level abstraction of state and inputs, observable behavior is repeatable*.

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 do this.

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 every time code is modified, to ensure that the modification has not broken the system. *Flaky tests* [23] 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, 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 *correct* code-under-test. Second, failures in flaky tests are often ignored, and therefore serious software faults missed. Finally, to mitigate the problem, flaky tests are often run multiple times, wasting computing resources and delaying code changes. Flaky tests are, for us, simply a special case of the *horizontal* nondeterminism described below. Our focus is on detecting sources of flaky-ness, *before* they propagate.

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. Evaluating these approaches with realistic Python libraries, we show large improvements to both fault detection and test reduction.

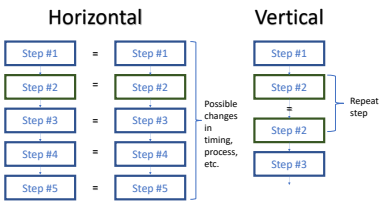


Fig. 1: Types of determinism

II. Lightweight Nondeterminism Detection

We define two basic types of determinism, shown in Figure 1: *horizontal* 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.

A. Horizontal Determinism

1) Determinism and Reflexive Differential Testing: Horizontal determinism can be best understood by thinking of nondeterminism detection as an unusual kind of *differential testing* [21]. 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 [21], [31] and file systems [9]. 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.

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.

While visisble value determinism provides very fine granularity, which is important for debugging purposes, it is also expensive, requiring checks on a potentially very large number of values. It is often sufficient to only compare final states of a computation performed by the system, which we refer to as *final state determinism.*

B. Vertical Determinism

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

However, for the specialized case of failure determinism, no specification is required. Failure determinism is the following restriction on an API: **If a call/action *fails*, and indicates this to the caller, it should not modify system state in any way; changes should be "rolled back."** Some behaviors of the Mac OS High Sierra root exploit (CVE-2017-13872 [24]) exhibited failure nondeterminism. Attempting to login with the root account with an empty password appeared to fail, then, on a repeated try, succeeded. 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 be automatically checked. The overhead should be even lower than for other vertical determinism, in that most failing operations are fast. Checking equivalence of full observable state, however, is still expensive, and requires defining the observable components of a state, suggesting the shortcut of simply *repeating the failing operation, and checking if it still fails* which still catches such issues as the Apple login bug.

C. Formal Definitions

We can formally distinguish the types of nondeterminism by using a variant of a labeled transition system (LTS), 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.

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) 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.

3) Failure Determinism: To define failure nondeterminism, we extend the transition relation to include a notion of *failure*: $T \subseteq S \times A \times S \times \texttt{F} : \texttt{bool}$, 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.

D. 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. Calls to `time` or `random` are obvious examples. **Concurrency** is a second common cause. 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.

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), which scrambles the layout of memory of the process in which an executable runs in order to make it harder to exploit memory-safety vulnerabilities, is probably the most important source of nondeterminism that arises (only) from change in process.

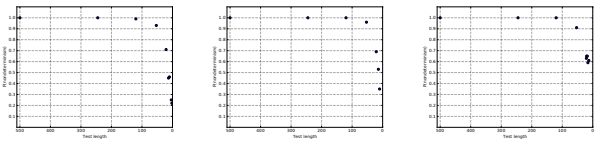
Another example of process-based nondeterminism arose when Python version 3.3 introduced automatic random salting of hashes on a per-process basis, in order to mitigate hash-based denial of service attacks [25]. Until version 3.6 this not only resulted in changes in exact hash values, but in the order of iteration on dictionaries. Many Python programs relied on a predictable ordering.

III. Probabilistic Test Reduction

Delta-debugging [33] 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 [10]: Given a test case, $T_{INIT} : test$ and a predicate $PRED : test \to bool$, 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



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

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

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. [8]) 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. E.g., a test may be "almost non-flaky" and we want $P(fail)$ to be closer to 50%, to aid debugging.

A. "Publication Bias" in Reduction

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. In fact, 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 a small chance it 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.* 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.

B. 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, and a desired accuracy for P can be obtained with a much smaller value $M \times N$ than a non-dynamically-sampled N.

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. Figure 2a shows one run of delta-debugging on a test of length 500. 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 from almost 100% 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 were performed).

IV. Implementation

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, 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 developer and tester intuitions about what should behave deterministically? Andrews et al. formalized a value-pool-based model [2] that is widely used in tools, including Randoop [26], where unit tests are *canonically* given as:

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 *at the granularity of value pool assignments*.

We implemented our approach as a modification of the TSTL [13] system, an open-source language and tool for property-based testing [18], [5] of Python code. TSTL has been used to detect (and usually fix) errors in a number of widely used Python libraries, the Python implementation itself, the Solidity compiler and a Solidity static analysis tool, and Mac OS. TSTL takes a harness (a definition of *what* to test, and what properties to check) and generates tests using random testing and various other approaches. TSTL is open source, and it and all examples used in this paper are available at: https://github.com/agroce/tstl. We implemented horizontal determinism checking (with both timing and process-based differentiation of executions), failure determinism checking, and a set of probabilistic delta-debugging strategies in TSTL, in order to examine their effectiveness, and to make them available to users performing Python library testing.

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. [2] 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 II-C, and transitions result from executing action code,—roughly, the array-canonical statements.

Because TSTL supports differential testing [13], 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). In TSTL, *visible value determinism* is thus based on comparing the values of *all non-opaque pool variables* after each test action. *Final State Determinism* simply performs the same comparison, but only

	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 I: Overview of experimental results

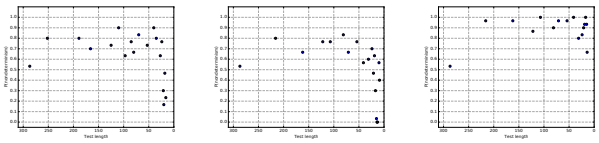
on the final values of all pool variables (or a set of designated pools) after a test has finished executing.

Because TSTL has an interface to AFL [32], 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 [32] 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.

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.

V. 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. Table I 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 imposed by nondeterminism checks varied,, 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. 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. Values are, unless otherwise noted, the mean of 30 runs.



(a) No modification (b) N=100 (2131s) (c) M=10, N=10

Fig. 3: Reducing a `redis-py` nondeterministic test

A. Parallel Sorting

Our first subject is an implementation of a parallel merge sort (https://gist.github.com/stephenmcd/39ded69946155930c347). When a sorting algorithm is implemented using concurrency, a key question to ask is whether the ordering of (equal under comparison) elements is consistent. TSTL can be used to show that a parallel sort implementation provides a consistent, deterministic ordering. For contrast, we also 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).

Our TSTL harness for parallel sorting 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 are sorted.. Both sorting algorithms always produce sorted output. The horizontal determinism detector, however, always generates a test case showing the `swapsort` behaving nondeterministically in less than 3 seconds, thus answering yes to **RQ1**. If we comment out the call to `swap_sort_parallel` and only check the parallel merge sort, TSTL finds no nondeterminism. This results in a slow-down of approximately 40% (**RQ2**).

Reducing a lengthy test showing nondeterminism of the swapsort usually takes less than two minutes (out of 10 trials, only one took four minutes). The resulting test will be usually be very compact, e.g. reducing the test from 56 to 8 steps.

B. The `redis-py` Library

The `redis-py` [19] module implements a (very) widely-used Python interface to the popular Redis in-memory database, cache, and message broker [28]. 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 them has proven difficult, as numerous Redis commands introduce nondeterministic behavior: thus the resulting tests are very often *flaky*. Using AFL, 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`, `expire`). However, issues such as whether the mechanism that allows a sequence of commands to be queued up and executed at once introduces nondeterminism are much more subtle.

Using the original TSTL harness, just over 20% of all `redis-py` regression tests (of length 200) generated were

flaky. We determined that the mean time to find all (known to us, over many experimental runs) sources of nondeterminism and produce minimal tests is 3.2 hours (**RQ1**).

We removed 11 calls from the `redis-py` 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. Resulting code coverage loss was minimal. Mean branch coverage was reduced by less than 5 branches (a less than 1% decrease). AFL stability was 56.26% with the harness allowing nondeterminism, but rose to 98.32% using the harness with nondeterministic operations removed. The overhead for 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.

As to **RQ3**, Figures 3a-3c show delta-debugging of a typical `redis-py` nondeterministic test, originally with 287 steps. The initial test behaves nondeterministically about half the time. Unmodified delta-debugging eventually 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 III-A, we see the same pattern. Simply making a predicate that "forces" the probability to remain high, with a large number of samples, does not work (Figure 3c) and requires over 2,000 seconds to produce a test with an even worse probability of nondeterminism. Using 10 samples and 10 replications, on the other hand, improves the probability of nondeterministic behavior, and produces a test with only 14 steps in just over 10 minutes.

C. *Berkeley* `datarray` *Inference Algorithms*

The `datarray` module [4] is a prototype implementation for numpy arrays with named axes, which also provides a set of algorithms for inference in Bayesian belief networks. An earlier version of these algorithms sometimes produced incorrect results due to dependence on the order of values in an iterator over a Python dictionary (see Section II-D1). Running our TSTL harness to test `datarray` consistently requires less than 10 seconds to find process-level nondeterminism in the `calc_marginals_sumproduct` function, answering **RQ1** again in the affirmative. Reducing this 60 step test to a 6-step test required another 92 seconds (**RQ3**) on average. The cost of checking for process nondeterminism is a mean 93% slowdown (**RQ2**).

D. *Vertical Determinism:* `pyfakefs`

The `pyfakefs` [20] 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. It is used in over 2,000 Python tests [20]. The TSTL harness for `pyfakefs` has been used to detect (and correct) over 80 faults.

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. Using `os.remove` to delete directories violates the Python `os` specification (and the POSIX standard). 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 with our failure determinism checks (an affirmative answer to **RQ1**). Moreover, the overhead for the check in a version of the code without the 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 state). Reducing the failing test to just 3 steps required less than a second (**RQ3**).

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` [11] to produce 2,350 mutants of the core file system code.

Non-differential testing, without failure determinism checks, consistently (in all runs) killed 872 mutants, a 37.1% kill rate. The differential harness, with an additional 60 seconds of testing time (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), consistently killed 1,148 mutants, improving the kill rate to 48.8%. Failure nondeterminism checking consistently added an additional 71 mutants to that total, *a 6% improvement even for this strong differential harness with a larger test budget*, nearly as large an improvement as the gain from using a full reference implementation. Using failure nondeterminism was the *only* way to push the mutation score above 50%. The kill rates are generally low because `universalmutator` includes many operations thatproduce hard-to-kill non-equivalent mutants not produced by other mutation tools. This is a strong affirmative answer to **RQ1**.

E. Threats to Validity

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. Because no other tools implement the kind of approach taken here, we were unable to perform a meaningful comparison with another nondeterminism detection tool.

VI. Related Work

The general topic of nondeterminism in software engineering has long been considered important [6], [34]. Gao et al. examined the question of how to make tests repeatable [7], in

the context of system-wide user interaction tests, focusing on systemic factors. Shi et. al [30] examined code that wrongly enforces a deterministic implementation.

The problem of test nondeterminism is closely related to the issue of flaky tests [23], [17], [27], [16]. How to handle flaky tests in practice is a major issue in Google-scale continuous testing [22]. Previous work on flaky tests has either focused on test inter-dependence [15], or large-scale empirical examination [17], [27]. Bell et al. proposed DeFlaker [3], 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 the test is likely flaky. iDFlakies [14] is a framework and dataset for flaky tests, but again focuses on whole-tests, and detecting flakiness by actually observing it.

We introduce the first variation of delta-debugging that properly handles probabilistic reduction criteria, in line with Harman and O'Hearn's proposal to simply accept that "All Tests Are Flaky" [12], and work with probabilistically failing tests.. Vertical/failure determinism is also a less-studied concept, and to our knowledge our formulation is novel. The kinds of errors that are exposed, however, are not new, e.g., faults related to the propagation of error conditions [29].

VII. 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, and leading to the costly problem of flaky tests [23], [16]. This paper proposes a formulation of types of nondeterminism, and a practical approach to using automated test generation to detect and understand nondeterminism in Python library code.

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] 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.

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

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

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

[6] 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.

[7] 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.

[8] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: Delta-debugging, even without bugs. *J. Software Testing, Verification, and Reliability*, 26(1):40–68, 2016.

[9] 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.

[10] Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2017.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] David R. MacIver. Hypothesis. http://hypothesis.works/, March 2013.

[19] Andy McCurdy. redis-py. https://github.com/andymccurdy/redis-py.

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

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

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

[29] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Programming Language Design and Implementation*, pages 270–280, 2009.

[30] 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.

[31] 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.

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

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

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